technische universität
dortmund

Master Thesis

Automated Vehicle Telematic Data Acquisition and Information Transformation

Thomas Richter

August 2016

Supervisors:
Prof. Dr. Jian-Jia Chen
Prof. Dr. Kristian Kersting

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Entwurfsautomatisierung für Eingebettete Systeme
http://ls12-www.cs.tu-dortmund.de

In cooperation with:
AVL SCHRICK GmbH
Dreherstraße 3-5
42899 Remscheid

Contents

Chapter I

Definition of Terms

Battery The *battery* stores energy generated from the ICE or, during regenerative braking, from the electric motor. Since the battery supplies power to the vehicle, it is larger and holds more energy than the batteries used in conventional vehicles.

BEV ***B**attery **e**lectric **v**ehicle* is a car architecture that uses energy stored in battery packs to propel the vehicle. Therefore it uses an electric motor which does not need fuel and emits no toxic gas at all.

Boost *Boost* is a hybrid function which is used to support the engine with additional torque. It is mostly used to assist the engine while launching or driving in low engine speed areas. Therefore it is possible to downsize the normal engine without performance losses, which results in a benefit regarding fuel consumption and toxic gas reduction.

BSG ***B**elt-driven **S**tarter **G**enerator* is a hybrid component in the electric power train. It uses an electric motor to contribute power to the internal combustion engine's crankshaft. It adds mild-hybrid capabilities like start-stop, regenerative braking and power assist.

Coastdown Contrary to coasting, *coastdown* is used to recuperate energy which can be used later on. Coastdown is possible when the powertrain is closed and no pedal is currently pressed. The energy which is generated during the decelerating process is stored in the battery.

Coasting *Coasting* was formerly known as *Start/Stop on the move*. It is a strategy to lessen fuel consumption. At the beginning, coasting was exclusively used for automatic transmission. The system detects the driver's easing of pressure on the gas pedal, decouples the engine from the transmission, and thus prevents the engine from consuming fuel. Drivers can already manually emulate this effect by disengaging the clutch on a downhill passage. Once a pedal is pressed, the engine is turned on again. The coasting functionality is also possible for manual transmission vehicles with an electronically controlled clutch. Effectively, the savings of fuel consumption will be equal to the fuel consumption in engine idle mode, which is about 2 %-6 % [5].

dbc ***d**ata **b**ase **CAN*** is a *Vector* file format which is used to store CAN related data messages. The DBC file type is primarily associated with *CANdb* by Vector Informatik GmbH. Each signal inside a CAN message decomposes to:

- name
- message id
- start-Bit and length
- byte-order (Intel/Motorola)
- data type (signed/unsigned/float)
- scaling factor
- unit string
- range of values
- default value
- comments

DCDC *DCDC-Converter* is used to regulate the voltage level for different components.

eClutch The ***e**lectronically controlled **clutch*** is a combination of hybrid powertrain and manual transmission. Depending on the vehicles configuration, the system can enable the coasting and recuperation functionality, which results in a reduced fuel consumption. Independently of the driver, the clutch decouples the engine from the transmission if the vehicle is no longer accelerating which results in an engine shutdown.

ECU ***E**ngine **C**ontrol **U**nit* is an electronic control unit that controls a series of actuators on an internal combustion engine to achieve optimal and safe engine performance.

eMotor The ***e**lectric **motor*** assists the gasoline engine when additional power is needed. It also acts as a generator: It converts energy from the engine while driving or from regenerative braking into electricity and stores it in the battery. It is also used to start the ICE instantly when needed.

eSC ***e**lectric **S**uper**c**harger* is a specific supercharger that uses an electric motor to pressurize the taken air. The air becomes more dense and is internally matched with more fuel, which results in short term generation of power.

Generator The *generator* converts mechanical energy from the engine or from regenerative braking into electricity which can be used by an electric motor or stored in the battery.

HCU ***H**ybrid **C**ontrol **U**nit* is an electronic control unit that controls a series of hybrid components to ensure their functionality and to derive the driver strategies.

HEV ***H**ybrid **e**lectric **v**ehicle* is a car architecture concept which uses two or more power sources. Its goals are to reduce fuel consumption as well as to limit the emission of toxic gas.

HMI ***H**uman **M**achine **I**nterface* is the communication and interaction interface for the connection between a human and a machine. In our case this is an Android application for a mobile device.

ICE ***I**nternal **C**ombustion **E**ngine* is the standard heat engine which is powered by either gasoline or diesel.

Load point moving ***L**oad **p**oint **m**oving* describes a process where the ICE is burdened with additional load from the electric motor in order to operate at a higher efficiency point to be able to cut the conversion losses. This surplus energy can be used to charge the battery or to propel the electric motor during boost phases.

Power-split device A *power-split device* is a gearbox connecting the ICE, generator and electric motor. It allows engine and motor to power the vehicle independently or together. Additionally, it allows the ICE to charge the batteries when overhead energy is created while driving with an optimal load point.

Recuperation *Recuperation* is the recovery of energy to load the battery system. The retrieved energy can then be used by the electric motor to support the engine while starting or accelerating. Recuperation is possible by braking or coasting down (*coast-down*) in a deceleration process or with load point moving while cruising.

SoC ***S**tate **o**f* **C**harge is defined as the available battery capacity expressed in a percentage value.

Start/Stop *Start/Stop* is a function that turns the engine off if the vehicle is standing still for more than a certain threshold time (used to be three seconds), e.g., while waiting for the traffic light to turn back to green. While this is working without further restrictions in a vehicle with automatic transmission, the driver of a manual transmission vehicle needs to open the powertrain by disengaging the clutch. Studies have shown that the usage of this feature reduces the fuel consumption by 2 to 3 % or in the case of vehicles which are mainly driven in the city by 4 % to 7 % [16].

UML ***U**nified* ***M**odeling* ***L**anguage* is a general-purpose modeling language which is developed to provide a standard way to visualize the design of a system.

Chapter 1

Introduction

The first question that comes into one's mind when talking about the electrification of automobile vehicles is the necessity. The motivation arguments which are leading to a change of technology are on the one hand air pollution and environmental impact, and on the other hand the diminishing petroleum resources over time. An ideal combustion process yields carbon dioxide (CO_2) and water (H_2O) while the real-world process additionally yields a lot of toxic byproducts. Furthermore, CO_2 is a leading greenhouse gas and 26 % of CO_2 emissions are caused by transportation [43]. Since the number of transportation vehicles is not going to decline soon, the emissions are neither. Additionally, the worlds petroleum resources are limited. The human beings are in need to find solutions that can decelerate the consumption and, as an ideal solution, replace it in its entirety. Hybrid electric vehicles are a first step into realizing such a deceleration process.

The concept of hybrid cars is to combine an internal combustion engine (ICE) and an electric motor that can assist the engine in certain cases - mostly acceleration. The batteries that power the electric motor are recharged automatically while driving. In a specific hybrid vehicle configuration it is also possible to propel the car up to 50 kilometers per hour (kph) in a pure electric mode without using classic combustion at all.

The idea of hybrid electric vehicles (HEVs) is surprisingly old. In 1900, Henri Pieper introduced a hybrid vehicle with two engines, one being a normal internal combustion engine and the other one being an electric motor [9]. The batteries could be charged when the vehicle was not accelerating or at a standstill. The electric motor assisted the engine in acceleration processes. The lack of electronic control mechanisms made the integration of the two power sources nearly impossible. Additionally, the motivation for this configuration was different from the current motivation for the electrification of vehicles. It mainly focused on the point to further accelerate a car instead of saving fuel and reducing toxic gas.

Because of the continuous growth and improvement in the technological sector, we can greatly benefit from device-to-device communication. In the current age of information

technology it is possible to transfer a lot of data between given endpoint devices wirelessly. The most common networks which are capable of providing such functionality are *Bluetooth* and *Wireless LAN*. Acquired vehicle data over these networks can be transformed and represented on nearly every device who supports the communication protocol. With real-time analysis and event detection it is possible to further increase the engineering efficiency. Changes on behalf of the hybrid control unit (HCU) can be done on the fly rather than after the test cycle is done.
The goal of this thesis is to provide real-time energy analysis features over a network connection with event detection and pattern matching to hybrid functions. These results should be displayed on a mobile Android device, so the engineering process can be done online in the car.

1.1 Description of the studied problem

Real-time analysis tools are not exactly new, but within the given scope of this thesis, there is - at least not publicly available - no such tool for any mobile device. Since the application is intended to be used on various vehicles, it needs to be modeled in a generic way.
This illustrates the first problem. Most automobile manufacturers are deploying their data messages over the CAN bus in a unique way which means that there is no standard yet. There are very few cases of them being absolutely equal. Therefore, a solution has to be found in order to map existing vehicle signals with the ones which are needed in the application.
Another difficulty is the computation and persistence of all the data values which are needed to classify the state of the car and to compute the analysis results in real time.
The classification process is not trivial as well. The state of the car has to be classified periodically and this classification has to be done in real time with as little time overhead as possible. The states/events of each car are not necessary equal as well, since they are dependent on the vehicle setup and the build-in components. Consequently, the task is to develop a model for each state/event with its restrictions in mind.

Constraints There are multiple types of constraints which we need to deal with:

- Vehicle constraint: Different CAN bus specifications (*.dbc files) and hybrid powertrain specifications. Maybe not all matchings and/or visualizations are possible. Selection of suitable signals.
- CAN bus constraint: Bus load, cycle time
- Memory Consumption: Logging/Maintaining of the data, Android heap, pattern matching computation in cycle time, calculations for the graphical interfaces

- Real-time constraint: Cycle time, amount of states/events, amount of signals

1.2 Structure

In the following, a brief overview of the structure of this thesis is given. All necessary terminologies which are used throughout this thesis are explained in Chapter I, while Chapter 2 lists the aims and objectives. Related research and the state of the art is presented in Chapter 3, whereas Chapter 4 elucidates our used technologies and gives a short overview of different hybrid architectures. The used design and implementation to achieve the aim and objectives are defined in Chapter 5, while Chapter 6 shows first analysis results and a quick summary of the application functionality on the mobile device. Chapter 7 concludes the thesis and points out which goals and objectives were reached. Chapter 8 illustrates additional ways to continue the thesis outcome and some general possible improvements.

Chapter 2

Aims and objectives

The main aim of this thesis is the development and design of a solution in which the energy behavior of a vehicle can be displayed on a mobile device. This includes retrieving the vehicle data over a telematic device, transforming them in a suitable manner, classifying the state of the vehicle in each time step, computing the necessary information out of the retrieved values with possible post-processing and representing the final results on a mobile device screen.

In a first step, the data need to be retrieved over a telematic device because it is not intended to use any wiring between the vehicle and the mobile device. The device should be able to either transmit its data over a Bluetooth or GPRS connection. Since vehicle specific data are found on its own CAN bus, the device needs to be able to retrieve data from the CAN bus and transfer these over a network protocol. Therefore, the device and its connection technology have to be selected initially.

Since the analysis of the energy behavior should work on a wide range of vehicles and the CAN specifications are different for each one, it is necessary to somehow specify which signal data are available in the current setup and where to find them exactly. With a given database of CAN messages (dbc), we need to select which signals are necessary for the computation and map them accordingly. Considering this, it would be great to have an automatic way to select messages or signals out of a dbc-file and then generate interchangeable JAVA-code for the signal mapping.

Once the signal mapping and the data retrieval are finished, the raw data need to be transformed into actual usable values. Since the raw data endianness can be different for each message, we need to distinguish between big endian and small endian byte order as well as signed and unsigned values. Therefore, suitable functions to transform the data into usable values have to be designed and implemented.

To classify the vehicle's state and to compute the necessary values for the graphical representation on the mobile device, a suitable listener and a proper storage structure have to be implemented as well as a transferring mechanism between Android activities.

The last objective is the final representation of the computed values on a mobile device with possible post-processing. In order to achieve that, a preferable existing API has to be chosen for the rendering and painting on the screen.

Chapter 3

Related research and state of the art

This chapter presents the related literature and research topics as well as the state of the art of in-vehicle human-machine interfaces (HMI).
Yi et al. [45] present a system which collects mobile sensor data and monitor them on an Android smartphone. The application is also able to stream received data over a wireless Internet connection. The Bluetooth connection is established with the Near Field Communication (NFC) technology. The application was developed in order to show body sensor signals as a healthcare tool. Furthermore, a Java based server is used to analyze the communication as well as data storage mechanism. Tahat et al. [41] introduce a system which monitors and tracks a vehicle by transmitting the obtained data to a mobile device via Bluetooth. This is done by collecting live data from the engines control unit utilizing the on-board diagnostic system (OBD). The Bluetooth module transmits and communicates with the Android mobile device over an interface which is also able to transmit the recorded data to a server using a cellular connection. Zaldivar et al. [46] propose an Android-based application that monitors the vehicle through an on-board diagnostics (OBD-II) interface in order to detect accidents and prepare counter measures. The resulting application is able to get data over a Bluetooth communication and designs an approach to detect accidents. Once an accident is detected the application is able to prepare counter measures, like sending a SMS or e-mail to pre-defined destinations. Spelta et al. [39] have implemented a system for a vehicle-to-driver and vehicle-to-environment communication, based on a smartphone core and Bluetooth communication. Since the authors focus on motorcycles, they have equipped them with an embedded CAN-Bluetooth converter that is interfaced with the smartphone, which acts as a gateway towards an audio helmet and a webserver. Al-Ani [1] has designed and developed an integrated system which provides infotainment services such as GPS-based navigation, road and traffic information as well as multimedia content. The Android system is used as a developing platform to implement an Android-based In-vehicle Infotainment (AIVI) system. It is able to display OBD content and can establish a Bluetooth connection with the smartphone to initiate calls or voice messages.

Another integrated HMI system solution is *Android Auto* [21]. It connects the smartphone to the vehicle's infotainment system and is able to use applications which are optimized for Android Auto. As of now, only a dozen applications, mostly in the communication and multimedia area, are certified to run on the infotainment system. It runs on an Android operating system which makes it possible to communicate with remote devices over Bluetooth and Wireless Lan protocols.

The *DOK-ING MyHMI* framework [15] however, offers possibilities for a HMI advancement in the domains of communication, data acquisition and processing, and visualization. It is a Java based application that can run on different platforms, including Android. It has full support of OpenGL as well as OpenGL ES and has the ability to integrate a data processing logic with different protocol stacks for the data specification and communication. Another application which provides analysis functions over a Bluetooth or cellular connection, is called *Torque* [25]. It needs an OBD-II adapter to get the vehicle information which are then monitored on a mobile device. Further analysis features like average fuel consumption or driving range are possible if the data are post-processed on a server or a device at home.

There are no specific studies dealing with an energy analysis for hybrid vehicles on a mobile device. However, Hofman and van Druten [26] have investigated the possibilities of energy and fuel savings in different hybrid configurations with regard to the conversion efficiencies in the hybrid function *regenerative braking*. The simulation results show that the energy recovery efficiency is strongly determined by the motor/generator efficiencies, sizes and vehicle mass. A fuel saving increase around 21 % is achieved in the final design while driving a New European Driving Cycle (NEDC).

The literate research has shown that most research groups use the OBD interface to get vehicle data. However, the usage of an OBD interface might be good for a conventional vehicle, but is not sufficient when operating with prototyped hybrid vehicles with a separate HCU CAN bus. Since it is not possible to get the HCU bus data over the normal OBD interface, this thesis uses an approach with the CAN interface and tries to narrow the gap between monitoring signals and providing energy behavior analysis features for hybrid vehicles.

Chapter 4

Fundamentals

This chapter introduces the basic technology concepts which are used to achieve the targeted goal. The main technologies are the Controller Area Network (CAN) bus, the CAN-link Bluetooth module, hybrid vehicles and the Android environment. Starting with a short summary of the CAN bus structure and the operating principle, the CAN message frame is explained. Afterwards, the chosen network device is defined and a short overview over the specifications is given. The next section deals with a general approach to elucidate hybrid electronic vehicles in an appropriate depth. This is necessary to define the possible of vehicles states later on. The last section introduces Android as the chosen operating system on the mobile device.

4.1 CAN bus

The development of the ***C**ontroller **A**rea **N**etwork* (CAN) started in the early 80th at Robert Bosch GmbH (Gerlingen, Germany). It was designed to allow electronic devices to communicate with each other without the need of an external computer. The message-based protocol was originally designed for multiplex electric wiring within automobiles [31], but is - as of now - also used in various other contexts.
CAN is composed of *nodes* which can range from a simple single I/O device to an embedded computer. The CAN network is only able to communicate if there are at least two nodes on the bus. The ISO specification states that a node needs to consist of [14, 20, 37]:

CPU Decides which messages to send and what to do with incoming messages.

CAN controller Fetches/Sends the bits serially until the whole message is received/-transmitted and therefore ready for the CPU to process.

Transceiver Converts Can bus levels while transmitting/receiving.

Each node can send and receive messages, but transmitting and receiving at the same time is not possible. The different message layouts and standards are described in the Section

4.1.2. In contrast to networks such as USB or Ethernet, CAN broadcasts many short messages over the whole network instead of sending big chunks of data from node to node. This however, guarantees data consistency for every node in the system [6, 33].

4.1.1 CAN protocol layers

Like many other networking protocols, the CAN protocol can be decomposed into abstraction levels. The *object layer* deals with message filtering as well as message and status handling, whereas the *transfer layer* receives messages from the *physical layer* and transmits them to the object layer. The transfer layer is responsible for error detection, fault confinement, arbitration, acknowledgment, message framing and synchronization [28]. ISO 11898-2:2003 norm defines the electrical aspects of the physical layer. However, the mechanical aspects have yet to be defined formerly. Nonetheless, there exists a de facto standard of a mechanical implementation, as following [28]:

- pin 2: CAN-Low (CAN-)
- pin 3: GND (Ground)
- pin 7: CAN-High (CAN+)
- pin 9: CAN V+ (Power)

4.1.2 CAN frames

A CAN network can either be used with the base frame format (CAN 2.0A/CAN 2.0B) or with the extended frame format (CAN 2.0B). The main difference is that the identifier length is 29 bits for the extended frame specification instead of 11 bits for the base one. To distinguish between an extended frame message and a base frame message the identifier extension bit can be used. A recessive bit indicates that it is an extended frame format while a dominant bit indicates a base frame format. CAN controllers which support extended frame format messages also support base frame format messages. However, controllers that only support the base frame format do not support the extended frame format. CAN can be divided into four different frame types which are explained in the following sections [19, 28, 37]:

- Data Frame: Includes node data for transmission.
- Remote Frame: Requesting the transmission of a specific ID (searching).
- Error Frame: Any node which detects an error can send an Error Frame.
- Overload Frame: Used to infuse a delay between data and Remote Frame.

Data frame

The data frame is the only frame which transmits actual data. It can send data in two different frame formats, one being the base frame format shown in Figure 4.1 and the other being the extended frame format (Figure 4.2). The only difference between those two formats is the ID length specification. While the extended frame uses a 29 bit identifier, the base format uses a 11 bit identifier. The data field however, is eight byte long regardless the frame format.

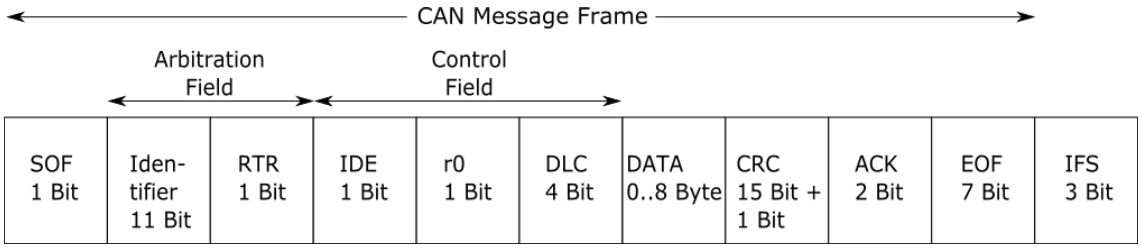


Figure 4.1: Structure of the CAN message frame with a CAN 2.0A standard frame.

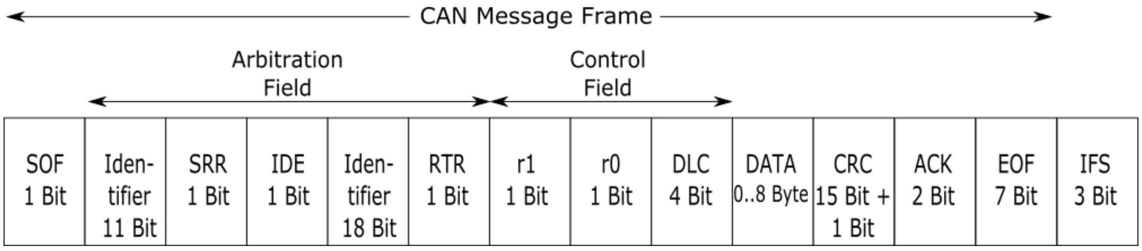


Figure 4.2: Structure of the CAN Message frame with a CAN 2.0B extended frame.

A brief description of the data frame elements is given below [28, 37]:

SOF The ***S**tart **o**f **f**rame* indicates the start of a new frame transmission. It is used to synchronize the nodes on the bus after being idle.

Identifier The *Identifier* is a unique representation of the CAN message and characterizes its name and priority. A lower value means a higher priority on the CAN bus. During the transmission of this field, the transmitter is always checking if there is another message with higher priority.

RTR The ***r**emote **t**ransmission* **r**equest bit is dominant (set to zero) when information is required. All nodes in the system are able to receive the request, but only the one with the matching identifier can process it. The response data is also visible on the whole bus and can be used by each participant.

SRR The ***s**ubstitute **r**emote **r**equest* replaces the *RTR* bit in an extended message format and is recessive. In this case the *IDE* bit is also recessive signaling that there is an extended identifier.

IDE The ***id**entifier **e**xtension* indicates whether the CAN identifier is in a standard or extended format. A dominant bit means that the identifier is a standard 11 bit value.

r0,r1 Reserved bits, with no possible future usage. r0 is used for the basic frame specification, whereas both r0 and r1 are necessary for the extended frame protocol.

DLC The ***d**ata **l**ength **c**ode* is a 4-bit value which denotes the number of bytes of data being transmitted.

Data Contains the raw application data and can take up to 64 bit at maximum.

CRC ***C**ycle **r**edundancy **c**heck* denotes the error code for previous information. The CRC checksum is used for error detection. The CRC delimiter must be recessive.

ACK The ***ack**nowledgment* field includes feedback from other participants at the correct receipt of the message. The ACK delimiter must be recessive.

EOF ***E**nd **o**f **f**rame* indicates the end of the data frame and transmission. It is composed of five recessive bits.

IFS ***I**nter **F**rame **S**pace* denotes the period for transferring a correctly received message.

Remote frame

The remote frame contains no data elements, but it transmits a remote request to all nodes on the bus to send a data frame with the requested identifier. If a node has the answer to the request it will transmit it to the bus and every listener can access it. It can be abstracted with a simple analog: A bus node asks the question "How fast am I driving right now?" and a node which holds the answer responds to everyone "40 kph" [28, 37].

Error frame

An error frame is generated by any node that detects a bus error. It consists of two fields, an error flag field followed by an error delimiter field. The error delimiter consists of eight recessive bits and allows the bus nodes to restart the bus communications cleanly after an error. There are, however, two forms of error flag fields. Their form depends on the *error status* of the node that detects the error. If an *error-active* node detects a bus error, the node interrupts the transmission of the current message by generating an *active error flag.* After completing the error frame, the bus activity returns to normal and the interrupted node attempts to resend the aborted message. If an *error-passive* node detects a bus error, the node transmits a *passive error flag.* When the bus master node generates an *passive error flag* then this may cause other nodes to generate error frames due to the resulting bit stuffing violation [28, 37].

Overload frame

The overload frame provides a forced pause between data and remote frames. It consists of two fields, an overload flag followed by an overload delimiter. An overload may occur in two cases. Firstly, the electronic of the receiver needs a delay before transmitting the next data or remote frame, e.g., because of a full buffer. Secondly, a delay is needed while identifying a dominant bit during the transmission of a message procedure while it is not able receive new content [28, 37].

4.2 Bluetooth

Bluetooth is one of the standards for exchanging data wirelessly over a short distance. It was invented by telecom vendor *Ericson* (Stockholm, Sweden) in 1994. The physical range from a private network with Bluetooth devices is typically less than 10 m, but can go up to 100 m in certain cases [40]. The two most prevalent implementations of the Bluetooth specification are the *Basic Rate/Enhanced Data Rate (BR/EDR)* and *Bluetooth with low energy (LE)*. Each implementation uses different hardware to achieve its goals. There are dual-mode chipsets to enable both use-cases to the application. Following, a short description of the differences between the Bluetooth modes is given [10]:

BR/EDR Establishes a short-range, continuous wireless communication.

LE Allows for short bursts of long-range radio connection, making it valuable for application that do not require a continuous connection and depends on longer battery life.

Dual-Mode A device with a dual-mode chipset is able to connect with a BR/EDR as well as a LE device.

4.2.1 RM CANlink Bluetooth

The *RM CANlink Bluetooth* module manufactured by *Proemion* (Fulda, Germany) was chosen as the communication device between the car and the mobile application and is shown in Figure 4.3. It is designed to transfer CAN data wirelessly by using the standard Bluetooth protocol [36].

Figure 4.3: Chosen Bluetooth module for the car to mobile application communication [36].

The device can operate in two different ways. On the one hand it can serve as a CAN to CAN bridge to transfer CAN data between the components. On the other hand it functions as a single CAN to Bluetooth interface, transmitting data to mobile Bluetooth devices such as PCs or Tablets for CAN data monitoring and analysis.
The CAN baud rates in Table 4.1 should be used as a reference regarding the maximum cable length of the bus line. The Bluetooth module in the test-vehicle uses a baud rate of 500 kbit/s which is the standard for most automotive CAN buses.

CAN Baud Rate	**Maximum bus cable length**
1 Mbit/s	25 m
800 kbit/s	50 m
500 kbit/s	100 m
250 kbit/s	250 m
125 kbit/s	500 m
50 kbit/s	1000 m

Table 4.1: Dependency between baud rate and cable length for CAN transmission [8].

4.3 Hybrid vehicles

This section elucidates the concept of hybrid automobiles and shows the most common system architectures and configurations.
A vehicle is generally called a hybrid vehicle if it utilizes more than one energy source to achieve propulsion. Essentially, this means a hybrid vehicle is composed of a traditional internal-combustion engine (ICE) as well as one or more electric motors and a battery pack. The dimension of the battery pack depends on the vehicles hybrid architecture. Hybrids are not to be confused with electric cars. The latter does not use gasoline-burning machines to move forward at all, but relies solely on an electric motor and a big battery pack. Hybrids however, utilize the electric components to collect and reuse energy that would go to waste in a common car. The gained energy is then used to support the internal-combustion engine with the electric motor, which decreases the fuel consumption by 10 % to 15 % [12, 13, 17].

4.3.1 E-Drive Configurations

The main challenge for hybrid electric vehicles design is on how to manage multiple energy sources, which are highly dependent on driving cycles, ICE sizing, battery sizing, motor sizing, and battery management. The goal is to maximize the efficiency of the whole driving system, while providing reasonable performance in terms of acceleration, range, dynamic response and comfortness. Hybrid electric vehicles can roughly be divided into three main configurations: *Series*, *Parallel* and *Series-parallel*. A short description of these configurations can be found in the succeeding paragraphs.

Series hybrid

A series hybrid system can be regarded as an extension of a BEV by adding an ICE to charge the battery in order to extend the driving range, which is one of the major drawbacks of pure BEVs. Normally, the ICE is not used in the normal city driving mode, mostly because it reduces the emission of toxic gas. In this case, the electric motor is propelling the vehicle single-handedly while draining power from the battery. Because this results in a considerable loss of energy, the car can recuperate during downhill slopes or regenerative braking. This further enhances the fuel economy as well as supports the braking system. The ICE is trying to operate at its most optimal load point to achieve the minimum energy conversion loss, so that the output power level efficiency is maximal. The resulting power overhead will be used to charge the battery when the power demand of the electric motor is lower than the generator output power. But if the power demand of the electric motor is higher than the output power of the generator, the generator power as well as additional power from the battery will be used to drive the motor. Therefore, the ICE is either off or operates at its highest efficiency point, and hence greatly improves the fuel economy of the vehicle as compared to conventional ICE vehicles [7, 17, 18, 22].

Parallel hybrid

Parallel hybrid systems have both an internal combustion engine (ICE) and a coupled electric motor. In contrast to the series hybrid configuration, the ICE is not solely used to provide engine to charge the battery, but to propel the vehicle with the support of an electric motor on the same transmission. When the demand power from the transmission is higher than the power output of the ICE, the electric motor will be turned on and contributes the difference in power. When the demand of the transmission is lower than the power output of the ICE, the electric motor will become a generator and the overhead power will be used to charge the battery set. In this way, this can be regarded as an electric-assisted ICE vehicle, where the electric motor acts as a supporter to smoothen the occurred loading to the ICE. The ICE can maintain its running operation almost fully on the most efficient loading point, which enables the vehicle to get the maximum efficiency level of power output, resulting in an improvement of fuel consumption. Since the power outputs from both the ICE and the electric motor are additive, downsizing of both components is possible to gain a reduce on additional cost and a fuel consumption profit while getting a similar performance. Because parallel hybrids use regenerative braking as well as coastdown and can additionally charge the battery with overhead power from using the ICE in an optimal load point, they are the most efficient vehicles for the urban traffic where a lot of so called "stop-and-go" situations occur [7,18,22].

Series-parallel hybrid

Series-parallel hybrids or power-split hybrids use power split devices to provide power to the wheels either mechanically or electrically. The main principle of this configuration is the decoupling of the power supplied by the engine from the power demanded by the driver. The ICE offers low torque output at low revolutions per minute (rpm) and thus a larger sized engine is needed in conventional gasoline vehicles to achieve acceleration from standstill. However, an electric motor has high torque output at low rpms and complements the weakness of an ICE. Adding an additional mechanical linkage and a generator between the ICE and the converter in a parallel HEV allows both series and parallel operations of the ICE and the electric motor. Obviously, this structure allows more flexible operations, but it is also complicated in the mechanical structure and costly in price. The popular Toyota Prius adopted this configuration [7,17,18,22].

Electric motor positioning

An electric motor is used in all hybrid vehicle configurations. However, there are five possible positions on where to integrate it in the vehicles power-train. Figure 4.4 depicts all possible positions of an electric motor in the power-train.

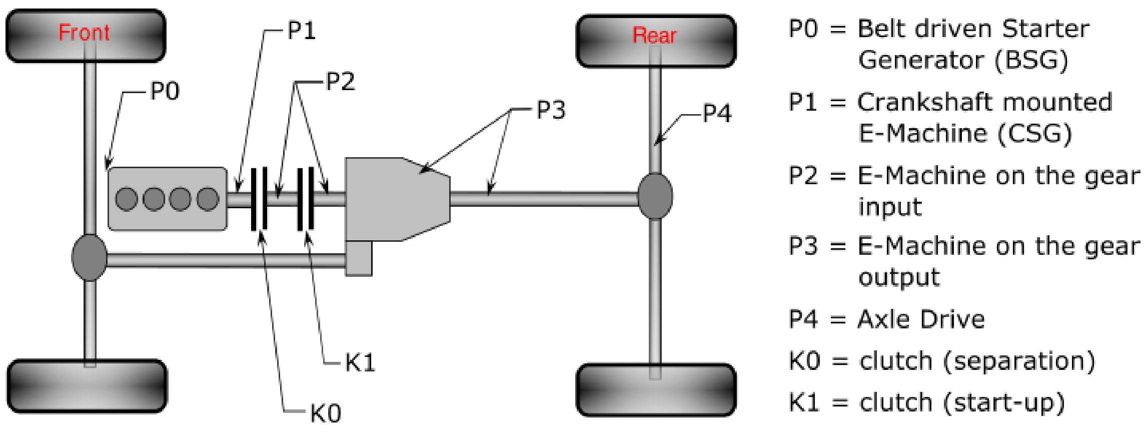

Front
P1
P0
P2
P3
Rear
P4
K1
K0
P0 = Belt driven Starter Generator (BSG)
P1 = Crankshaft mounted E-Machine (CSG)
P2 = E-Machine on the gear input
P3 = E-Machine on the gear output
P4 = Axle Drive
K0 = clutch (separation)
K1 = clutch (start-up)

Figure 4.4: Possible configurations of hybrid eMotor components inside the vehicles power-train.

P0 uses a classic belt-driven starter engine (BSG) which operates on the engines belt drive, but therefore is limited in its power output. An advantage lies in its low integration costs compared to the other solutions. *P1* uses a crankshaft-mounted integrated starter generator (CSG/CISG) which is integrated between the engine and transmission. The power output is mostly limited by the size of the starter generator, but has higher costs due to the difficult mechanical integration process. If the ICE is decoupled by a secondary clutch, it is possible to recuperate more energy than with a BSG, because losses due to drag torque are being omitted. If the electric motor is not directly coupled to the ICE, but located on the gear input with a secondary clutch, it is a *P2* configuration. Because the ICE is decoupled from the remaining power-train, regenerative braking and electric driving are more efficient. *P3* uses the electric motor on the gear output and is therefore not restricted by the transmission. This enables the motor to work with higher speeds. A *P4* configuration means that the electric motor is fully integrated into an axle and supplies either front or rear wheels with power. This enables the system to drive with an ICE on the front wheels while using the axle drive for the rear wheels. Unfortunately, the integration process is complex and the control unit of this configuration is not trivial either [22,24,44].

4.3.2 Hybrid Levels

While there are three main types of hybrid power-train concepts, there are different degrees of hybridization as well. This section introduces the different types and tries to deepen the understanding which is needed for the definition of the hybrid states for the final implementation. Table 4.2 shows all current degrees of hybridization.

Functions	**Micro & Micro-Mild Hybrid**	**Mild Hybrid**	**Full Hybrid**	**Plug-In Hybrid (PHEV)**	**EV**
Start & Stop	✓	✓	✓	✓	✓
Regenerative braking	✓(micro-mild only)	✓	✓	✓	✓
Additional electric power for a few seconds	✓(micro-mild only)	✓	✓	✓	✓
Electric power for low distance (city)			✓	✓	✓
Electric power for long distance + recharge				✓	✓
Energy savings	5-10 % (up to 25 % in the city)	10-25 %	25-40 %	50-100 %	100 %
Electric power	1.5-10 kW	5-20 kW	30-75 kW	70-100 kW	30-100 kW

Table 4.2: Hybrid Level functionality and power characteristics [11].

The micro hybrid offers only a small advantage over the conventional powered vehicle. It adds a start-stop system which shuts down the engine while standstill time and powers it again when the clutch or brake is pressed. If the feature of regenerative braking or boosting power, which means an additional amount of torque for a short period of time, is added, we talk about a mild hybrid level of hybridization. Full hybrids are different in the way

that they offer electric power for low distance driving which is greatly influenced by the dimension of the included battery pack. A plug-in hybrid adds an additional feature of driving longer ranges exclusively with an electric motor by greatly increasing the battery pack as well as adding an external charging method to recharge the battery over night. The full electric vehicle does not use a conventional ICE anymore and propels the vehicle solely with electric power sources.
The fuel consumption savings are therefore ranging from small (micro hybrid) to somewhat high (full hybrid), while the plug-in hybrid is able to have no fuel consumption at all. Because a pure electric vehicle has no build in ICE anymore, it does not need fuel either and does not emit any toxic gas with the drawback of a longer recharging process and limited range.

Micro Hybrid

Micro hybrid is a term given to vehicles that uses some kind of start-stop mechanism to automatically shut down the engine when idling, e.g., when waiting at a traffic light. In general, micro-hybrid vehicles cannot be considered as real hybrids because they do not use two different power sources for either support or propulsion. Nevertheless, the fuel consumption is less than in a conventional gasoline-only vehicle. Figure 4.5 shows the general structure of a micro-hybrid vehicle. It is propelled by an ICE and has an integrated starter generator (ISG) to realize the start-stop system. The ISG is connected to a battery which offers the energy to start the vehicle from standstill [17].

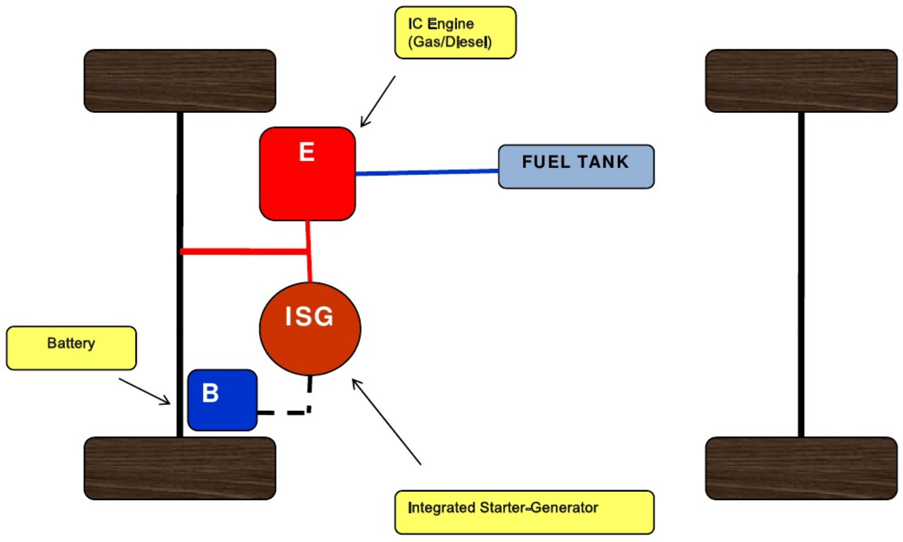


Figure 4.5: Power-train overview of a micro-hybrid vehicle system. The ICE is used for propelling the vehicle while the starter generator is used to make a start-stop system when in idle position [35].

Figure 4.6 elucidates the power flows in the different operating modes, which are possible for a micro-hybrid vehicle. In image *a)* the ICE supplies power to the wheels while also charging the battery through the generator. Additional energy can be gained through regenerative braking as shown in image *b)*. Kinetic energy of the braking process is recuperated and directed to charge the battery through the generator. The ability of regenerative braking is however not implemented in all micro-hybrid systems.

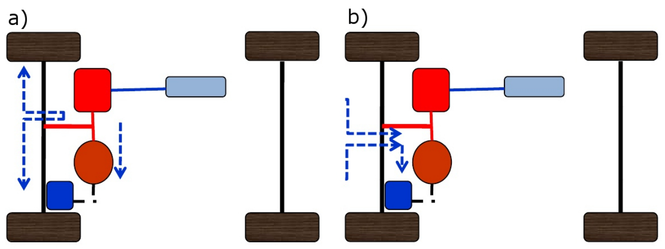


Figure 4.6: Operating modes and energy/power flows of a micro-hybrid vehicle structure: a) Driving with normal ICE while charging the battery; integrated starter generator used for starting from stop/idle. b) Braking - Regenerative braking to charge the battery. [35]

Mild Hybrid

A mild-hybrid architecture has a higher degree of hybrid feature utilization. Typically, a mild-hybrid vehicle uses a parallel system configuration which makes it possible to use start-stop systems and some form of regenerative braking. Depending on batteries dimension, a boost feature to support the engine with additional torque at low revolutions per minute is available. Unlike full- or plug-in-hybrids they can not solely propel the vehicle with an electric motor. Mild hybrids are sometimes called power assist hybrids as they use the ICE for primary power, with a torque-boosting electric motor connected to a mostly conventional power train. The electric motor, which can be mounted in different locations (see Section 4.3.1), is essentially a very large starter motor, which operates not only when the engine needs to be turned over, but also when the driver demands additional torque while accelerating. The electric motor can also be used to restart the combustion engine, deriving the same benefits from shutting down the main engine at idle, while the enhanced battery system is used to power the normal system, e.g., radio or navigation systems [11]. Figure 4.7 shows the general architecture of a mild-hybrid vehicle. The ICE supplies power to the wheels for normal propulsion. The integrated starter generator can be placed at different locations in the vehicles power-train (see Section 4.3.1), which results in different hybrid capabilities. The battery pack size is larger compared to the micro hybrid because the system supports boosting to support the ICE in situations of low revolutions per minute [17].

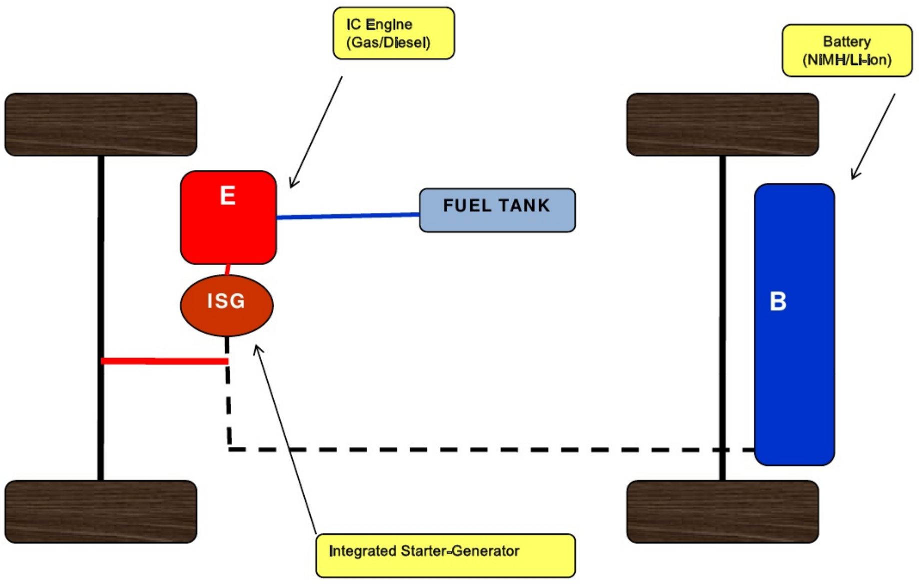


Figure 4.7: Overall structure of a mild-hybrid configuration [35].

Figure 4.8 depicts the energy and power flow of the possible operating phases of a mild-hybrid vehicle.

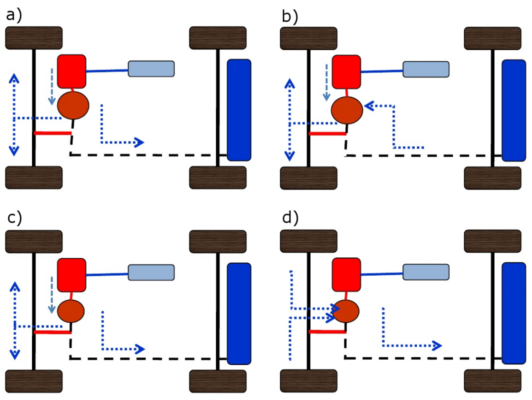


Figure 4.8: Operating modes and energy/power flows of a mild-hybrid vehicle structure: a) Starting/Moving from standstill - ICE supplies power to wheels as well as charges battery. b) Overtaking and accelerating - ICE as well as electric motor supplying power to wheels. c) Cruising - ICE supplies power to wheels as well as charges battery. d) Braking - Braking energy is recuperated, converted to current and charges the battery [35].

Phase a) describes the starting or moving of the vehicle from standstill, in which the ICE is used to propel the car. Depending on the location of the starter generator, it can support the ICE with additional torque. In phase b) the generator supplies power through the battery and supports the propulsion from the ICE by adding additional torque. Phase c) shows the power and energy flow while driving normally. When the engine operates at an efficient load point, the loss of energy is minimal compared to a normal operation mode. This overhead on energy can then be used to charge the battery. In regenerative braking (phase d), the electric motor is reversed so that, instead of using electricity to turn the wheels, the rotating wheels turn the motor and create electricity. Using energy from the wheels to turn the motor slows the vehicle down.

Full Hybrid

Full-hybrid vehicles are able to run in different propelling modes. An ICE as well as an electric motor are coupled with a power-split device. Therefore both the engine and the motor are able to independently propel the vehicle, but can also add their torque to move the vehicle together. A large battery pack is needed in order to enable electric driving. Full hybrid vehicles have a split power path that allows more flexibility in the power-train by converting mechanical and electrical power. They can use regenerative braking and are able to reduce fuel consumption by operating the engine at an optimal load point [17]. An overview of the structure can be seen in Figure 4.9.

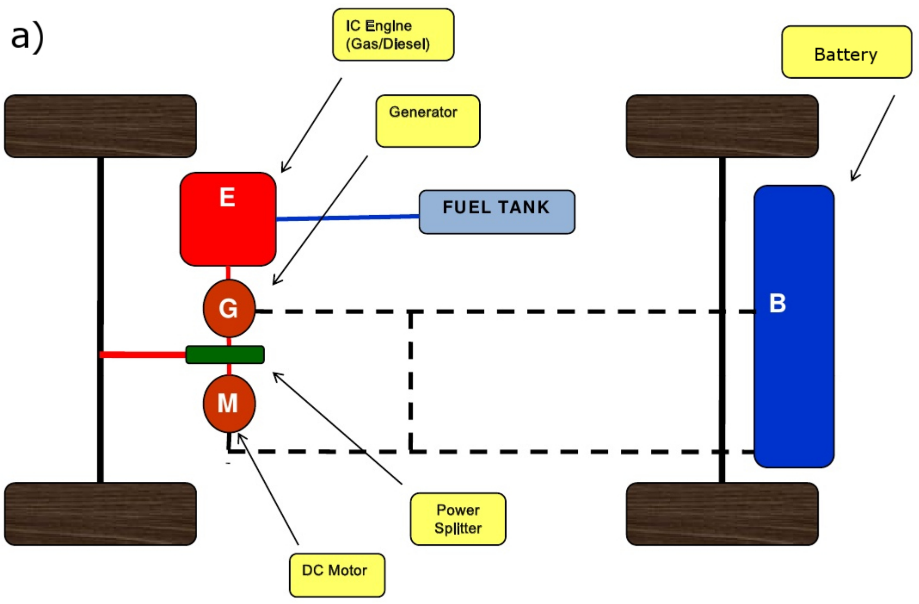


Figure 4.9: An overview of the full-hybrid vehicle power-train structure. An ICE as well as an electric motor are coupled with a power-split device. Therefore both the engine and the motor are able to independently propel the vehicle, but can also add their torque to move the car together. The generator can either be used to charge the battery or to give input to the electric motor [35].

Figure 4.10 shows the operating phases of a full-hybrid vehicle during a driving session. Phase a) elucidates the starting process or moving from standstill, where the electric motor supplies the power to the wheels, propelling the car alone. In phase b) the vehicle is driving at highway speed and the ICE supplies power to the wheels as well as generates power over the generator when operating in an optimal load point. This power can either be used for the electric motor or to charge the battery. Driving at low speed, e.g., in the urban environment, is illustrated in phase c). The electric motor drains energy from the battery pack and propels the vehicle while the ICE is off. This reduces the fuel consumption as well as the emission of toxic gas. In case the battery pack does not provide enough energy, the ICE is started and charges the battery. Phase d) depicts the acceleration and overtaking process. While additional torque is requested for a short amount of time, the electric motor draws energy from the battery to support the ICE with additional torque [35].

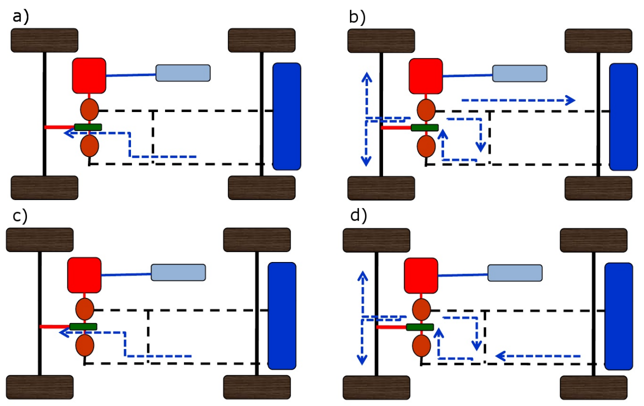


Figure 4.10: Operating modes and energy/power flows of a full-hybrid vehicle structure: a) Starting/Moving from standstill - Electric motor supplies power to wheels. b) Cruising or highway driving - ICE supplies power to wheels as well as charges the battery. c) Low speed or city driving - Electric motor supplies power to wheels. d) Overtaking and accelerating - In addition to engine power, power is drawn from battery through the electric motor [35].

Plug-in Hybrid

Plug-in hybrid vehicles are full hybrids with the possibility to charge the vehicle with an external power source. Like the full hybrid, they can run both the electric motor and ICE independently as well as together. A difference is the larger dimensioned battery pack to obtain greater driving ranges. Because the battery can also be charged through the ICE when cruising, the driving range is larger than that of a full hybrid. Figure 4.11 shows the power-train overview of the plug-in hybrid vehicle. The car can be propelled by using the electric or the ICE or a combination of those two. The generator is able to charge the battery or to supply electricity to the electric motor while cruising. The motor is able to provide energy for low-speed driving, e.g., in the urban environment, but will meet its limits on driving at highway speed. Therefore the ICE is used for these situations. However, during acceleration and overtaking processes the electric motor supplies additional torque to the ICE [11, 35].

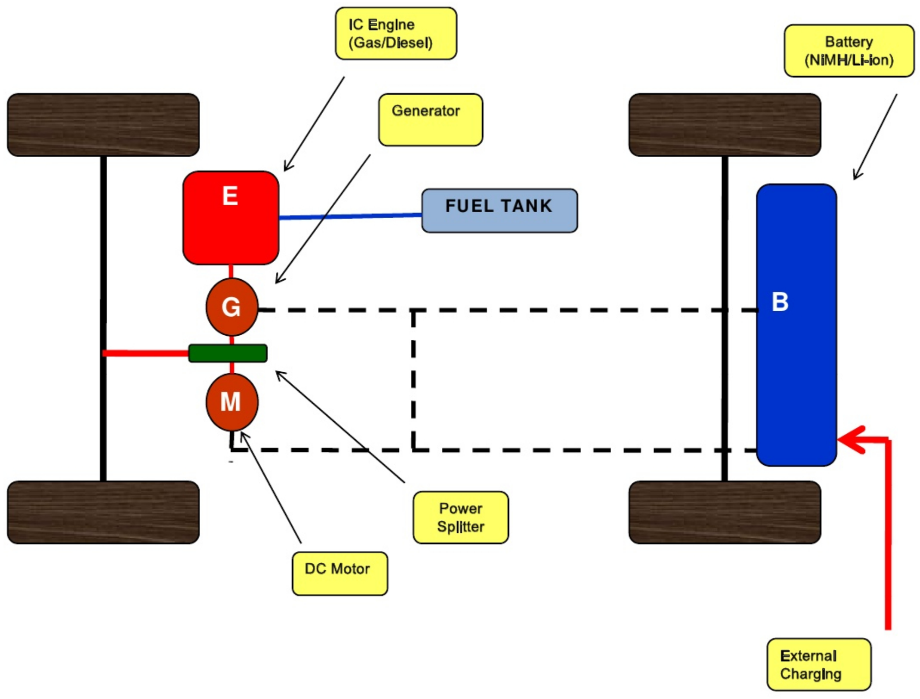


Figure 4.11: Power-train architecture of a plug-in hybrid vehicle. The ICE or electric motor are used for propulsion; either in independent modes or in conjunction. The generator can supply overhead energy to charge the battery. With an external charge mechanism the battery can also be loaded over night, resulting in a higher driving range [35].

Figure 4.12 elucidates the power and energy flows of the plug-in hybrid. In phase a) the starting process or moving from standstill is presented, where the electric motor provides all the necessary power to propel the car. Phase b) shows cruising, which is driving without accelerating, at highway speed, where the ICE is able to charge the battery and provide torque to the wheels at the same time. Regenerative braking directs the recuperated energy to charge the battery, which is depicted in phase c). While driving at low speed the electric motor is providing the power to the wheels. This can be seen in phase d). In phase e) the acceleration or overtaking process is shown, where the electric motor supplies the ICE with additional power to achieve a short moment of boost. The last operating mode f) is the external charging which regenerates the battery state of charge [35].

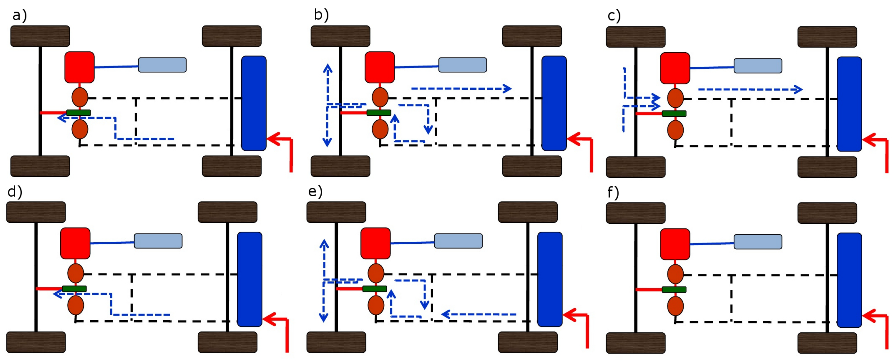


Figure 4.12: Operating modes and energy/power flows of a plug-in hybrid vehicle structure: a) Starting/Moving from standstill - Electric motor supplies power to wheels . b) Cruising or highway driving - ICE supplies power to wheels as well as charges the battery. c) Braking - Kinetic energy of braking is recuperated and directed to charge the battery. d) Low speed or city driving - Electric motor supplies power to wheels. e) Overtaking and accelerating - In addition to engine power, power is drawn from the battery through the electric motor. f) External charging - The battery can be externally charged, e.g., overnight. [35]

KIA Optima Mild Hybrid

This section describes the test-vehicle (Figure 4.13) which was used for all data input and measurements and shows its overall system architecture.
The car was originally delivered as a commercially available 2.0L turbo-diesel. It was turned into a 48V mild-hybrid vehicle which is now used for internal testing as well as customer presentations.

Figure 4.13: The KIA Optima which was used as the test-vehicle for all measurements and data input.

Figure 4.14 displays the overall system architecture of this new 48V hybrid system combined with an electric supercharger (eSC). The e-booster is placed in upstream position to the enlarged turbocharger (TC). The conventional alternator is replaced by the 48V BSG which is integrated into the belt-drive of the engine. The BSG is connected to the 48V battery by an inverter. Via a DC/DC converter the 12V battery is charged out of the 48V system.

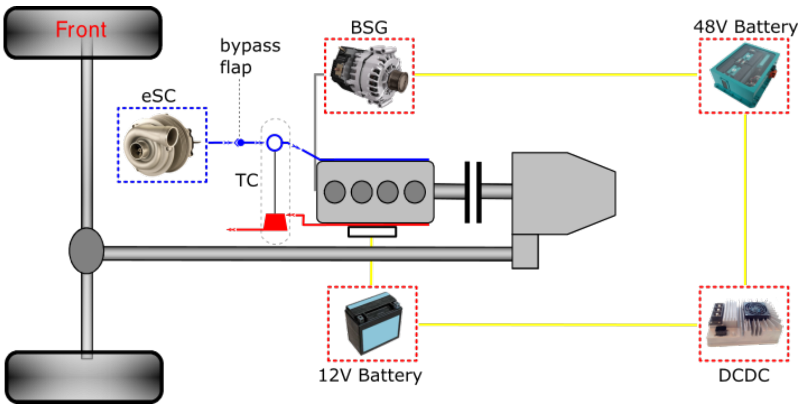


Figure 4.14: Internal hybrid component structure in the power-train of the KIA Optima Mild Hybrid which was used as the test-vehicle.

4.4 Android

Android is an operating system (OS) as well as a software platform for smartphones and tablet-computer. It is based on a Linux-kernel which was modified in every iteration. The OS is open source and anyone can write code and applications for his own device. Most applications are written in Java and run within individual instances of the Dalvik virtual machine (DVM). The key goals of the Android architecture are performance and efficiency, both in application execution and in the implementation of reuse in application design [2,42]. The Android operating system is a stack of software components which con be roughly divided into five sections and four main layers.

Kernel The linux kernel is positioned at the bottom of the software stack, providing a level of abstraction between the device hardware and the upper software stack. The kernel supports low-level core system services such as memory, process and power management as well as providing a network stack and drivers for audio, display and other peripherals [2,42].

Android runtime Android allows multitasking execution so multiple processes can run at the same time. Each process within the Android operating system runs in its own dalvik virtual machine. Running applications in virtual machines provides a number of advantages. Firstly, applications are essentially sandboxed, in that they cannot interfere with the operating system or other applications, nor can they directly access the device hardware. Secondly, this enforced level of abstraction makes it platform neutral. DVM is like Java virtual machine (JVM) but it is optimized for mobile devices. It consumes less memory and provides fast performance [2,42].

Core libraries The Java Interoperability Libraries are an implementation of a subset of the Standard Java core libraries that have been adapted and transformed for applications running within a Dalvik VM. They also include Java-based libraries that are specific to Android development including the application framework libraries in addition to user interface building, graphics drawing and database access [2, 42].

Application framework On the top of Native libraries and Android runtime, there is the Android framework. Android framework includes Android API's such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers. It provides a lot of classes and interfaces for Android application development [2, 42].

Applications Applications comprise both the native applications provided with the particular Android implementation, e.g., email applications, and the third party applications installed by the user after purchasing the device [2, 42].

Chapter 5

Design & Implementation

This chapter illustrates the general concept and design of the application and elucidates the objectives which were formulated in Chapter 2. Figure 5.1 depicts the most important classes and interfaces which are responsible for the applications functionality as an *Unified Modeling Language (UML) diagram*. In order to transmit data over a Bluetooth connection the application has to be paired to the remote device. The class *Bluetooth-CommunicationService* realizes the implementation of both the pairing as well as the read and write operations of an ongoing connection. The inner class, which is responsible for a pairing attempt, extends the Android *Thread* class. When an application is launched, the system creates a thread which interacts with all the user interface components, therefore often called UI thread. Time-consuming computations or processes can block the UI thread which results in a freezing screen. In order to prevent this, a Bluetooth connection attempt should be called from a separate thread. The mentioned inner class, which provides this feature, is called *ConnectThread*. It gets all currently paired devices and tries to connect to the one the user chooses. Once the pairing is successfully established the *ConnectThread* class starts another thread which handles all incoming and outgoing data transmissions. The implementation is realized in the class *ConnectedThread* which is only started once a pairing is successfully established. It gets the *Input-* and *OutputStream* of the Bluetooth protocol over a secured socket and reads incoming data bitwise. The class *BmCmd*, which is not shown in Figure 5.1, originates from the RM CANlink Bluetooth device API and checks if the read buffer is a complete message each time a new bit is incoming. Once the EOF field is being recognized, the full message is split into its components (specified in Section 5.2) and a new *BmcCanMessage* is created. This class contains all necessary information about one CAN message transmission, e.g., the identifier, the length of the data field, the raw data array as well as which type of frame was transmitted. After creating a new *BmcCanMessage* object, the *BmCmd* instance invokes the *CanMessageReceived()* method from the *ByteCommandCallbacks* interface. The concept of callbacks offers the possibility to use an event-driven model. In Java this is realized by defining a simple

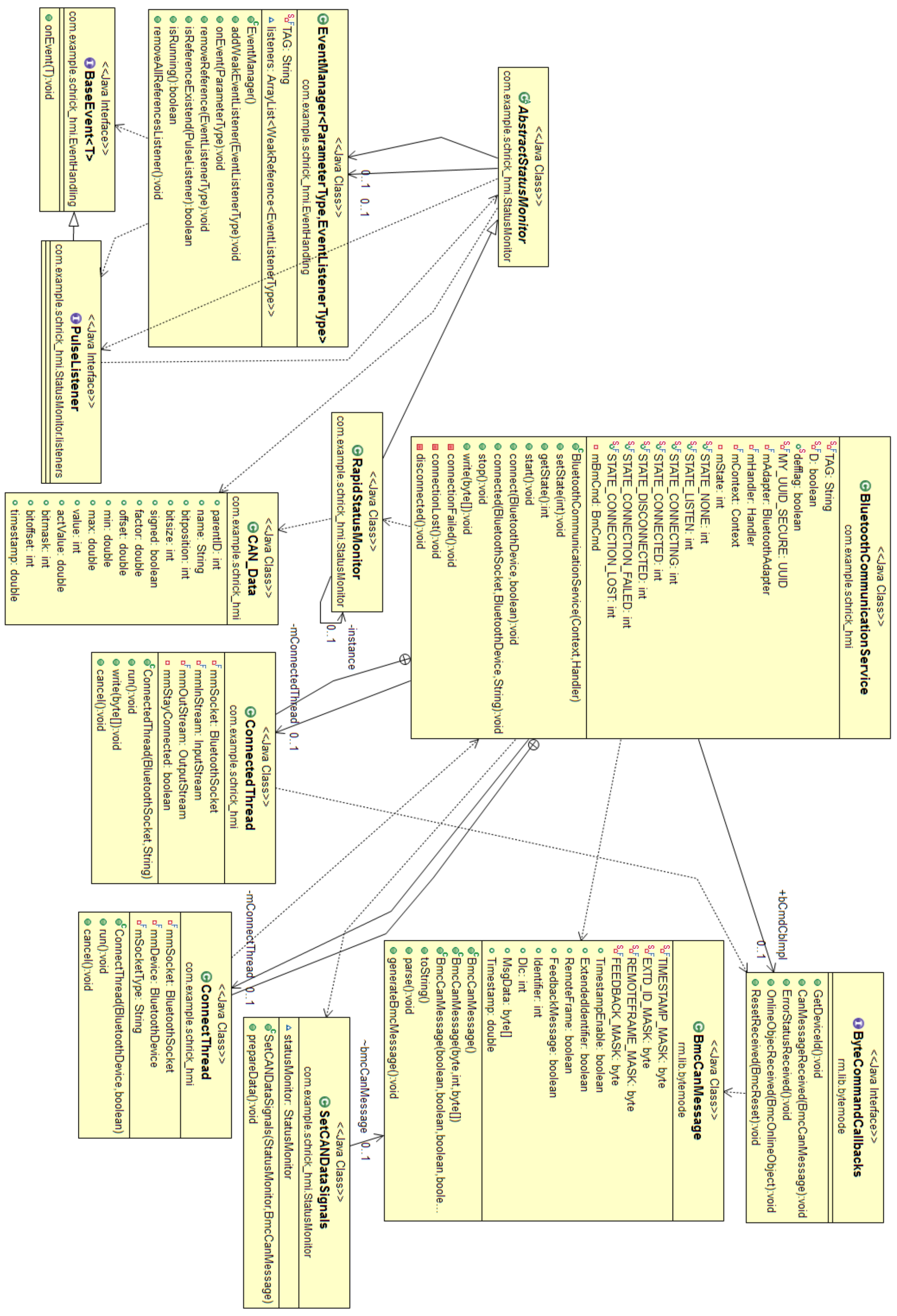
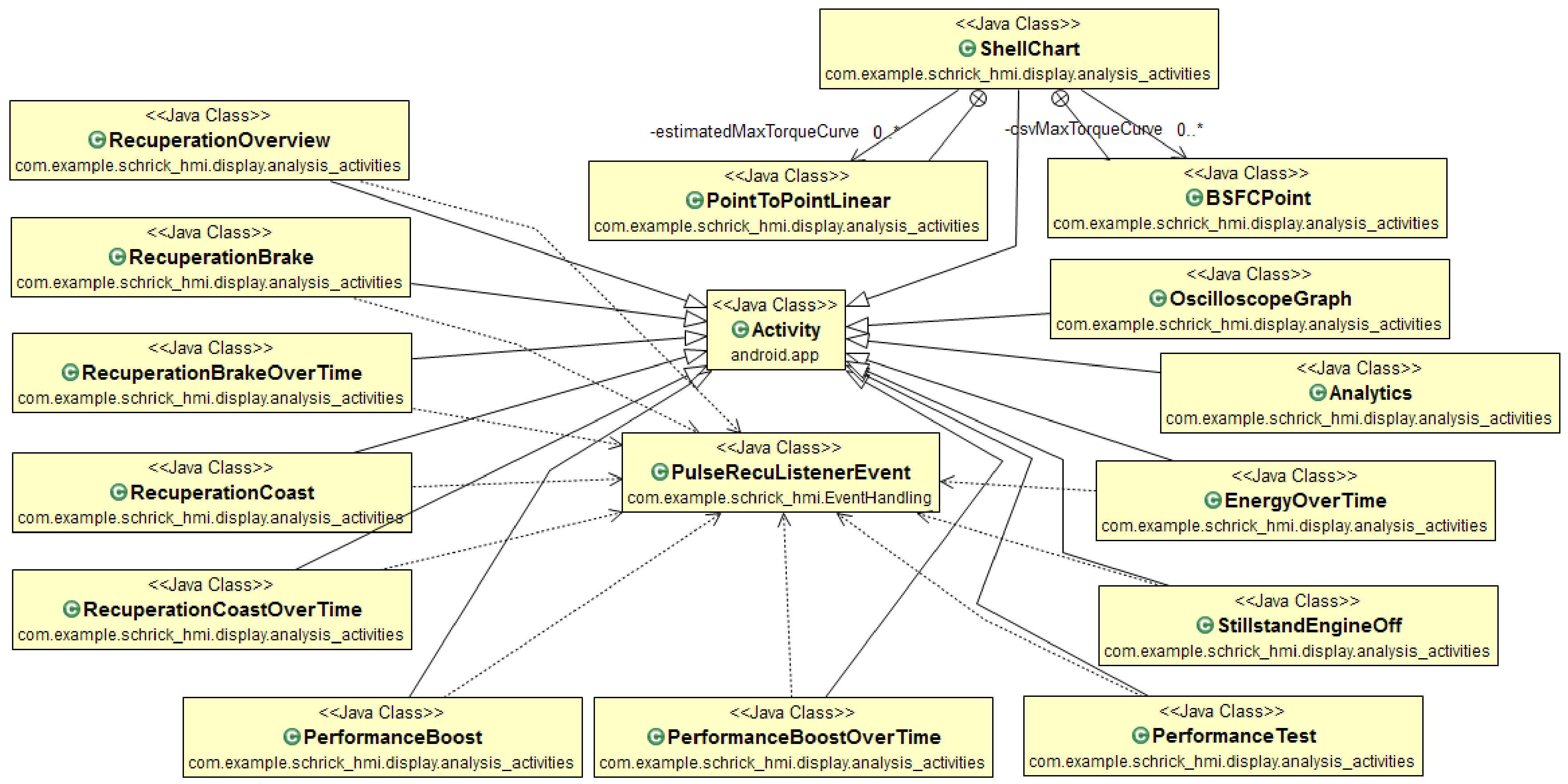


Figure 5.1: UML representation of the used communication, storage and event handling structure inside the application.

interface that declares the methods which should be invoked at some point. Every class which implements the interface is now able to get the callbacks once one of the specified methods is invoked.The class *BluetoothCommunicationService* implements the interface *ByteCommandCallbacks* and gets a *BmcCanMessage* object each time the *CanMessageReceived()* method is invoked. The containing data is then transformed into specific signal data which is stored into the corresponding *CAN_Data* object.
The storing and the graphical updating process is also shown in Figure 5.1. The abstract class *AbstractStatusMonitor* contains a *Map<K,V>* object where *K* is the type of keys maintained by this map and *V* is the type of mapped values or objects. In our case the key would be of type *String* and the mapped object is from type *CAN_Data*. Therefore, the signal and message data which are needed in order to do the computation have to be specified and *CAN_Data* objects are to be created with the according unique key. *AbstractStatusMonitor* uses the concept of *weak references* which means that the referenced object is not protected from the collection of a garbage collector. This is not the case when using *strong references*. An object, referenced only by weak references, is considered weakly reachable and can be treated as unreachable and may be collected at any time. Additionally, when an object has other registered objects - particularly in event handling situations - if a strong reference is kept, these objects must be explicitly unregistered, otherwise a memory leak occurs, whereas a weak reference removes the need to unregister the objects [34]. *AbstractStatusMonitor* contains one to many *EventManager* objects which are holding the listener objects in an array. Additional methods for removing the referenced objects were added in order to have the possibility to remove a listener manually. In the current scenario there is only a *PulseListener*, which is updating in a certain interval. The abstract class *AbstractStatusMonitor* implements a *Handler* object which executes a *Runnable* object which invokes the listeners in the specified update interval. The *Runnable* invokes the *onEvent()* method for each *PulseListener* which is in the *EventManagers* listeners list. An Android Activity or Fragment, which implements the *PulseListener* interface, can therefore execute code periodically. Since the *AbstractStatusMonitor* is given as a reference for a *PulseListeners onEvent()* method, it has access to the *Map* which contains the *CAN_Data* objects. These objects are updated once a new Bluetooth CAN message is being transmitted. Given the unique key, the corresponding *CAN_Data* object is accessible and the stored values can be retrieved. This can be done for multiple keys at once and the resulting values are then used to compute the vehicle state and to create a graphical representation. With the subclass *RapidStatusMonitor* it is possible to manually set an update interval on the creation process. This update interval is the frequency in which the *onEvent()* method for all listeners is being invoked.
Figure 5.2 shows an overview of the used layout design and the event handling process in order to visualize the values on the screen. For each graphical representation layout an Android Activity was created. Each Activity holds its own UI components and computes



<<Java Class>>
ShellChart
com.example.schrick_hmi.display.analysis_activities
-estimatedMaxTorqueCurve 0..*
-csvMaxTorqueCurve 0..*
<<Java Class>>
RecuperationOverview
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
PointToPointLinear
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
BSFCPoint
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
RecuperationBrake
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
Activity
android.app
<<Java Class>>
OscilloscopeGraph
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
RecuperationBrakeOverTime
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
Analytics
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
PulseRecuListenerEvent
com.example.schrick_hmi.EventHandling
<<Java Class>>
RecuperationCoast
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
EnergyOverTime
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
RecuperationCoastOverTime
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
StillstandEngineOff
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
PerformanceBoost
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
PerformanceBoostOverTime
com.example.schrick_hmi.display.analysis_activities
<<Java Class>>
PerformanceTest
com.example.schrick_hmi.display.analysis_activities

Figure 5.2: Representation of the used layout structure components as well as the corresponding event handling connections.

different energy behavior aspects. Therefore, the values which are stored in the *CAN_ Data* objects need to be retrieved in the *Map* object of the subclass from *AbstractStatusMonitor*. Because it would be bad practice to implement a *PulseListener* which executes nearly the same code in each Activity, the *PulseListener* was implemented in the main Activity of the application. When a measurement is started, the values are stored in a background thread, but only presented in a graphical way once one of the pictured Activities is started. However, because the implementation of the *PulseListener* is not implemented in each Activity anymore, the necessary values for further computations are only present in the main Activity. Therefore, an event bus system is used, where the class event *PulseRecuListenerEvent* is defined, which is invoked each time the *PulseListener* is updated in the main Activity. Each analysis Activity has to subscribe to the event bus in order to get the posted message events. This is a fast and reliable way to coordinate data communication between Activities.
In the following sections some of the steps are elucidated in a more detailed way and code snippets further clarify the implementation process.

5.1 State Definition

In order to define the necessary states which are used for the analysis, we first have to understand the basic concepts of different hybrid configurations, described in Section 4.3. Since there is no standard hybrid configuration, there are vehicles with different usable hybrid operating phases. The goal is to define states which are suitable for most hybrid, electric and conventional vehicles. Therefore a minimum set of signals is used for the decision process. It may be possible that the chosen test vehicle does not support a specific defined state. Furthermore, the designed solution prevents getting into a state which the car does not support. However, with this approach of generalizing as much as possible we can only model the most basic hybrid features as a state, which results in the following states:

- Brake
- Coastdown
- Boost
- Load Point Moving (LPM)
- Recuperation
- Zero State

The signals which are used to match the vehicle state need to be easily accessible and should be available on each of the vehicles CAN buses. If the signal is not on the CAN bus, the integration shall be as easy as possible. Because of these circumstances the chosen signals for the matching algorithm are:

- Accelerator Pedal Position [%]
- Brake Pedal Position [%]
- Vehicle Speed [kph]
- Battery Current [A]
- Battery Voltage [V]

The signal *Accelerator Pedal Position* indicates the current state of the accelerator pedal. If the driver is not pushing the accelerator at all, the signal is equal to zero. If it is pressed to the maximum, the value is one-hundred. The signal for *Brake Pedal Position* works in the same way. *Vehicle Speed* defines how fast the vehicle is currently driving in kilometers per hour, while the signals *Battery Current* and *Battery Voltage* indicate how much power the battery gains or loses. With the help of these five signals, we are able to describe the mentioned states.
The state *Brake* indicates that the vehicle is situated in a process where the braking pedal is being actively used. Because standstill braking is not desired, a threshold value for the velocity is necessary. For this state it is not important if it is regenerative braking or not. It only shows if the car is in a braking process. Therefore, we define a braking event like 5.1.1.

5.1.1 Definition (Brake). *The **Brake-State** can be defined with:*
$BrakePedalPosition > 0.5\ \%$
$AcceleratorPedalPosition \leq 0.5\ \%$
$VehicleSpeed > 2\ kph$

Similar to the state *Brake* is the state *Coastdown* because it also describes a deceleration process, but with no pedal being actively used. Basically, the engine brake is actively used during this process which can be pictured as a vehicle rolling until it stands still or accelerates again. Like in the previous state we do not look at the power or energy output, because they are different for each hybrid configuration. The vehicle can still use *Coastdown* even if it is not recuperating energy in the process. This just indicates that it is either not possible to harvest energy because of the vehicles configurations or the calibration is not good enough and it is losing in efficiency. Therefore, we define a coastdown event like 5.1.2.

5.1.2 Definition (Coastdown). *The **Coastdown-State** can be defined with:*
$BrakePedalPosition \leq 0.5$ %
$AcceleratorPedalPosition < 0.5$ %
$VehicleSpeed > 2$ kph

The state *Boost* indicates that the electric motor is supplying power from the battery to support the ICE. Hereby, the electric motor needs to drain at least a certain amount of power from the battery while the driver is pushing the accelerator pedal. This state has a small drawback, because the threshold value for the necessary drained battery power might not be high enough for all system configurations. If a vehicle has a lot of enabled electronics which need power from the battery, the application might compute the state *Boost* because the power consumption could be over 500 W. Therefore, the threshold value needs to be calibrated accordingly. The best method to achieve a proper threshold value would be the analysis from the battery power consumption while driving in a conventional car with all electronic systems enabled. Unfortunately, we only had one test vehicle to calibrate the power threshold value.

5.1.3 Definition (Boost). *The **Boost-State** can be defined with:*
$BrakePedalPosition \leq 0.5$ %
$AcceleratorPedalPosition \geq 0.5$ %
$BatteryPower < -500$ W

Load Point Moving (LPM) is defined as an energy gain while driving. The engine operates at an efficient load point which generates more energy due to a higher combustion conversion efficiency and is therefore charging the battery. The accelerator pedal needs to be pressed, otherwise state *Coastdown* would be computed. Therefore, we define *LPM* like 5.1.4.

5.1.4 Definition (LoadPointMoving). *The **LPM-State** can be defined with:*
$AcceleratorPedalPosition > 0.5$ %
$BatteryCurrent > 0$ A

Recuperation is the recovery of energy to charge the battery system. In this case, the state *Recuperation* is computed when neither the states *Brake* nor the state *Coastdown* is applicable. However, state *LPM* is not regarded as a subset of *Recuperation*, while it technically is a recuperation process, because we are not necessarily in a decelerating process. State *Recuperation* is detected when the car is generating power while standing, e.g at a traffic light without a start-stop system activated. In this case the generator charges the battery. Therefore, state *Recuperation* is defined like 5.1.5.

5.1.5 Definition (Recuperation). *The **Recuperation-State** can be defined with:*
$AcceleratorPedalPosition \leq 0.5$ %
$BatteryCurrent > 0$ A

State *Zero* occurs when none of the above states is applicable in the current situation. For example, when the driver accelerates slowly and the hybrid strategy does not decide to boost or when the driver cruises but the system cannot use LPM to generate additional energy to charge the battery. In these scenarios the state *Zero* is applied.

5.1.6 Definition (Zero). *State Zero occurs when none of the previous states applies.*

With the above defined states and the chosen signals we are able to describe and analyze some basic hybrid features. It is possible to calculate the energy and power values for each point in time and relate them to the corresponding state. Additionally, with the model it is also possible to investigate certain scenarios, like how much energy was recuperated during brake event X, in a detailed way.

5.2 Network device & connection protocol

The first design decision was made regarding the communication structure. Figure 5.3 illustrates the two possible configurations which were considered. Configuration a) uses a communication device with GSM and GPRS module so that we are able to send the data wherever the HMI is positioned as well as transmitting data to a server. Configuration b) communicates with a network device as well, but does not want to send CAN data to a server. Therefore, the communication device for configuration b) does not have to be GSM-capable.

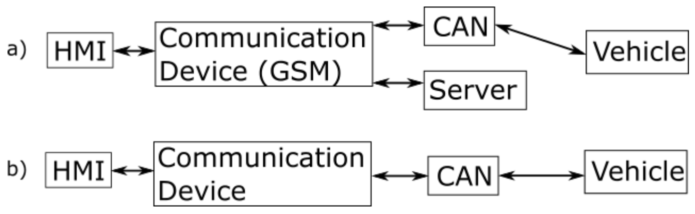
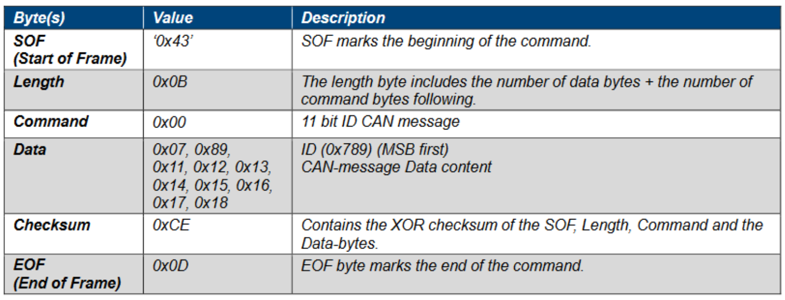


Figure 5.3: Possible communication structure: a) HMI communicates with a GSM-capable device which sends and receives data to the CAN bus and a server. b) HMI communicates with a networking device which sends and receives data from the CAN bus.

If we analyze the two configurations, there are drawbacks in both of them. While configuration a) is connected with a GSM-module and therefore reachable over the world regardless the position, the data transmitting latency will rise. In addition, a server must be provided which has ongoing costs. Additionally, if there are multiple projects with different customers, the data management and data security need to fulfill a certain standard which may be hard to realize. However, a server can bring the data to other platforms and the post-processing possibilities of stored data are huge. Configuration b) can either use a short communication network device which only communicates over the Bluetooth stack or the same GSM-module which would have been used for configuration a). When using a

Bluetooth module for the data communication, the latency will be better but the distance to the HMI has to be shorter as well. The positive effect of being able to get data from all over the world will be gone. Aside from that, the data storage needs to be done on the device instead of a server which adds substantial weight on a smart implementation.
While researching GSM- and Bluetooth-modules, like the Telit EVK-ATOP, Locator-G3, Axotec Telematic Box or the RM CANlink Bluetooth/Mobile, the first choice was the *RM CANlink Bluetooth* module manufactured by *Proemion* (Fulda, Germany). The decision was based on multiple factors. The most decisive criteria were: initial investments for the hardware as well as ongoing maintenance costs, vehicle integration expense, further implementation expenses to integrate the features, data security aspects, transmit/receive data message latency. The idea of backing up the customer data on a server for further post-processing was neglected, because the initial hardware investments as well as ongoing maintenance costs are too high for a pilot project. Another argument against a GSM-module in general was the high latency during a message transaction. A stable connection cannot be guaranteed in all circumstances. However, a continuous data stream is necessary for a representation and computation in real-time. Bluetooth has a lower range but provides a stable connection with acceptable latency. The *RM CANlink Bluetooth* module was chosen because of prior experience with the component. It also has an API for the CAN protocol and the Bluetooth communication which can be used to a certain degree.
Figure 5.4 illustrates the *CANlink Bluetooth* message frame specification. The structure is similar to the CAN 2.0A message frame which was introduced in Section 4.1.2.



Byte(s)	Value	Description
SOF (Start of Frame)	'0x43'	SOF marks the beginning of the command.
Length	0x0B	The length byte includes the number of data bytes + the number of command bytes following.
Command	0x00	11 bit ID CAN message
Data	0x07, 0x89, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18	ID (0x789) (MSB first) CAN-message Data content
Checksum	0xCE	Contains the XOR checksum of the SOF, Length, Command and the Data-bytes.
EOF (End of Frame)	0x0D	EOF byte marks the end of the command.

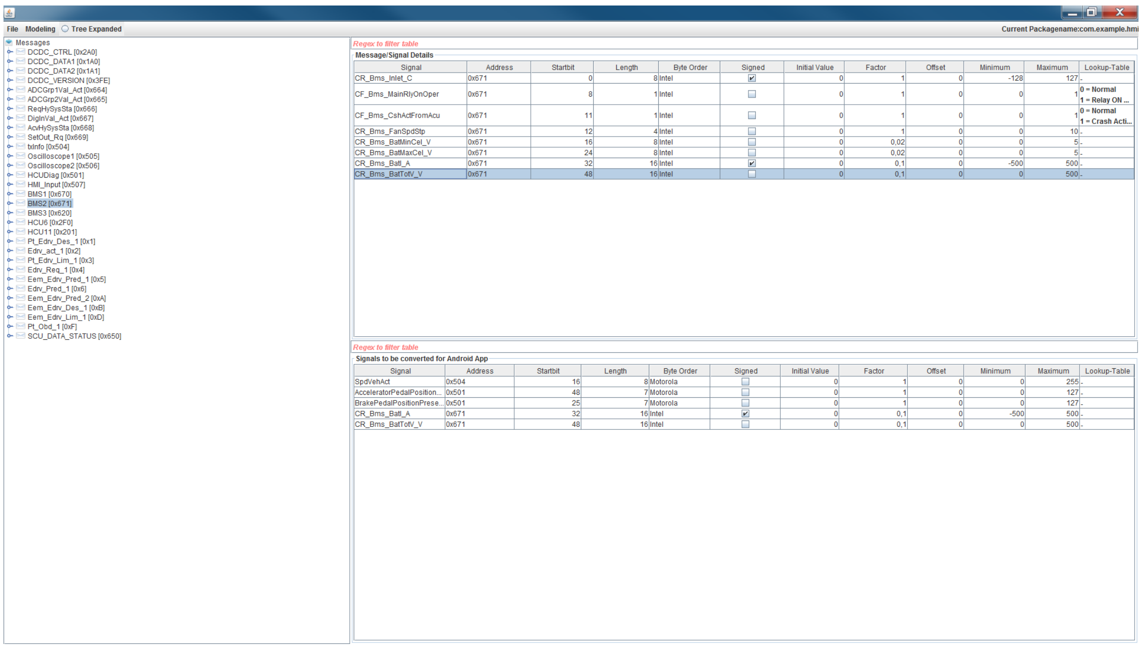
Figure 5.4: Transmission and reception message specification which is used by the *RM CANlink Bluetooth* device [36].

The *Start of Frame* (SOF) marks the beginning of a new command. The value `C (0x43)` is the default setting, but for some device this value is changeable. *Length* decodes the number of following data bytes as well as command bytes and can range from `0x00` to `0xFF`. The *Command* byte denotes direct commands which can range from `0x00` to `0xFE`. For example, value `0x00` stands for a received 11 bit Identifier CAN message. The *Data* field contains the 11 or 29 bit CAN Identifier followed by additional data bytes. An XOR checksum over SOF, Length, Command and the data bytes content results in the *Check-*

sum field. The *End of Frame* (EOF) marks the end of the command. For a detailed specification description and command set specification see [36].

5.3 Message & signal selection

Because there is no standardized version of the vehicle CAN bus configuration and signal mapping, the signals needed for computation or displaying might be in different messages with different endiannesses. Since it is necessary to specify the message ID as well as the factor, min/max restrictions, offset, startbit and length of each signal, a Java-tool with code-generation feature was developed. The specification is needed, because otherwise we cannot match and read the raw data content from a Bluetooth message stream to a valid message inside the application. With the developed tool, which can be seen in Figure 5.5, *dbc* and *Excel* CAN bus specifications can be read.



Messages
- DCDC_CTRL [0x2A0]
- DCDC_DATA1 [0x1A0]
- DCDC_DATA2 [0x1A1]
- DCDC_VERSION [0x3FE]
- ADCGrp1Val_Act [0x664]
- ADCGrp2Val_Act [0x665]
- ReqHySysSta [0x666]
- DigInVal_Act [0x667]
- AcvHySysSta [0x668]
- SetOut_Rq [0x669]
- txInfo [0x504]
- Oscilloscope1 [0x505]
- Oscilloscope2 [0x506]
- HCUDiag [0x501]
- HMI_Input [0x507]
- BMS1 [0x670]
- BMS2 [0x671]
- BMS3 [0x620]
- HCU6 [0x2F0]
- HCU11 [0x201]
- Pt_Edrv_Des_1 [0x1]
- Edrv_act_1 [0x2]
- Pt_Edrv_Lim_1 [0x3]
- Edrv_Req_1 [0x4]
- Eem_Edrv_Pred_1 [0x5]
- Edrv_Pred_1 [0x6]
- Eem_Edrv_Pred_2 [0xA]
- Eem_Edrv_Des_1 [0xB]
- Eem_Edrv_Lim_1 [0xD]
- Pt_Obd_1 [0xF]
- SCU_DATA_STATUS [0x650]

Message/Signal Details

Signal	Address	Startbit	Length	Byte Order	Signed	Initial Value	Factor	Offset	Minimum	Maximum	Lookup-Table
CR_Bms_Inlet_C	0x671	0	8	Intel	✓	0	1	0	-128	127	-
CF_Bms_MainRlyOnOper	0x671	8	1	Intel	☐	0	1	0	0	1	0 = Normal 1 = Relay ON ...
CF_Bms_CshActFromAcu	0x671	11	1	Intel	☐	0	1	0	0	1	0 = Normal 1 = Crash Acti...
CR_Bms_FanSpdStp	0x671	12	4	Intel	☐	0	1	0	0	10	-
CR_Bms_BatMinCel_V	0x671	16	8	Intel	☐	0	0,02	0	0	5	-
CR_Bms_BatMaxCel_V	0x671	24	8	Intel	☐	0	0,02	0	0	5	-
CR_Bms_BatI_A	0x671	32	16	Intel	✓	0	0,1	0	-500	500	-
CR_Bms_BatTotV_V	0x671	48	16	Intel	☐	0	0,1	0	0	500	-

Signals to be converted for Android App

Signal	Address	Startbit	Length	Byte Order	Signed	Initial Value	Factor	Offset	Minimum	Maximum	Lookup-Table
SpdVehAct	0x504	16	8	Motorola	☐	0	1	0	0	255	-
AcceleratorPedalPosition...	0x501	48	7	Motorola	☐	0	1	0	0	127	-
BrakePedalPositionPrese...	0x501	25	7	Motorola	☐	0	1	0	0	127	-
CR_Bms_BatI_A	0x671	32	16	Intel	✓	0	0,1	0	-500	500	-
CR_Bms_BatTotV_V	0x671	48	16	Intel	☐	0	0,1	0	0	500	-

Figure 5.5: The Java-tool enables the user to read Excel and dbc CAN bus specifications. The resulting CAN messages and signals are displayed in the left window. After selecting a specific message, it will collapse and show the containing signals on the top right table. The bottom table is responsible for the export feature. All containing signals are exported into Java files.

A vehicle CAN specification contains information about all messages and signals which are transmitted over the bus system, their ID, startbit, length, factor and so on. The application reads the information and presents all CAN messages on the left side of the window. If the user selects a message, it will collapse, showing the containing signals. The signals will also be depicted in the upper right table. Since we do not want to receive all messages which are specified on the vehicle CAN bus, the signals needed for the applica-

tions illustration of the layout views and analysis feature have to be selected beforehand. This prevents an overhead of data signals which are not providing any desired content for the application. In order to select specific signals, the tool allows the user to use drag and drop mechanisms to put signals from the top right table to the bottom right table. The bottom table stores the signals which are necessary for the application working process. In our example five signals from different messages were dragged into the table. The desired output are Java-class files which can be instantly imported into the source files of the Android application and provide a different set of signal specifications. Therefore, we are able to swap vehicle specifications with little effort and only one building process instead of selecting the values by hand and implementing them in the right places.
The export to Java-classes produces two classes with different structure. The first Java-class, called *StatusMonitor*, defines each signal and puts it into a *HashMap* object under a given name, which also represents the key, while creating a new *CAN_Data* object. CAN_Data objects contain simple signal information like factor, minimum value, maximum value, offset as well as functions which calculate the actual values with regard to the specifications. Listing 5.1 shows an excerpt of the output if the signal table from Figure 5.5 is exported. It specifies every signal and puts it into an underlying *HashMap* with the signal-name as the key. The CAN_Data object for each signal contains data like: factor, minimum and maximum value, offset, signed or unsigned as well as the message identification number and the name.

```java
private StatusMonitor(){
  super(500);
  //Definition of each signal
  canState.put("SpdVehAct", new CAN_Data(0x504,"SpdVehAct",16,8,false
  ,1.0,0.0,0.0,255.0,0));
  canState.put("AcceleratorPedalPositionPresent",
  new CAN_Data(0x501,"AcceleratorPedalPositionPresent",48,7,false
  ,1.0,0.0,0.0,127.0,0));
  canState.put("BrakePedalPositionPresent",
  new CAN_Data(0x501,"BrakePedalPositionPresent",25,7,false
  ,1.0,0.0,0.0,127.0,0));
  canState.put("CR_Bms_BatI_A", new CAN_Data(0x671,"CR_Bms_BatI_A",32,16,
 true,0.1,0.0,-500.0,500.0,0));
  canState.put("CR_Bms_BatTotV_V", new CAN_Data(0x671,"CR_Bms_BatTotV_V"
  ,48,16,false,0.1,0.0,0.0,500.0,0));
  super.Initialize();
}
```

Listing 5.1: Specifying signal values into StatusMonitor. Each signal will create a new CAN_Data object with underlying data information. Each object is referenced with its signal-name as the key.

The second class which is generated during the export process is called *SetCANDataSignals*. An excerpt is shown in Listing 5.2. Every time the Bluetooth stream sends a new message, the data is set in the object it belongs to. The CAN_Data object is referenced with its name and *bmcCanMessage.MsgData* contains the raw byte array data information from the Bluetooth stream. The underlying functions transform the raw byte data into the actual value and place it into the CAN_Data object.

```
  public void prepareData(){
    if(bmcCanMessage.Identifier == 0x501){
      statusMonitor.setValuesWithKey("AcceleratorPedalPositionPresent",
 bmcCanMessage.MsgData);
      statusMonitor.setValuesWithKey("BrakePedalPositionPresent",
 bmcCanMessage.MsgData);
    }else if(bmcCanMessage.Identifier == 0x504){
      statusMonitor.setValuesWithKey("SpdVehAct", bmcCanMessage.MsgData);
    }else if(bmcCanMessage.Identifier == 0x671){
      statusMonitor.setSignedIntelValueWithKey("CR_Bms_BatI_A",
 bmcCanMessage.MsgData);
      statusMonitor.setIntelValueWithKey("CR_Bms_BatTotV_V", bmcCanMessage
 .MsgData);
    }
  }
```

Listing 5.2: Sets the values into the underlying CAN_Data signal object with regard to the endianness and the signed status.

There is a function for each endianness and signed status, because the value has to be transformed differently each time. A more detailed explanation of the raw data transformation is given in Section 5.5.

5.4 Bluetooth connection

An important first step is to determine whether the mobile Android device supports Bluetooth. If that is the case, the feature has to be turned on. In order to use a Bluetooth connection in the application, the devices have to be paired first. The pairing needs to be done in the Android system settings and it is good practice to unpair all other devices that might be in the list. Once the devices are paired, the application can set up a Bluetooth connection to receive and send data.
The first step to set up a Bluetooth connection is the creation of a *BluetoothAdapter* object using the function *getDefaultAdapter()*. If the function call returns null, the Android device does not support Bluetooth. The next step is illustrated in Listing 5.3, where it is examined whether Bluetooth is enabled or not. If its not enabled an Intent is transmitted to the Android system which enables the feature.

```
if (!mBluetoothAdapter.isEnabled()) {
  Intent enableBluetoothIntent = new Intent(BluetoothAdapter.
    ACTION_REQUEST_ENABLE);
  startActivityForResult(enableBluetoothIntent, 1);
}
```

Listing 5.3: Turns on Bluetooth if its not enabled yet.

In order to use a Bluetooth connection the devices have to be paired. Therefore, it is necessary to get all possible devices to which the application is able to connect. Listing 5.4 shows the implementation of this request. With the function *getBondedDevices()* we are able to gather all Bluetooth devices which were paired to the device at one point.

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
  for (BluetoothDevice device : pairedDevices) {
    ...
  }
}
```

Listing 5.4: Get all paired devices.

With the set of usable devices a dialog window is created from which the user can pick a device to connect to. At this point the Bluetooth device is stored into the *BluetoothDevice* object. The next objective would be to establish a connection between both devices. Commonly, this takes place in another thread to avoid blocking the main thread with possible heavy computing. Listing 5.5 shows the code as an inner class of the class which specified the previous connection settings as well, in our case the class *BluetoothCommunicationService*. This thread requires a BluetoothDevice as a parameter and uses it to create a BluetoothSocket. This socket is what Bluetooth uses to transfer data between devices. The used UUID provides the socket with the information that data will be transferred serially, which means one byte at a time.

```
private class ConnectThread extends Thread {

private final BluetoothSocket mSocket;
private final BluetoothDevice mDevice;
private final String mSocketType;

public ConnectThread(BluetoothDevice device, boolean secure) {
  mDevice = device;
  BluetoothSocket tmp = null;
  mSocketType = secure ? "Secure" : "Insecure";
  try {
    if (secure) {
      tmp = device.createRfcommSocketToServiceRecord(UUID_SECURE);
    } else {
      tmp = device.createInsecureRfcommSocketToServiceRecord(UUID_SECURE);
    }
  } catch (IOException e) {
    ...
  }
  mSocket = tmp;
}
...
@Override
public void run() {
  setName("ConnectThread" + mSocketType);
  // Always cancel discovery because it will slow down a connection
  mAdapter.cancelDiscovery();

  try {
    mSocket.connect();
    ...
  }
  ...
  // Start the connected thread
  connected(mSocket, mDevice, mSocketType);
}
...
```

Listing 5.5: This thread runs while attempting to make an outgoing connection with a device. It runs straight through; the connection either succeeds or fails.

In order to establish a connection the thread has to be started with the previous created *BluetoothDevice* object as well as the information if a secured connection is desired. Listing 5.6 shows the implementation of the thread initialization. Once the *run()* method of the *ConnectThread* is finished it calls the function *connected()* which will start another thread. The so called *ConnectedThread* provides the tools to read the incoming data as well as write data bytes on its own.

```
// Start the thread to connect with the given device
mConnectThread = new ConnectThread(device, secure);
mConnectThread.start();
setState(STATE_CONNECTING);
```

Listing 5.6: Start the connect thread to establish a connection to the given device.

An excerpt of the code realization from the *ConnectedThread* class can be seen in Listing 5.7. Like connecting, transferring data is time-consuming and can block the UI thread, therefore it should also run in a separate thread. The thread requires a *BluetoothSocket* as a parameter and uses it to create an *InputStream* and an *OutputStream*. The *InputStream* is used for reading data coming from the remote device while the *OutputStream* is used to transmit data to the remote device.

```java
private class ConnectedThread extends Thread {
  private final BluetoothSocket mSocket;
  private final InputStream mInStream;
  private final OutputStream mOutStream;
  private boolean mStayConnected;

  public ConnectedThread(BluetoothSocket socket, String socketType) {
    mSocket = socket;
    InputStream tmpIn = null;
    OutputStream tmpOut = null;

    // Get the BluetoothSocket input and output streams
    try {
      tmpIn = socket.getInputStream();
      tmpOut = socket.getOutputStream();
    } catch (IOException e) {
      ...
    }
    mBmCmd = (new BmCmd(bCmdCbImpl, null));
    mInStream = tmpIn;
    mOutStream = tmpOut;
    mStayConnected = true;
  }

   @Override
   public void run() {
     byte[] buffer = new byte[1024];
     int bytes;

     // Keep listening to the InputStream while connected
     while (mStayConnected) {
       try {
         // Read from the InputStream
         bytes = mInStream.read(buffer);
        mBmCmd.s8RxData(buffer, bytes);
         ...
```

Listing 5.7: This thread runs during a connection with a paired remote device. It handles all incoming and outgoing transmissions.

Reading the *InputStream* is not an easy task. Based on the *RM CANlink Bluetooth* message specification (see Figure 5.4) the *RM* API will prepare the data and split them into the specified protocol fields. Once a whole message is being received, a *CanMessageReceived* method from the *ByteCommandCallbacks* object is invoked with the containing raw values for each field. The *BluetoothCommunicationService* class implements the interface and

receives the data. If there is no error in the received data theay are transformed as described in Section 5.5.
In order to start the *ConnectedThread* the function *connected()*, which is illustrated in Listing 5.8, is called. At first, the thread that completed the connection as well as any thread which is currently running a connection attempt, are canceled. Afterwards the *ConnectedThread* is started and a message is send to the UI thread in order to display a message about the successfully established connection.

```java
public synchronized void connected(BluetoothSocket socket, BluetoothDevice
    device, final String socketType) {
  // Cancel the thread that completed the connection
  ...
  // Cancel any thread currently running a connection
  ...
  // Start the thread to manage the connection and perform transmissions
  mConnectedThread = new ConnectedThread(socket, socketType);
  mConnectedThread.start();
  // Send the name of the connected device back to the UI Activity
  Message msg = mHandler.obtainMessage(MainActivity.MESSAGE_DEVICE_NAME);
  Bundle bundle = new Bundle();
  bundle.putString(MainActivity.DEVICE_NAME, device.getName());
  msg.setData(bundle);
  mHandler.sendMessage(msg);
  setState(STATE_CONNECTED);
}
```

Listing 5.8: Starts the *ConnectedThread* which enables the application to read the incoming data as well as write data bytes on its own. Also sends information back to the UI thread in order to display an Android Toast which states that the connection was successfully established.

If the connection is lost because of any reasons the tread will be in a *STATE_ DISCONNECT* or *CONNECTION_ LOST* state which results in an immediate reconnection attempt. If this attempt is not successful the user has to manually try to reestablish the connection by connecting to the device again.

5.5 Transforming raw CAN data

After establishing a Bluetooth connection as well as receiving a *ByteCommandCallback* event after preparing the data, the data field can be transformed into usable values. The CAN message data field contains up to eight data bytes in which multiple signals can be assigned. Because the start bit and the length of each specified signal are known, the raw data can be transformed.
The first step is to distinguish the CAN messages based on their message ID. Listing 5.2 shows the implementation inside the *BluetoothCommunicationService* class. If the

incoming CAN message has one of those specified IDs the application calls one of the transformation functions in order to get an actual value for each signal which is specified as a CAN_Data object under the given message ID. One of the transformation method implementations can be seen in Listing A.3 in the appendix. The transformation function mainly uses two assisting functions to compute the desired signal values. Listing 5.9 is a recursive function which calculates the most significant bit position of a signal, given its length and start position. In order to transform the raw data, knowledge about the end of the signal is needed.

```java
private int getMSBbitFromSignal(int lsb, int length) {
  //Compute assigned bits per byte
  int rest = 8 - (lsb % 8);
  //Assigned bits are compared with signal length
  if (rest < length) {
    //New lsb value
    lsb += rest - 2 * 8;
    //New signal length
    length -= rest;
    //Recursive call with new values
    return getMSBbitFromSignal(lsb, length);
  } else {
    //Calculate msb
    return (lsb + length - 1);
  }
}
```

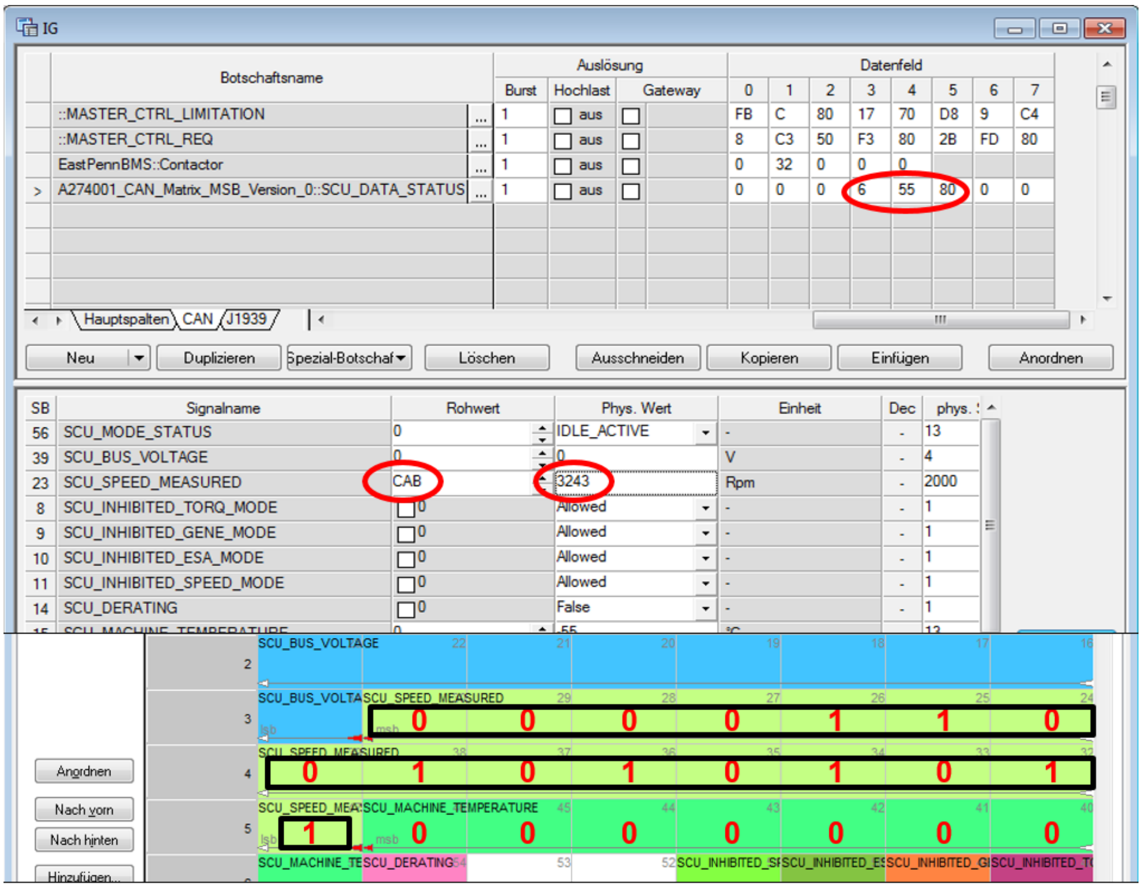
Listing 5.9: Computes the most significant bit (MSB) given the least significant bit (LSB) and the signal length.

The second function implementation is illustrated in Listing 5.10. It computes the value for a single byte if a bit mask and a bit offset is given.

```java
private static int getIntFromBits(byte Byte0, int pMask, int pBitOffset) {
  int result;
  result = ((Byte0 & pMask) >> pBitOffset-1);
  return result;
}
```

Listing 5.10: Computes the value for a single byte with a given mask and bit offset (range from 1 to 8).

The algorithm will be explained based on an example which is illustrated in Figure 5.6. In the bottom half of the picture, a data field layout with assigned signals is displayed. The signal *SCU_SPEED_MEASURED* is currently being transmitted over the Bluetooth connection. Once the CAN message is received and split into its components, the data field will be `(0 0 0 6 55 80 0 0)`. These are the raw values which are transmitted for each byte. The signal should be received has his least significant bit on position 47 and



	Botschaftsname		Auslösung			Datenfeld							
			Burst	Hochlast	Gateway	0	1	2	3	4	5	6	7
	::MASTER_CTRL_LIMITATION	...	1	☐ aus	☐	FB	C	80	17	70	D8	9	C4
	::MASTER_CTRL_REQ	...	1	☐ aus	☐	8	C3	50	F3	80	2B	FD	80
	EastPennBMS::Contactor	...	1	☐ aus	☐	0	32	0	0	0			
>	A274001_CAN_Matrix_MSB_Version_0::SCU_DATA_STATUS	...	1	☐ aus	☐	0	0	0	6	55	80	0	0

SB	Signalname	Rohwert	Phys. Wert	Einheit	Dec	phys.
56	SCU_MODE_STATUS	0	IDLE_ACTIVE	-	-	13
39	SCU_BUS_VOLTAGE	0	0	V	-	4
23	SCU_SPEED_MEASURED	CAB	3243	Rpm	-	2000
8	SCU_INHIBITED_TORQ_MODE	☐ 0	Allowed	-	-	1
9	SCU_INHIBITED_GENE_MODE	☐ 0	Allowed	-	-	1
10	SCU_INHIBITED_ESA_MODE	☐ 0	Allowed	-	-	1
11	SCU_INHIBITED_SPEED_MODE	☐ 0	Allowed	-	-	1
14	SCU_DERATING	☐ 0	False	-	-	1

Byte								
2	SCU_BUS_VOLTAGE							
3	SCU_BUS_VOLTA	0	0	0	0	1	1	0
4	0	1	0	1	0	1	0	1
5	1	0	0	0	0	0	0	0
6	SCU_MACHINE_TE	SCU_DERATING			SCU_INHIBITED_SF	SCU_INHIBITED_ES	SCU_INHIBITED_GI	SCU_INHIBITED_TC

Figure 5.6: Example of a CAN message with signal specification. The data field layout shows bytes 2 to 5. The light green signal is the signal currently being transmitted with a value of 3243 which raw data representation is illustrated in the bottom half of the picture.

the signal length is 16 bit. There are no information about the MSB and the bit offset in the implementation yet. With the help of the function *getMSBfromSignal()* the MSB position, which is 30 in the example case, is calculated. Since the signal is spread over three bytes the application has to combine the values for each byte. In the beginning, the first byte is computed. The computation of the mask value results in `128` or `0x80` since only the bit of the is of interest. The value is calculated with the help of the function *getBits()* which calls the function *getIntFromBits* implemented in Listing 5.10. In the current example, the computed value is equal to *1* for the first byte. The second byte is read with a mask value of `255` or `0xFF` since the whole byte contains data from the specified signal. Only updating the input values for the byte and bit offsets, the function *getBits* will be used again for the computation of the second byte value. The computation results in number *85*, but since one bit from the previous computed byte is already known and the computation is ranging from LSB to MSB, the number has to be multiplied by $2^{\text{number of previous calculated bits}}$. In this case this is equal to 2^1 and therefore the value from byte one is added to the value from byte two, resulting in the value *171*. The same procedure is used for the last byte. At first, the byte and bit offset for the new byte are computed. Afterwards the computation of the bit mask as well as the calculation of previously read bits have to be done. For the last byte, the resulting value *6*, which needs to be multiplied with $2^9 = 512$, results in *3072*. The summation of all values for each byte is equal to the transmitted value of *3243*. The algorithm works exactly the same for each amount of data bytes. The only computation which takes place for each new byte is the calculation of the bit mask and the remaining length to compute the required factor. In order to compute different endiannesses the computation has to be slighty changed. Therefore, four transformation functions for each endiannesses as well as their unsigned or signed status have been implemented.

5.6 Computing states & dispatch analysis event

The class *RapidStatusMonitor* has a *Handler* object which is specified in the Android API [3]. The *Handler* implements a *Runnable* object which triggers an update event in a specified update interval time. For each triggered update event it is possible to get a current snapshot of the underlying CAN_Data objects. The CAN_Data objects are containing the latest values which were received from the Bluetooth connection. Listing 5.11 shows the implementation of a listener which is invoked each time the event is triggered. The value of the CAN_Data object can be retrieved using the method *getActValue(key)*, where *key* is the name of the signal. This name is the specified key for the underlying *HashMap* object structure. Once the signals are retrieved, one of the smoothing algorithms described in Section 5.7 can be used.

```
public static PulseListener signalListener2 = new PulseListener() {
@Override
  public void onEvent(AbstractStatusMonitor monitor) {
    //Get the snapshot of values
    signalValueVelocity = monitor.getActValue("SpdVehAct");
    ...
    //Smoothing if enabled
    ...
    //Comuputing the state
    state = computeStates(...);
    //Trigger an event on the EventBus
    ...
  }
}
```

Listing 5.11: The *PulseListener* object which is invoked each time the Android Handler executes the Runnable. Once the reception is complete, the data values can be read and smoothed if necessary. Afterwards the state is computed and an event bus post is published.

In order to compute the state the measurement values are taken as an input for the *computeStates()* algorithm method. Finally, the computed state and the smoothed measurement values have to be send to the activities which are responsible for the implementation of the graphical representation. Therefore, an event is posted on an *EventBus*. The steps are elucidated in the following pages.

Computing states To compute the active state based on Section 5.1 the measurement values, which are present in each *CAN_Data* object, are used. Listing 5.12 shows the implementation of the computation algorithm. The measurement values *Vehicle Speed* (spd), *AcceleratorPedalPosition* (acc), *BrakePedalPosition* (brake), *BatteryCurrent* (ihv) and *BatteryVoltage* (uhv) are the input for the function.

```
private static int computeStates(double spd, double acc, double brake,
    double ihv, double uhv) {
  if (brake > 0.5 && acc <= 0.5 && spd > 2) {
    return 2;    //State Brake
  } else if (brake <= 0.5 && acc < 0.5 && spd > 2) {
    return 3;    //State Coastdown
  } else if (acc >= 0.5 && brake <= 0.5 && ihv*uhv < -500) {
    return 4;    //State Boost
  } else if (acc >= 0.5 && ihv > 0) {
    return 5;    //State LPM
  } else if (acc <= 0.5 && ihv > 0) {
    return 1;    //State Recuperation
  } else {
    return 0;    //State Zero
  }
}
```

Listing 5.12: This function is responsible for the state computation. It is used every time the *PulseListeners* onEvent method is invoked.

The algorithm has an integer output in the range from 0 to 5, where:

- 0 = State Zero
- 1 = State Recuperation
- 2 = State Brake
- 3 = State Coastdown
- 4 = State Boost
- 5 = State LPM

The state number as well as the five measurement values are then used to post an event on the *EventBus*.

EventBus *EventBus* is an Android optimized event bus, developed by *greenrobot*, that simplifies the communication between Activities, Fragments, Thread and more [30]. The general idea behind an *EventBus* system concept is illustrated in Figure 5.7. A certain amount of publisher nodes post events on the event bus which can be obtained by a consumer node if it has subscription to the event type.

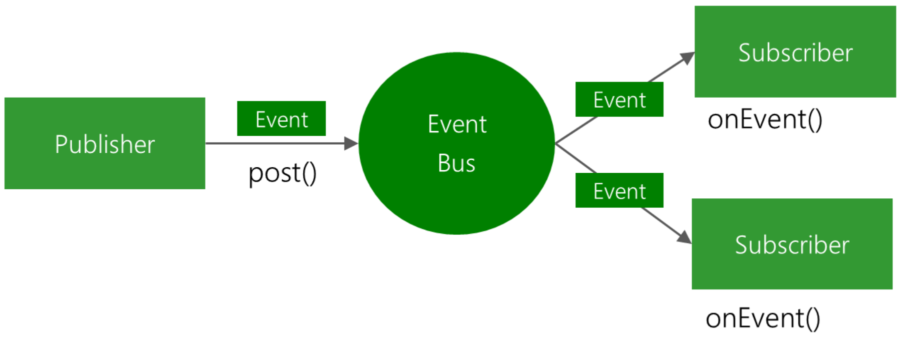


Figure 5.7: A publisher, which can be located in any class, posts an event and the *EventBus* transmits the event to every subscriber who is currently registered to the bus [23].

The *EventBus* simplifies the communication between Android components and decouples event senders and receivers. It also avoids complex and error prone dependencies and Android life cycle issues. Additionally, there is no boilerplate code which would have been necessary if the same architecture would have been realized with an observer pattern. Furthermore, the library provides features such as [23]:

- High performance
- Build time indexing of annotations
- Different thread delivery
- Zero configuration
- Configurable with the builder pattern

Because the library is easy to integrate into the application and the developer does not have to write a lot of additional boilerplate code with a self designed observer pattern structure, which might perform poorly, the choice fell on the greenrobot's *EventBus* library. In order to post messages on the event bus an event has to be defined at first. Figure 5.8 defines the plain old java object (POJO) for the event. It defines all chosen member variables as well as the constructor. With the *get()* methods it is possible to retrieve the content after an event is transmitted.
The second step is to register the subscriber to the event bus, shown in Listing 5.13. The subscriber can be any kind of Java class.

```
EventBus.getDefault().register(this);
```

Listing 5.13: Register the class as an *EventBus* subscriber.

Additionally, the subscription method has to be declared, as depicted in Listing 5.14. The subscribers implement event handling methods that will be invoked when an event is posted. These are defined with the *@Subscribe* annotation.

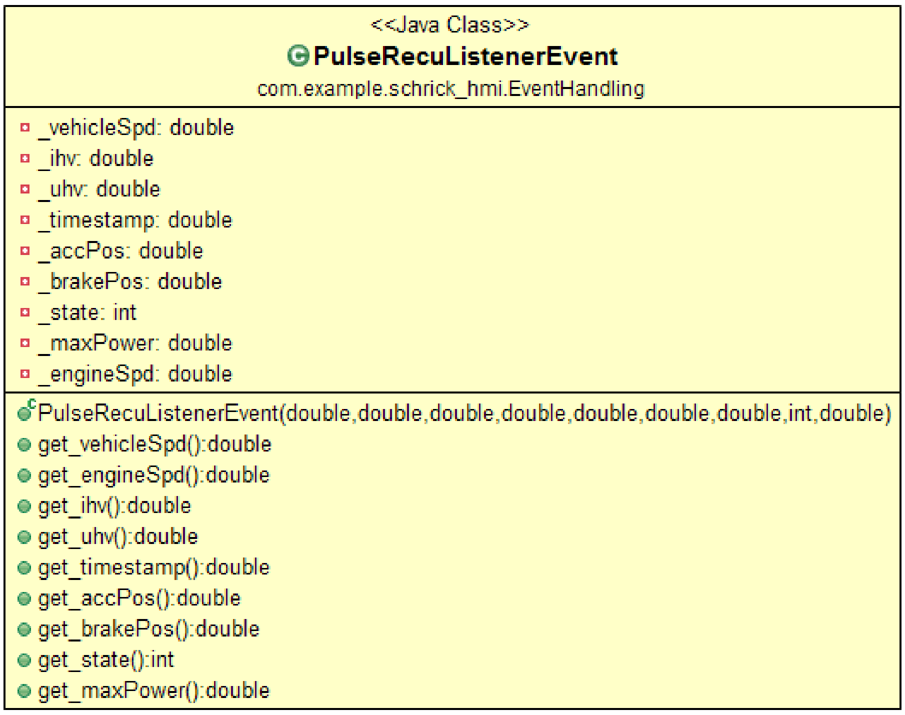


Figure 5.8: *Plain Old Java Object* (POJO) definition of the listener event which defines all the necessary values for computation purposes as well as *get()* methods.

```java
// This method will be called when a MessageEvent is posted
@Subscribe
public void onMessageEvent(PulseRecuListenerEvent event){
vehicleSpeed = event.get_vehicleSpd();
...
//Update graphical view with the new data points
}
```

Listing 5.14: Subscriber function to receive the transmitted bus event. The annotation *Subscribe* enables the function to be regarded as a listener for incoming events of the type *PulseRecuListenerEvent*.

EventBus can handle threading, which means that events can be posted in threads different from the posting thread. However, we have to keep in mind that changes to the user interface (UI) have to be done on the UI thread. But tasks which are time consuming can be shifted to another thread which lessens the workload of the main thread. Listing 5.15 shows the annotation changes which have to be made to define the thread in which the subscriber will be called.

```java
@Subscribe(threadMode = ThreadMode.POSTING)
@Subscribe(threadMode = ThreadMode.MAIN)
@Subscribe(threadMode = ThreadMode.BACKGROUND)
```

Listing 5.15: Different thread modes which are possible on the *EventBus*.

Upon choosing *POSTING* the subscriber will be called in the same thread who posted the event. Subscribers will be called in the main thread, which is often called the UI thread, when choosing *MAIN*. If the annotation is defined as *BACKGROUND*, the subscriber will be called in a background thread which will execute all its events sequentially. Once we have implemented and declared the subscriber methods, the last step is to actually post the event on the event bus. Listing 5.16 illustrates the posting definition.

```
EventBus.getDefault().post(new PulseRecuListenerEvent(...));
```

Listing 5.16: Posts an event on the *EventBus*.

The *post()* method is transmitted each time the *Runnable* is re-executed inside the Handler, which enables the application to continuously provide events to update the graphical representation in the respective Activity.

5.7 Post-processing and data smoothing

Because resolution of some signal values is not high enough, we need to smooth the data to avoid a ramp-like structure. This is especially important for the velocity signal. Since we are calculating the energy difference - explained in Section 5.10 - between two timestamps, the velocity resolution has to be as high as possible. Unfortunately, the signal is only able to provide changes in kilometers per hour as an integer value which is insufficient for the computation.
Smoothing is generally used to eliminate noise and filling in missing data values. Depending on the needs and the datasets, a suitable algorithm has to be chosen inside the application. In order to refine the small resolution of the velocity signal, multiple smoothing algorithms are implemented. Only one can be activated at a time, but it is possible to change the algorithm at runtime. The application does not have to be restarted to show the variously smoothed data.

5.7.1 Moving average smoothing

Moving average is a basic and simple smoothing algorithm. In its simplest form it is the unweighted mean of the previous n data points. Commonly, the mean is taken with an equal number of data values on either side of a so called *central value* [32]. It is given by the formula

$$s_t = \frac{1}{n}\sum_{t-n}^{t} x_t \text{ with } t \geq n, \tag{5.1}$$

where s_t is the current smoothed statistics and x_t the observation value at time t. This approach needs a cold start time with at least n elements before it can begin to compute results. Moving average is working like a window frame which iterates over data points. Once the threshold of the cold start phase is overcome, a new value will replace the oldest one. As an example, we define $n = 5$, so it is necessary to wait for at least five incoming data values to start the algorithm. If we assume a data array with values $1, 2, 3, 4, 5, 6$, only the last five elements are in the frame which is moving to the right once a new value is inserted. The first element in the previously defined array is therefore not necessary for the computation anymore. Furthermore, we do not need to persist more than n elements to achieve the results. However, there is a big drawback in the moving average approach. The average age of this approach is equal to

$$age_{movavg} = \frac{n+1}{2} \cdot \text{frequency of data update in [ms]}. \tag{5.2}$$

This is the amount of time which the data series will lag behind the original one. If we assume $n = 31$ and an update frequency of 50 Hz, the delay of the forecast at time t is 320 ms which could be too big for certain use-cases. Therefore, a suitable frame length n has to be considered when using this algorithm.

Implementation

The algorithm described above was implemented for the application. Listing 5.17 shows the code realization of the algorithm. In line 1 an array, which has the ability of a circular ring buffer, is declared. That means the array contains n values, before a new element replaces the oldest one. In the defined function *movingAverageSmoothVelocity* the smooth statistic is the return value, which is computed by using formula (5.1). By iterating over the array, every entry is summed up. The calculated value is divided by the number of observations which are to be considered by the specification of the length n.

```java
1  private ArrayCircularBuffer outputMovingAverageVelocity;
2
3  public double movingAverageSmoothVelocity(double inputValue) {
4    double result = 0;
5    outputMovingAverageVelocity.insert(inputValue);
6    for (int i = 0; i < outputMovingAverageVelocity.size(); i++) {
7      result += (double) outputMovingAverageVelocity.getItem(i);
8    }
9    return (result / outputMovingAverageVelocity.size());
10 }
```

Listing 5.17: Implementation of the simple moving average smoothing algorithm for the velocity signal in Java.

With the chosen design of the algorithm implementation it is possible to generate smoothed forecast values during the cold start phase. Keeping that in mind, the first value which is not error-prone is calculated when the measurement is running long enough to fill the array with n data values. The value for the frame length n needs to be specified by the user depending on the vehicle and the signal resolutions. This can be done in the application's options menu. Listing 5.18 shows a code snippet which elucidate the internal initialization as well as functions calls. The activation of the *average smoothing* algorithm as well as the frame length n has to be selected at the option menu. Once the measurement starts, the array is initialized with the length n, which can be seen in line 2. As long as new input values are triggering and the cold start phase filled the array with n values, the returned values can be used for a correct computation. But if the cold start phase is active, which is illustrated in line 5, a value is returned due to the implementation method, which should not be used for further computation.

```
1 double velocity_value = 0;
2 initArray(n);
3 while(newValue){
4   velocity_value = rMon.movingAverageSmoothVelocity(inputValue);
5   if(counter_of_cold_start_values > n){
6     // Do something with the smoothed value
7   }
8 }
```

Listing 5.18: Example of moving average function calls and initialization.

5.7.2 Exponential smoothing

Exponential smoothing is a standard technique for smoothing time series data. It is an easily learned and applied procedure for approximatively calculation of values. It is commonly applied to smoothen data, acting as a low-pass filter to remove high-frequency noise. Since we have signals with high-frequency noise, this algorithm might be the most suitable for the approximation. The simplest form of exponential smoothing is given by the formula

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1}, \tag{5.3}$$

where s_t is the current smoothed statistics by building a simple weighted average of the current observation x_t and the previously smoothed statistic s_{t-1}. The smoothing factor α ranges between 0 and 1. Values of α close to one have less of a smoothing effect and give a greater weight to recent changes in the data, while values of α closer to zero have a greater smoothing effect and are less responsive to recent changes. It can be seen that exponential smoothing produces a smoothed value as soon as two observations are available [32].

Deriving the formula

The time series data are build recursively. Theoretically, a data series at time t already has infinite previous values. Practically, we define a certain start value for s_0 with $t = 0$. This start value greatly depends on the behavior of the data series. Exponential smoothing puts substantial weights on past observations. So the initial value of demand will have a large effect on early forecasts. However, in order to overcome this weakness and to determine a good starting value, the smoothing series can run a certain amount of time which results in a better approximation because some values are already known [32]. By listing the data series recursively, starting with s_0, we get the following equations:

$$\begin{aligned} s_1 &= \alpha x_1 + (1-\alpha)s_0, \\ s_2 &= \alpha x_2 + (1-\alpha)s_1, \\ s_3 &= \alpha x_3 + (1-\alpha)s_2, \\ \ldots \end{aligned} \tag{5.4}$$

If we solve the recursive function by substituting the values for s_t, we get equations (5.5) and (5.6) for the times t and $t-1$, respectively.

$$s_t = \alpha x_t + \alpha(1-\alpha)x_{t-1} + \alpha(1-\alpha)^2 x_{t-2} + \alpha(1-\alpha)^3 x_{t-3} \ldots \tag{5.5}$$

$$s_{t-1} = \alpha x_{t-1} + \alpha(1-\alpha)x_{t-2} + \alpha(1-\alpha)^2 x_{t-3} + \alpha(1-\alpha)^3 x_{t-4} \ldots \tag{5.6}$$

Multiplying (5.6) with $(1-\alpha)$ yields:

$$(1-\alpha)s_{t-1} = \alpha(1-\alpha)x_{t-1} + \alpha(1-\alpha)^2 x_{t-2} + \alpha(1-\alpha)^3 x_{t-3} + \alpha(1-\alpha)^4 x_{t-4} \ldots \tag{5.7}$$

If we subtract (5.7) from (5.5), the resulting equation reads

$$\begin{aligned} s_t - (1-\alpha)s_{t-1} &= \alpha x_t \\ \Leftrightarrow s_t &= \alpha x_t + (1-\alpha)s_{t-1}, \end{aligned} \tag{5.8}$$

which is exactly the formulation of the recursively defined function (5.3).
As time passes the smoothed statistic s_t becomes the weighted average of a greater and greater number of the past observations x_{t-n}, and the weights assigned to previous observations are in general proportional to the terms of the geometric progression $\{1, (1-\alpha), (1-\alpha)^2, (1-\alpha)^3, \ldots\}$. A geometric progression is the discrete version of an exponential function, hence the name exponential smoothing [32].

Implementation

A simple exponential smoothing algorithm was included in the application. It implements equation (5.3) with regard to a starting value which can be set independently.
Listing 5.19 shows the exact implementation of the algorithm. The array `output_vel` holds the smoothing statistics for $t-1$. At the beginning, the starting value has to be initialized with the function `initExpSmoothVel` which puts the given double value into the respective array element. In case of new velocity input values the function `expSmoothVel` computes the smoothed value with regard to the previously calculated smoothing statistic in array element zero and the factor α. The resulting computation is saved into both array elements and given as a result value for further post-processing.

```java
private double[] output_vel = new double[2];

public double initExpSmoothVel(double value) {
  output_vel[0] = value;
  return value;
}

public double expSmoothVel(double inputValue, double ALPHA) {
  output_vel[1] = inputValue * ALPHA + (1 - ALPHA) * output_vel[0];
  output_vel[0] = output_vel[1];
  return output_vel[1];
}
```

Listing 5.19: Implementation of the exponential smoothing algorithm for the velocity signal in Java.

Listing 5.20 illustrates a simple example on how to use the functionality. While there is a continuous stream of new signal values, the values are constantly smoothed by using them as an input value for the algorithm. If it is not initialized, the function sets a given starting value beforehand. Afterwards, the computed value can be used for the representation on a chart or for further calculations.

```java
double velocity_value = 0;
while(newValue){
  if(!initExpSmooth){
    velocity_value = rMon.initExpSmoothVel(initValue);
    initExpSmooth = true;
  }else{
    velocity_value = rMon.expSmoothVel(inputValue,ALPHA);
  }
  // Do something with the smoothed value
}
```

Listing 5.20: Usage example of the exponential smoothing algorithm for the velocity signal in Java.

5.7.3 Binomial smoothing

Binomial smoothing is essentially the same as a weighted moving average smoothing algorithm, but with a binomial weighting distribution. The considered window frame is defined by a threshold value. Like the moving average approach, the binomial smoothing algorithm needs a cold start time to fill the frame with enough data points. The difference between the moving average and the binomial smoothing approach is that the values are summed up with a binomial weighting function in the latter case. For large numbers of applications, the weights become Gaussian and the smoothing approximates Gaussian kernel smoothing [4]. The binomial smoothing function is defined as

$$s_t = \frac{1}{n} \sum_{t-n}^{t} x_t \cdot w_t \text{ with } t \geq n, \tag{5.9}$$

where s_t is the current smoothed statistic, x_t is the observation value at time t and w_t is the corresponding weight value at time t. The weight values are provided by repeatedly applying the moving average filter for two elements that have the weights $(\frac{1}{2}, \frac{1}{2})^l$. For example, with $l = 4$ applications the binomial filter weights are given by $(\frac{1}{2}, \frac{1}{2})^4 = (\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16})$. This is the same as the *pascal triangle*, except the denominator which normalizes the weights [4]. There is the same drawback as in the moving average approach. The average age of a data value with the binomial filter approach is equal to

$$age_{bin} = \frac{n+1}{2} \cdot \text{frequency of data update in [ms].} \tag{5.10}$$

Therefore, a suitable frame length is as important as in the moving average approach.

Implementation

The binomial smoothing algorithm was implemented as a data filter alternative for the application. Listing 5.21 illustrates the code realization to achieve the calculation. An *ArrayCircularBuffer* provides a ring buffer, which means that after initialization with a certain threshold value the latest incoming element will replace the oldest one without the array growing in size. The array *binomCoeff* contains the weight values for each element in the window frame. The weighting Array was computed upon the initialization of the ring buffer. The code is illustrated in Listing A.1 in the appendix. The code conversion from equation 5.9 can be seen in line 8. Each element in the ring buffer is summed up and multiplied with its weighting value.

```
 1 private ArrayCircularBuffer outputBinomVelocity;
 2 private static double[] binomCoeff;
 3 
 4 public double binomSmoothVelocity(double inputValue) {
 5   double result = 0;
 6   outputBinomVelocity.insert(inputValue);
 7   for (int i = 0; i < outputBinomVelocity.size(); i++) {
 8     result += binomCoeff[i] * (double) (outputBinomVelocity.getItem(i));
 9   }
10   return result;
11 }
```

Listing 5.21: Implementation of the binomial smoothing algorithm for the velocity signal in Java.

An example of the initialization order as well as the calculation of the weighting function is given in Listing 5.22. After the ring buffer and the weighting function are generated, a new input value can be added each time a new data point is retrieved. The value should not be used until the threshold value for the cold start phase has been passed.

```
double velocity_value = 0;
initArray(n);
pascal(n);
while(newValue){
  velocity_value = rMon.binomSmoothVelocity(inputValue);
  if(counter_of_cold_start_values > n){
    // Do something with the smoothed value
  }
}
```

Listing 5.22: Example of binomial filter function execution and initialization.

5.7.4 Event post-processing

Once the measurement has ended, the application computes event objects which are based on the previous computed vehicle states. While the analysis is able to provide event computation in real-time, it is currently not possible to relate equal states to an event that lasts an unspecified amount of time. However, this is possible after the measurement has ended because the data object, which is responsible for storing the measurement data, is accessible. Figure 5.9 shows the designed structure in an UML diagram.

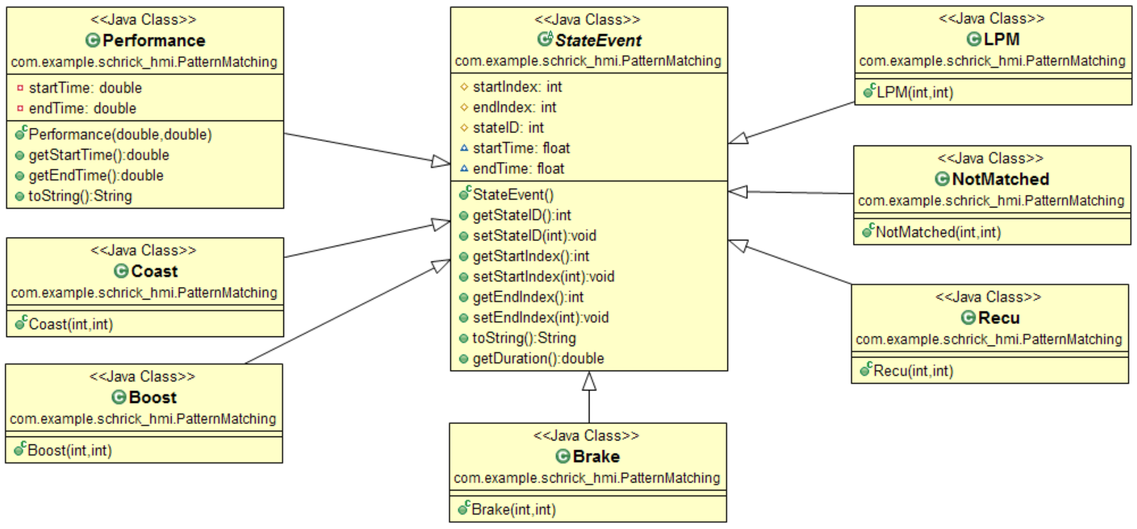
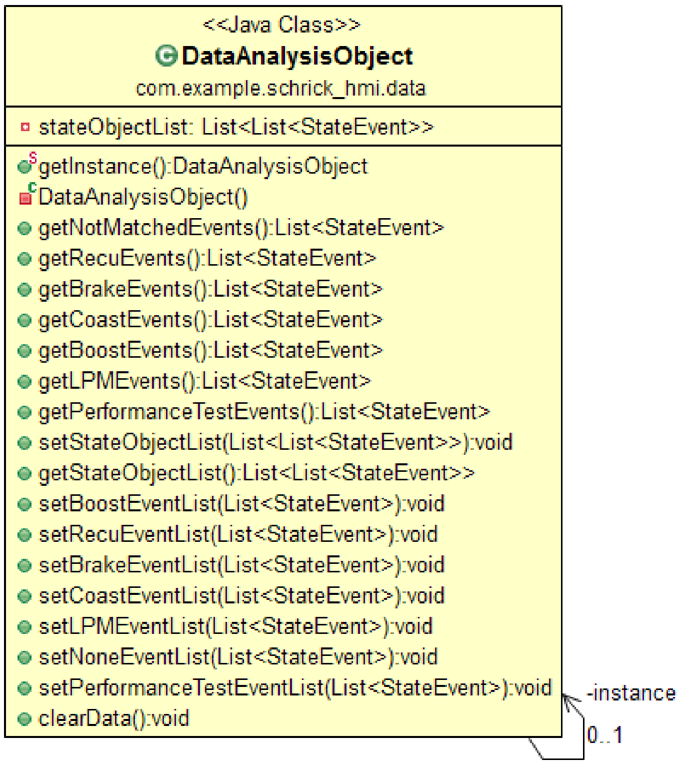


Figure 5.9: UML diagram of the used states and their connection. The *StateEvent* class is the superclass for all other classes. Each application state has an own class with which we are able to persist the events. A creation of an object needs the start- and end-time of an event which enables us to get the respective data out of the underlying data object.

Each state which was defined in Section 5.1 has a single class with a superclass *StateEvent*. An additional event state *Performance* was created to make it possible to save vehicle

performance tests which are described in Section 6.4. With the help of the implemented algorithm illustrated in Listing A.2 we are able to get the start and end times of an event. By iterating over the state array of the data object it can be checked whether a state at timestamp t is followed by a state at timestamp $t + 1$. If the states are the same they belong to the same event. Once a state differs from the previous one the event is over and a new one starts.
The resulting event-data objects are stored into the *DataAnalysisObject* (Figure 5.10) which is used when the Android Activities are called in an offline mode, meaning there is no ongoing measurement. The *DataAnalysisObject* can also be persisted for later usage.



<<Java Class>>
DataAnalysisObject
com.example.schrick_hmi.data
stateObjectList: List<List<StateEvent>>
getInstance():DataAnalysisObject
DataAnalysisObject()
getNotMatchedEvents():List<StateEvent>
getRecuEvents():List<StateEvent>
getBrakeEvents():List<StateEvent>
getCoastEvents():List<StateEvent>
getBoostEvents():List<StateEvent>
getLPMEvents():List<StateEvent>
getPerformanceTestEvents():List<StateEvent>
setStateObjectList(List<List<StateEvent>>):void
getStateObjectList():List<List<StateEvent>>
setBoostEventList(List<StateEvent>):void
setRecuEventList(List<StateEvent>):void
setBrakeEventList(List<StateEvent>):void
setCoastEventList(List<StateEvent>):void
setLPMEventList(List<StateEvent>):void
setNoneEventList(List<StateEvent>):void
setPerformanceTestEventList(List<StateEvent>):void
clearData():void
-instance
0..1

Figure 5.10: The singleton *DataAnalyisObject* stores all information about an event.

5.8 Data storage & persistence

All measurement values have to be stored somewhere in order to load and analyze them again. Since the amount of data points the chart can hold is limited (see Section 5.9) the whole test cycle cannot be displayed and therefore it is necessary to design a structure for data persisting purposes. Figure 5.11 illustrates the Java object *DataObject* which can save measurement data values for up to nine signals. Furthermore, it contains additional elements to save the timestamps as well as the computed states. Once a measurement is started the object goes into a *isRunning* state which disables read calls and further computation on the underlying data structure. The idea behind the concept is pretty straightforward. Because the saved data values are not being used during the ongoing measurement, there is no need to worry about thread concurrency and synchronization as well as concurrent read and write calls on the chosen *List* structure. If we want to read the data values we just have to make sure the object is not in an *isRunning* status. But this is only the case when the measurement was stopped or has ended. As a data container for each signal a simple *ArrayList* was defined. *ArrayList* is a general list implementation suitable for most use cases. It is backed by an Object array, which size is dynamically adjusted while the user adds or removes elements from the list. The operation *add()* runs in amortized constant time. This means that the average insertion time at the end of the list across the entire runtime is constant, in this case *O(1)*. However, the array copying overhead grows significantly as the size of the collection increases because the number of elements that need to be copied with each insertion increases. The *ArrayList* needs to be copied when the capacity is exhausted and the internal array has to be extended. If we manually set a minimum capacity we may inflate the random access memory (RAM) but improve the performance. This is not used in the current implementation, but might be good for future investigations.

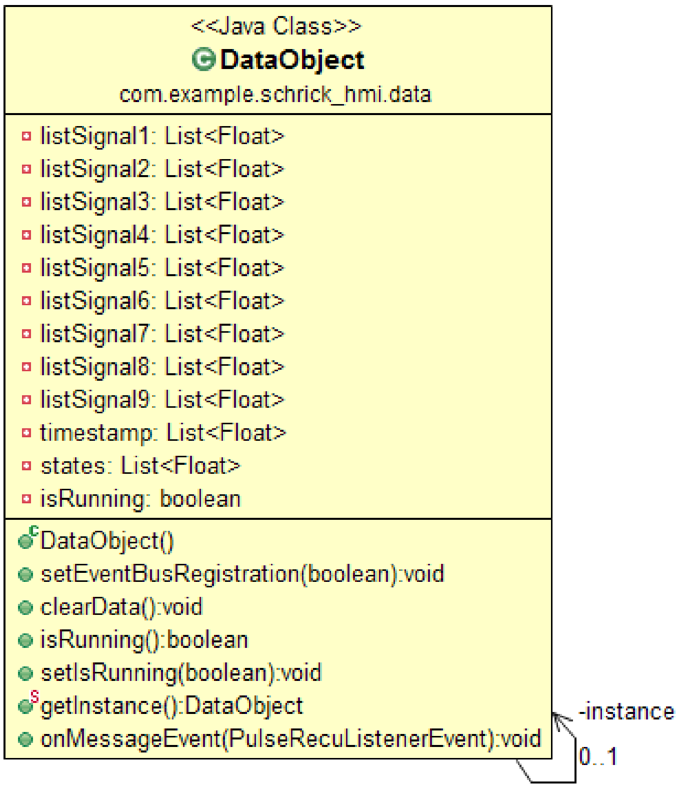


Figure 5.11: The singleton *DataObject* stores all measurement data into the corresponding arrays. Each signal has its own array.

The *DataObject* is a subscriber to the *PulseRecuListenerEvent* which is posted on the event bus (see Section 5.6). Each time the *MessageEvent* is called the measurement values are stored into the corresponding *ArrayLists*. Listing 5.23 shows the implementation of the *MessageEvent* in the *DataObject* class. The method is called as soon as the *PulseRecuListenerEvent* is posted on the bus and the annotation indicates that it is called on a separate background thread. With this implementation it is guaranteed that the main thread is not blocked in the data persistence process.

```
@Subscribe(threadMode = ThreadMode.BACKGROUND)
public void onMessageEvent(PulseRecuListenerEvent event){
  //Save data into corresponding arrays
}
```

Listing 5.23: Declaration of the subscriber method for the *DataObject* class.

In order to save the data on the device an *ObjectOutputStream* serializes the objects which are given as a *List<Object>* input parameter. The list contains the *DataObject* as well as the *AnalysisDataObject*. The resulting file is stored with the extension *.dat* on the external file system of the Android mobile device. It is possible to load saved data files and to analyze the whole data set in an offline mode. The system is only able to save the data input when no measurement is active at the moment. After the measurement has ended and post-processing is finished the same analysis features are provided without saving the data beforehand. However, the data is lost if the application is closed or restarted without prior storage.

5.9 Chart API & visualization

In order to visualize the smoothed and computed values a chart view API, which is able to provide real-time data updates, is needed. The first choice was the API *MPAndroidChart* from Philipp Jahoda because it supports real-time updates up to 30.000 data points and its core features are very good [29]:

- 8 different chart types
- Scaling on both axes possible
- Able to combine chart types
- Separate and customizable axes
- Highlighting values
- Save chart as picture
- Customizable legends
- Animations for axes
- Limit lines
- Customizable paints, background, gestures

These feature make the chart API an excellent choice for the representation of the analysis layouts. The possible chart types are: LineChart, BarChart, PieChart, ScatterChart, CandleStickChart, BubbleChart, RadarChart and the CombinedChart. The use of a chart and its continuous filling with data points requires a definition in the extensible markup language (XML) layout file of the corresponding activity. Listing 5.24 illustrates the implementation in the XML file. A *LineChart* object, which is dimensioned like its parent container, is specified and equipped with an additional identification name so it can be referenced in the source code implementation.

```xml
<com.github.mikephil.charting.charts.LineChart
    android:id="@+id/linechart"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Listing 5.24: Definition of the chart type which should be used in the view. The XML specification implements a linechart with the sizes for both dimensions being the same as the parent container.

Once the XML layout file specifies the necessary positioning of all user interface (UI) components, we are able to use the identification name as a reference to retrieve the *LineChart* object for our activity. Listing 5.25 shows the creation of the *LineChart* object *mLineChart*.

```java
LineChart mLineChart = (LineChart) findViewById(R.id.linechart);
```

Listing 5.25: Retrieve the LineChart object in the activity by using the XML reference.

Now we have to add data to the chart. Listing 5.26 gives a rough overview of an easy implementation to realize real-time updates to the chart. Firstly, we have to create an initial data object which holds the data sets of a single chart. Afterwards, the data object needs to be added to the chart. However, the data object alone contains no values. Therefore, a suitable amount of underlying data sets have to be created, where each data sets can contains values for one signal. Line 7 is defining a new *LineDataSet* with no initial values (null) and the name "Vehicle Speed [kph]" for the data series. It is then added to the data object. To really add data to the chart, we have to add an *Entry* object - which consists of the current measurement value and the position it should be added to - on the corresponding *LineDataSet*. Line 11 adds an X-axis value with the current timestamp to the chart, whereas line 12 adds the corresponding Y-value to the *LineDataSet* and therefore to the chart as well. The function *getEntryCount()* gives the number of entries which are currently being saved in the *LineDataSet* object. A new *Entry* object is always placed at the last position. Line 13 and 14 are necessary in order to notify the chart about the change of data and to invalidate the graphical representation. To achieve real-time updates line 11 to 14 have to be repeated each time a new value is incoming.

```java
1 //Creating a new LineData set
2 LineData mData = new LineData();
3 //Setting it to the chart
4 mLineChart.setData(mData);
5
```

```
//Defining and setting an underlying data set for mData
ILineDataSet mIset = new LineDataSet(null,"Vehicle Speed [kph]");
mData.addDataSet(mIset);

//Adding an entry and refreshing the chart
mData.addXValue(String.valueOf(timestamp));
mData.addEntry(new Entry(vehSpeed,mIset.getEntryCount()),0);
mLineChart.notifyDataSetChanged();
mLineChart.invalidate();
```

Listing 5.26: Creates a new data object and sets it to the chart. To be able to visualize values of a specific signal, a LineDataSet is created and given as a reference to the data object. After adding a new X and Y value, the chart is notified of the change in the underlying data structure and will be invalidated.

Listing 5.27 shows another necessity for a moving frame: A maximum threshold value for visible X-axis entries. Afterwards the viewport is moved to the right so that the latest value is always positioned at the right of the chart frame.

```
//Get a moving chart (500 values), latest value on the right side
mLineChart.setVisibleXRangeMaximum(500);
mLineChart.moveViewtoX(mData.getXValCount()-501);
```

Listing 5.27: Moves the viewport of the chart with regards to a threshold value which is equal to 500.

An important thing to keep in mind is that the chart representation is only running smoothly up to 30.000 data points. So depending on the update cycle time we cannot necessarily hold all the data for a test drive. This limitation does not have a great influence on the analysis feature because for looking at the values in real-time there is no reason to hold the measurement values for such a long amount of time. Once the measurement ended the whole data, which were persisted in a background thread, can be loaded and presented accordingly. In order to get a smooth running application, we need to remove the first Y-entry object and X-axis value when a certain threshold of measurement values is hit. Basically, the oldest value at the beginning is replaced with the latest update which guarantees a stable amount of visualized data points.

5.10 Theoretical energy consideration

To offer an additional analysis feature besides the illustration of the energy consumption or recuperation, a new method for calculating the theoretical amount of power present at the wheel was developed and implemented. In order to get that value as a reference for the energy efficiency the difference in kinetic energy is considered. The kinetic energy of a vehicle is the energy it contains due to its motion. The vehicle maintains the amount of kinetic energy unless the speed changes. The difference in the kinetic energy of two successive velocity values is the energy which could have been recuperated in an ideal system without efficiency losses. The kinetic energy is given by

$$E_{kin}(t) = \frac{1}{2}mv(t)^2,$$

where t is the current timestamp, m is the mass of the vehicle and v is the current velocity. To compute the change in the maintained energy while the vehicle is driving with decreasing velocity (recuperation process), the kinetic energy at timestamp t_i is subtracted from the kinetic energy at the previous timestamp t_{i-1}:

$$\begin{aligned} E_{theo}(t_i) &= E_{kin}(t_i) - E_{kin}(t_{i-1}) \\ &= \frac{1}{2}m(v(t_i)^2 - v(t_{i-1})^2). \end{aligned}$$

To display power instead of energy the energy difference has to be divided by the time-interval difference from t_i and t_{i-1}, since the dimension of power is energy divided by time. This results in the following equation:

$$P_{theo}(t_i) = \frac{\frac{1}{2}m(v(t_i)^2 - v(t_{i-1})^2)}{t_i - t_{i-1}}.$$

P_{theo} is a first usable value which can be presented in the application charts in order to compare the actual power and the theoretical power at a given deceleration process. However, this is not a representative value because there are several efficiency losses in the process of the energy conversion. We are able to filter out at least some of them. Each vehicle loses on efficiency because of the aerodynamic drag and the rolling friction of the tires. The resulting force is defined as

$$F_{wheel}(v) = av^2 + bv + c,$$

where a, b, c are efficiency coefficients to describe the resulting quadratic function which is velocity dependent. When the velocity value growths, the resistance gets bigger as well, which results in an over all quadratic growth. The coefficients are calculated by accelerating the vehicle to a certain velocity and then letting it roll out until standstill. With enough measurements an approximation can be achieved. To get the amount of power which cannot be used for the recuperation process, the force is multiplied with the current velocity $v(t_i)$, resulting in

$$P_{wheel}(t_i) = av(t_i)^3 + bv(t_i)^2 + cv(t_i).$$

Finally, the difference between the theoretical power P_{theo} and the power which is lost at the wheel due to resistance losses P_{wheel} yields a better suited value for the efficiency comparison:

$$\begin{aligned} P_{result}(t_i) &= P_{theo}(t_i) - P_{wheel}(t_i) \\ &= \frac{\frac{1}{2}m(v(t_i)^2 - v(t_{i-1})^2)}{t_i - t_{i-1}} - av(t_i)^3 - bv(t_i)^2 - cv(t_i). \end{aligned}$$

However, there are still efficiency losses which are not considered in this approach. Additionally, the current approach only works on a route without slope because there are no hardware sensors which can measure the orientation and the inclination of the vehicle. Such sensors would be a great addition for further iterations of the application. The coefficients a, b, c and the vehicles mass have to be entered into the options *Params* layout and they have to be changed for each time the vehicle changes.

Chapter 6

Analysis & Usage

This chapter elucidates the analysis process of some key components which were important for the implementation and gives a short explanation of the bottlenecks of data transferring over bluetooth as well as a quick guideline to use the application. In the first part, the analyses of battery usage, bluetooth latency and state detection is introduced with further details and results. The second part deals with the introduction of the application and its functionality and representation.

6.1 Battery consumption analysis

An important factor for mobile devices is the battery consumption of an application. The goal is to minimize the power drainage while using the implemented functionality. This is per se not a trivial task. Some restrictions arise from existing hardware components which are required for the data communication, like bluetooth.
Since we are using the bluetooth protocol to transfer data to the mobile device and vice versa, we are constantly streaming data and therefore use electrical energy to achieve this task. With the chosen implementation it is possible to connect/disconnect the bluetooth connection and with the concept of *Weak References* (see Chapter 5) the usage of the update listener objects is held at a minimum while being active. In order to test the battery consumption, benchmarks with various starting states of charges have been performed. The test bed was a HTC Nexus 9 with a Dual-Core 2.3 GHz Denver processor and 2GB RAM operated on Android Marshmallow (6.0.1). The battery capacity amounts to 6700-mAh [27]. All connectivity functions, like WLAN or GPRS, were disabled during the benchmarks. The display brightness was set to the highest value to generate a worst case calculation.
Listing 6.1 shows the implementation of a receiver object which fetches the values for battery-level and device-temperature. The values are stored into an ArrayList each time a battery changed event is broadcasted from the Android system. An internal timestamp is

added for each of those events. Line 14 to 17 show the registration and bounding process to the specific battery changed event.

```
1  private BroadcastReceiver mBatInfoReceiver = new BroadcastReceiver(){
2    @Override
3    public void onReceive(Context c, Intent i){
4      //Battery level
5      int level = i.getIntExtra("level", 0);
6      //Device Temperature
7      double temperature = i.getIntExtra("temperature", 0) / 10;
8      time = SystemClock.elapsedRealTime() / 1000 - initTime;
9      batteryLvl.get(0).add(time);
10     batteryLvl.get(1).add(level);
11     batteryLvl.get(2).add(temperature);
12   }
13   //Create filter to get an update after battery status changes
14   IntentFilter mIntentFilter = new IntenFilter();
15   mIntentFilter.addAction(Intent.ACTION_BATTERY_CHANGED);
16   //Register the receiver
17   registerReceiver(mBatInfoReceiver, mIntentFilter);
18 }
```

Listing 6.1: Sets up a BroadcastReceiver to listen for incoming battery changes.

After the end of the measurement the ArrayList was exported as a comma-separated-value (csv) file for further usage. A least squared fitting, in which the sum of the squared differences between the model curve and the measurement values is minimized, results in the third-degree polynomial model for the battery consumption based on the decreasing state of charge

$$SoC = -1.462 \cdot 10^{-11} t^3 + 1.620 \cdot 10^{-7} t^2 - 7.345 \cdot 10^{-3} t + 96.081.$$

Figure 6.1 presents the experimental data of the battery consumption extracted out of the generated export files. The state of charge SoC is plotted against the time t and it can be seen that the battery consumption does not follow a linear discharge trend. The battery consumption behaves linear up to a certain degree but changes its behavior at a state of charge of around 48 %. Multiple evaluations showed that a third-degree polynomial curve fits the measurement values best. According to this model, the battery is completely discharged after approximately 210 minutes.

Unfortunately, the applications run time is much smaller than the specified 570 minutes average indicated in the Nexus 9 technical specifications [27]. However, the average battery service life does not correspond to the operating time under full load with the highest display brightness. Additionally, the 210 minutes are more than enough for every test an engineer can perform with a test vehicle.

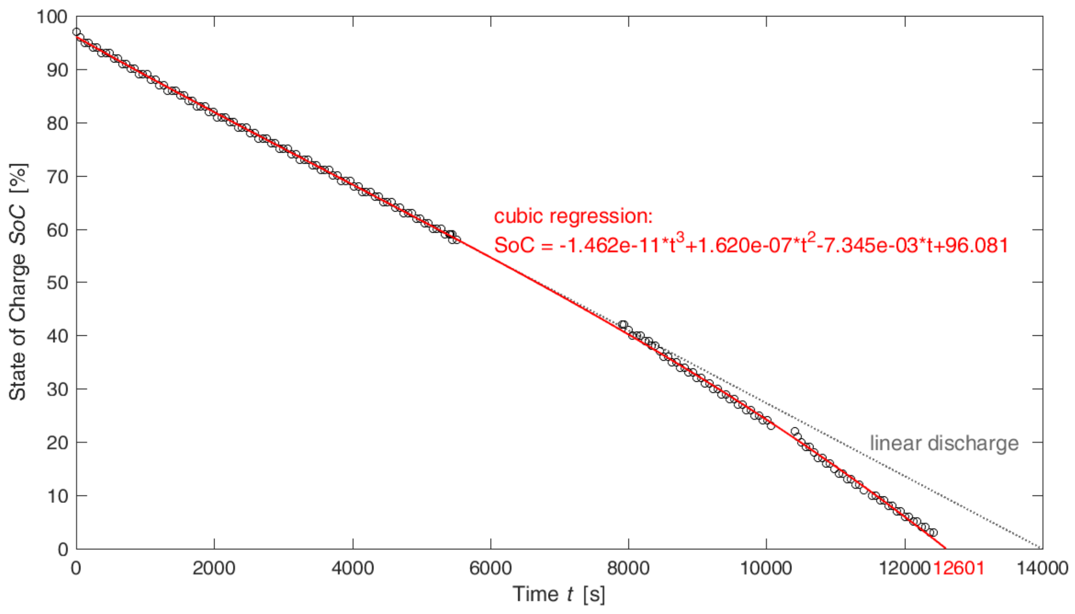


Figure 6.1: Time dependent battery level discharge of HTCs Nexus9 tablet. The stated third-degree polynomial function fits the measurement values in a least-squares sense.

In order to verify our measurement results and to give additional insights, the same test was executed on a Samsung Galaxy Tab S2 tablet with an Octa-Core processor (4 x 1,9 GHz + 4 x 1,3 GHz) and 3 GB RAM operated on Android Marshmallow (6.0.1). The battery capacity is equal to 5870 mAh and therefore less, in comparison to the HTCs 6700-mAh [38]. All other features are turned off in order to guarantee the same test conditions. Figure 6.2 shows the illustration of the exported measurement values. Once again the state of charge is plotted against the time t and a least squared fitting results in the following second-degree polynomial model for the battery consumption:

$$SoC = -3.590 \cdot 10^{-8} t^2 - 3.670 \cdot 10^{-3} t + 100.249.$$

According to this model the battery is completely discharged after approximately 373 minutes. A comparison of both battery performances shows that the battery consumption is not entirely explainable by the Bluetooth power consumption. Furthermore, the CPU load has an additional effect. During the measurement, the HTC device was constantly working under a performance load between 70 % and 80 % while the Samsung device operated under a performance load between 15 % and 20 %. This indicates that the battery consumption is related to the computation and rendering of the graphical representation which can be dealt with if we get a better device.

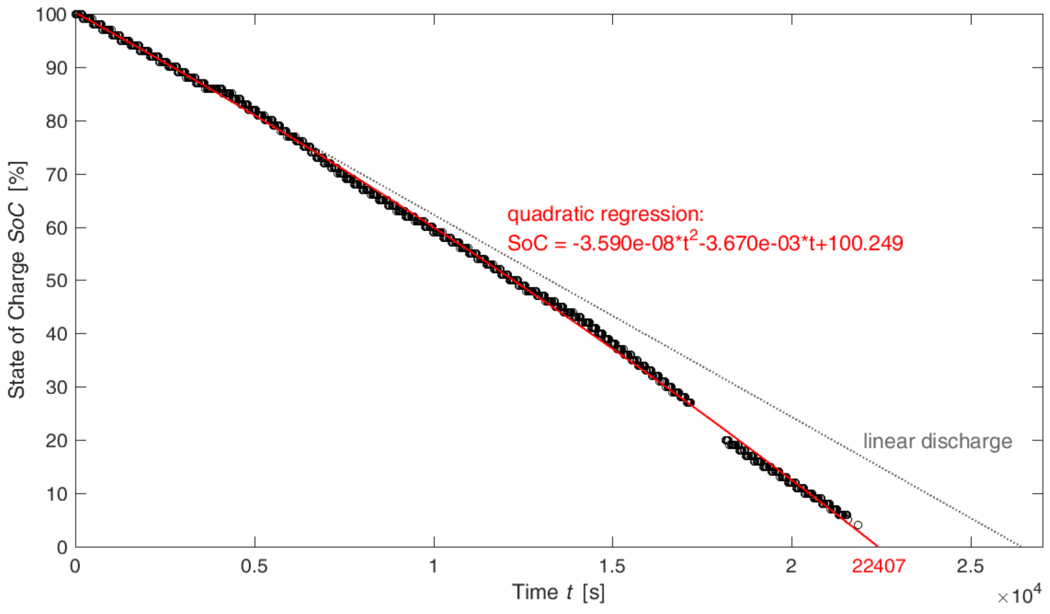


Figure 6.2: Time dependent battery level discharge of Samsungs Galaxy Tab S2 tablet. The stated second-degree polynomial function fits the measurement values in a least-squares sense.

6.2 Bluetooth latency analysis

An important factor for real-time applications is the latency between data messages. In the current case, CAN messages are transmitted over a Bluetooth connection. Once the message is received, the application performs multiple calculation within the analysis scope of this thesis. Therefore, a reasonable latency has to be predominant to ensure a sound graphical representation. An Android mobile device has the feature to enable a Bluetooth connection snoop process, where it logs all incoming and outgoing Bluetooth tranfer. It can be enabled in the developer options menu of Android since version 4.4. When the analysis process has finished populating the capture file by running the application being tested, the file generated by Android and saved into the external storage of the device can be analyzed. The program *Wireshark* was used to analyze the resulting log file. Table 6.1 provides an excerpt of the snoop file with an additional column with the time difference (Δt) between two messages. Each row of the table contains a single message package which was received by the Android mobile device with information about the source point, the destination, the used protocol and the message length. Considering the whole Bluetooth snoop file it can be calculated that the average time between messages is approximately 11 ms. However, the CAN bus is configured to transmit for each node in a 10 ms interval. This means that the Bluetooth connection throughput is rather slow compared to the CAN bus speed. If we keep in mind that CAN messages are transmitted subsequently with regards to their identification number priority, it might be possible for a higher prioritized message to block out a message with a lower priority. This means that messages with a higher identification number may have some internal delay as well. Another abnormality, which could not be solved yet, are some high message transmission time differences at random times.

Time t [s]	Source	Destination	Protocol	Length [Byte]	Info	Δt [s]
83,273983	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001239
83,275222	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001241
83,276463	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,496287
83,77275	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001209
83,773959	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,00126
83,775219	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,496297
84,271516	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001191
84,272707	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,00125
84,273957	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,497145
84,771102	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001856
84,772958	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001069
84,774027	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,495751
85,269778	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,002013
85,271791	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,001237
85,273028	(CANlink Bluetooth)	localhost ()	SPP	29	Rcvd UIH Channel=1	0,000855

Table 6.1: Excerpt of a Bluetooth snoop file recorded by a mobile device with additional calculated time differences between two message packages.

As can be seen in Table 6.1 there are four rows in which the message transmission time difference is rather high with close to half a second. However, there are periods of time where none of these occur for minutes. If we calculate an average excluding these outliers, the resulting message transmission speed is around 1,5 ms. This is a reasonable transmission speed and all the necessary data messages can be transfered in time before new ones are being transmitted. Further analysis of the outlier occurrence and a possible termination of these events need to be done in order to guarantee a delay free transmission at all time.

6.3 State detection analysis

While Section 6.1 and Section 6.2 deal with hardware and communication type analysis, this section provides information about the quality of the state design. The available states, which are defined in Section 5.1, are tested in multiple test cycles with the KIA Optima test-vehicle (see Section 4.3.2). The states were designed in such a way that each point in time refers to a specific state. Because the test-vehicle has internal states which are calculated by the hybrid control unit (HCU), we are able to analyze the correlation between the HCU states and the self-defined ones. However, we need to keep in mind that the HCU determines the states based on much more signals and components because these are specialized for this vehicle. Additionally, there are more HCU states than self-defined states so a correct assignment has to be defined. The HCU has the following internal states for the vehicle:

- Off
- Brake
- Boost
- Launch
- Idle
- Drive
- Coastdown
- Shutdown

However, there is a difference in some of those states if we compare them with the application states. HCU state *Brake* only applies when energy is recuperated through the braking process which is contradicting to the application state. Additionally, the brake pedal position trigger threshold seems to be on a higher value. The HCU state *Launch* is essentially a boost function to support the engine in lower revolutions per minute during a motor start process. Therefore, this state can be regarded as a *Boost* state. The HCU state *Idle* occurs when no gear is engaged and the engine is not turned off. The state *Drive* describes normal driving without any other function being active at the same time. *Shutdown* is active before the state *Off* is reached, which is during an engine shutdown process.
Figure 6.3 provides a detailed connection overview of the states. On the left side the HCU states are listed, whereas the application states are depicted on the right side. A green link represents an exact match and a black link indicates a similar match under the right circumstances.

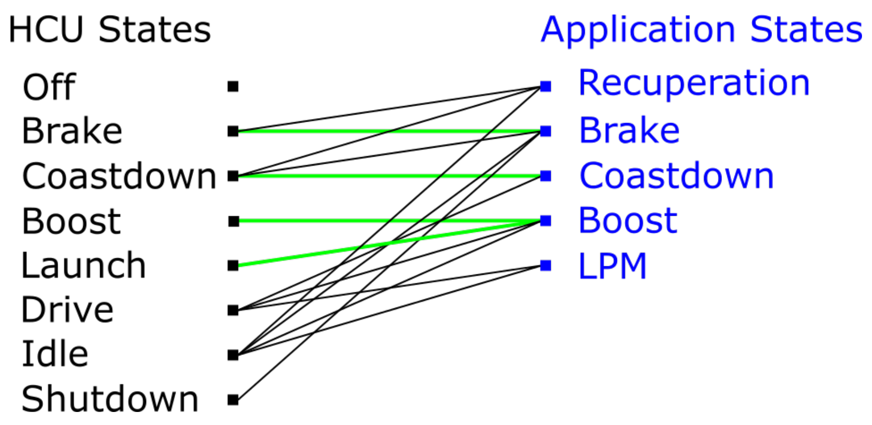


Figure 6.3: Relationship between the internal HCU states and the self- defined application states. A green link represents an exact match and a black link a similar match under certain circumstances.

For example, if an application state *Brake* is present, the HCU state needs to be *Brake* as well for a direct match of states but can also be *Coastdown, Idle* and *Shutdown* for a similar match, but only when the signal values also meet the required restrictions. An application *Brake* state can also be a HCU *Coastdown* state if the brake pedal position is beyond the threshold which was defined in Section 5.1. There are exceptions for each HCU state which make some of them appear as a similar match to an application state, because the basis on which the HCU decides the internal states is not known. However, by evaluating several test cycles and the connection of similar states when a certain application state is present could be determined and is shown in Figure 6.3. The quality of matches in all driven test cycles is listed in Table 6.2. The application state *Recuperation* has no exact match on the HCU side, but around $99,9$ % similar hits with the HCU state *Idle*. The misses are assigned to HCU state *Launch* which is most likely a delay issue since the states are contradicting. Following the evaluation of state *Brake* it can be seen that the exact match percentage of around 38 % is not as high as assumed beforehand. However, the measurement data show that a HCU *Brake* state is only applied when the brake pedal position is higher than approximately five percent, otherwise it is counted as a *Coastdown* state. Since the HCU state *Brake* only applies when energy is recuperated, the state *Drive* is chosen when no energy can be gained while driving. A combination of exact and similar matches for the *Brake* state results in approximately $98,5$ %. The remaining misses are assigned to the state *Off*, most likely because of the vehicle speed threshold. The application states *Coastdown* and *Boost* have a large number of exact matches with around 71 % and 75 %, respectively. Similar matches are accountable to the HCU states *Drive* and *Idle*. The misses are provided by the HCU state *Off* where the vehicle speed threshold (defined in the application states) is once again the error factor. It has to be set to a higher value in order to get better matching results and to filter out the HCU states *Off* and *Idle*. Nevertheless, the states *Coastdown* and *Boost* have matching percentages of around $99,8$ % each. The last application state *LPM* has no direct match, therefore the HCU states *Drive* and *Idle* are a valid choice for a similar hit if they recuperate energy at that point in time. HCU states like *Boost* or *Launch* however, would result into a miss. The match percentage of approximately $99,9$ % illustrates the quality of the definition as well, while the misses refer to the HCU state *Boost*, most likely because the power threshold for a detection is not set to the optimal value.

	Recuperation	Brake	Coastdown	Boost	LPM
Exact Match [%]	0	37,84228761	70,98459801	75,17307737	0
Similar Match [%]	99,90623352	60,66408626	28,83575485	24,80892407	99,96773504
Miss [%]	0,093766482	1,49362613	0,17964714	0,01799856	0,032264965

Table 6.2: Matching percentages for each application state.

All in all, the application states are well defined and multiple test drives with the vehicle support that claim as well. IF the brake pedal is pressed, a *Brake* state is provided while the vehicle is in state *Coastdown* when no is pedal pressed. Table 6.2 provides matching percentages for each state and it can be seen that the misses are exceptionally small. Nevertheless, further improvements about the calibration of the threshold values have to be done. In order to achieve that the tests have to be realized on multiple vehicles.

6.4 Usage

This chapter gives a short layout overview and a usage guide of the final implementation. If the application is started, the main view is shown on the display (see Figure 6.4).

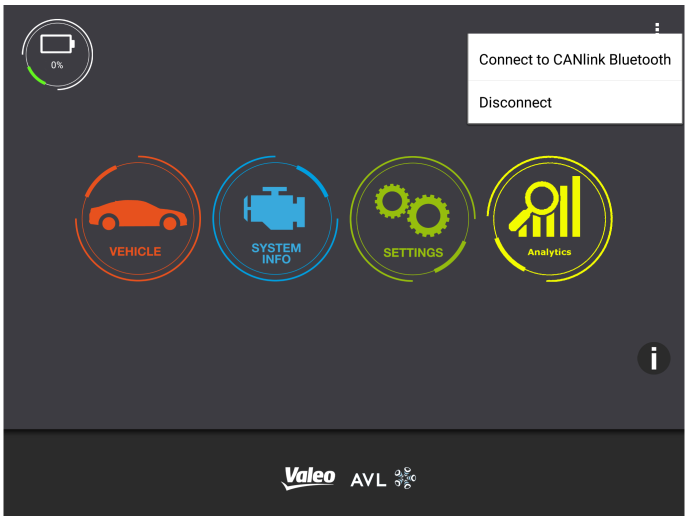


Figure 6.4: Start view of the Application. We can navigate into other activities like the cars energy flow, a settings menu, a value diagnose view and the analysis functionality.

The user has the option to go into another activity or to connect the application to a Bluetooth device. Without a Bluetooth connection the graphical representation will show no values or measurements, because no values are recorded. Once a connection is established the whole functionality can be used. A touch event on the *Analytics* button triggers an invocation of the underlying activity and its layout is being created as shown in Figure 6.5. There are multiple options which are possible from here on. The *Recuperation* button leads to functionalities with respect to gain energy through braking and coastdown. Whereas the *Performance* button directs to an exact opposite scenario, because it depicts the usage of energy while using the generator motor to support the combustion engine. The *Energy & State Analysis* button displays a view which contains the graphical representation of the energy usage over time in each considered state, as well as its distribution in total.

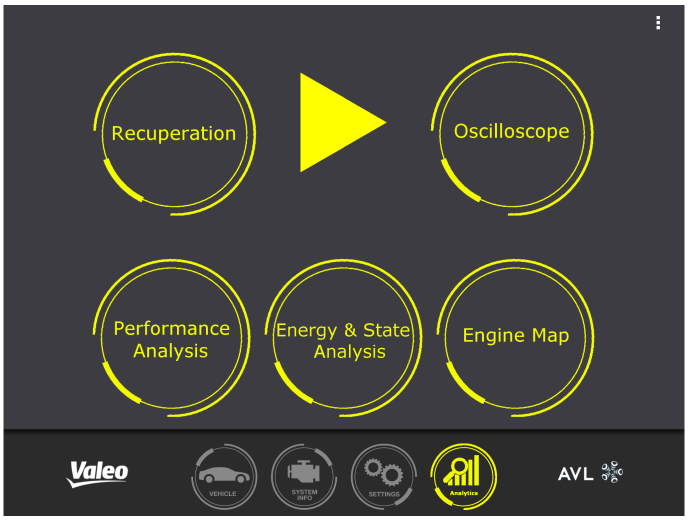
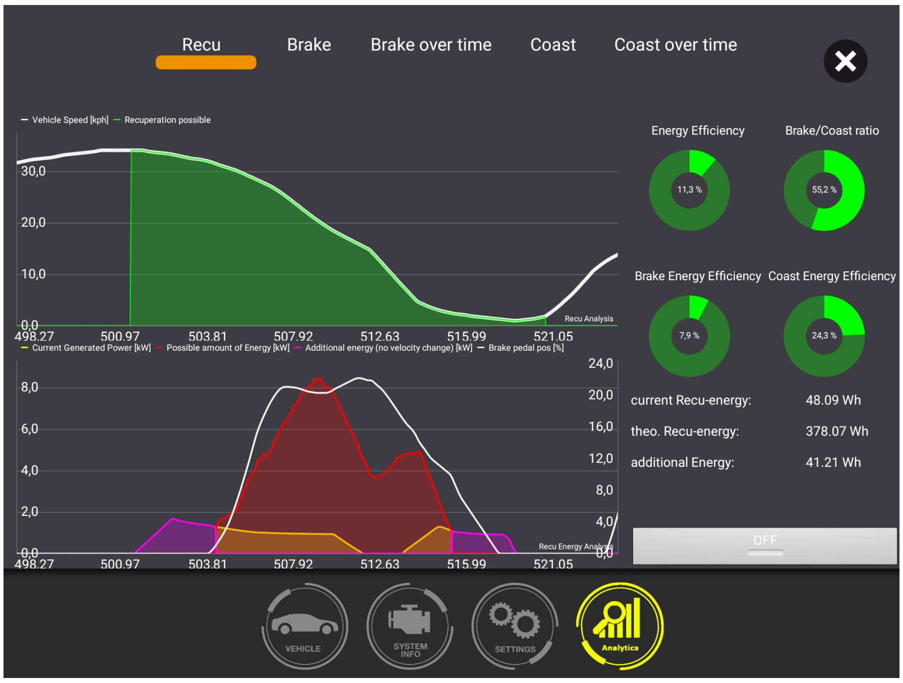


Figure 6.5: Overview of the analysis functionality.

A touch event on *Recuperation* loads a set of selectable sub-layouts with different functionality. The main layout shows an overview of the current driving cycle and is depicted in Figure 6.6. The graph in the upper half of the screen indicates when it is physically possible to recuperate energy (green area) while the white curve indicates the velocity over time. In this case recuperation can either be a brake or a coastdown event. They are not distinguishable by color in this view.

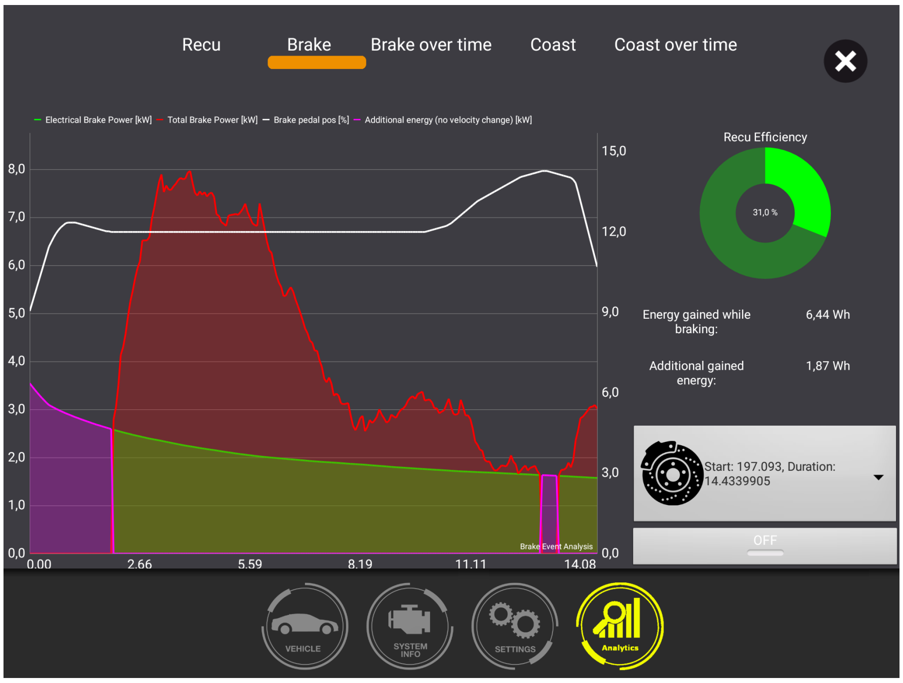


current Recu-energy: 48.09 Wh
theo. Recu-energy: 378.07 Wh
additional Energy: 41.21 Wh

Figure 6.6: Recuperation layout with a previous recorded dataset.

The bottom graph shows the corresponding energy behavior if recuperation is possible.

The red area depicts the computed maximum power the vehicle could gain. The yellow shape defines the amount of power which can be recuperated in this moment while the magenta colored curve is counted as additional power, because the theoretical power is less than the current power. This occurs because the vehicles surroundings, like hill slope, are not considered while computing. For the computation of the theoretical power we basically assume that the car is driving on a flat route without a slope. If there is a slight slope in either direction the value is still acceptable but not entirely correct, because the change in potential energy is not considered. The energy efficiency in total as well as separated in brake and coast efficiency, is displayed on the top right of the layout. Below, the numerical values to the corresponding curve over time are listed. The *ON/OFF* button indicates whether a measurement experiment is ongoing at the moment or if the application is working in the offline mode.
A touch event on the text *Brake* or a swipe event in the top half outside the graph switches this view to the desired layout which is shown in Figure (6.7). In this view it is possible to analyze each brake event separately. The graph illustrates the power values for this event over time with additional information on how the brake pedal is positioned. A gauge on the upper right side of the layout shows the recuperation efficiency which can also be expressed as electrical braking power relative to theoretical braking power (see Section 5.10). Respective numerical values of the harvested energy are published on the bottom right side. With the drop-down menu another brake event can be selected and the layout will be updated accordingly.



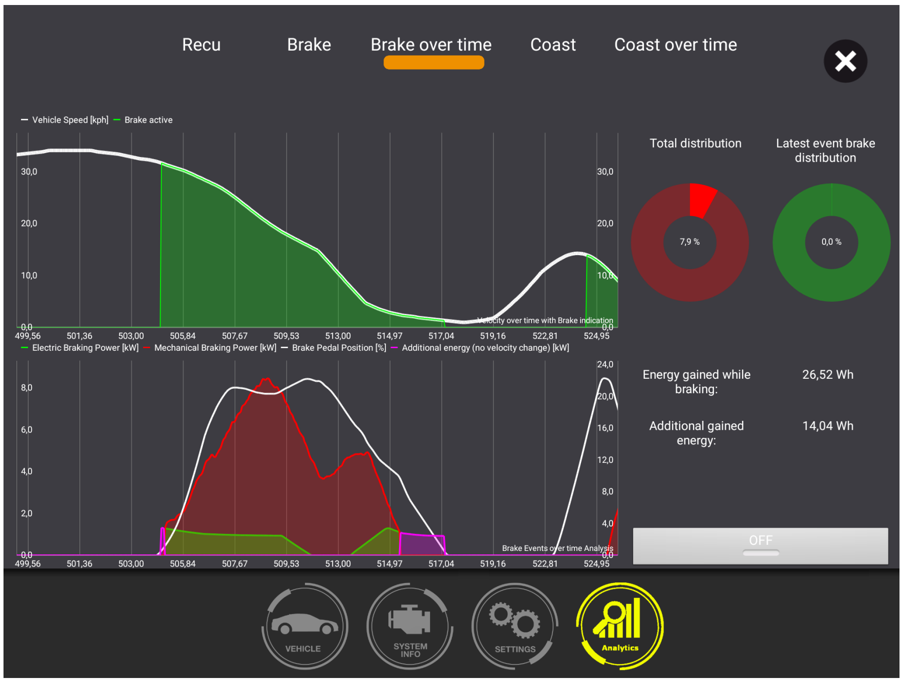
Energy gained while braking: 6,44 Wh

Additional gained energy: 1,87 Wh

Start: 197.093, Duration: 14.4339905

Figure 6.7: Representation of the energy distribution of a single brake event.

Additionally, it is possible to see all brake events over time as shown in Figure 6.8. This view is similar to Figure 6.4, but the bottom chart only shows a visualization of power values if the car is braking. The green area in the top graph indicates that the state *Brake* is active. A red gauge shows the total energy distribution over all brake events and the green one just for the last occurring event. The same functionality can be applied to the state *Coastdown*. It is possible to analyze each coast event separately or as a whole.



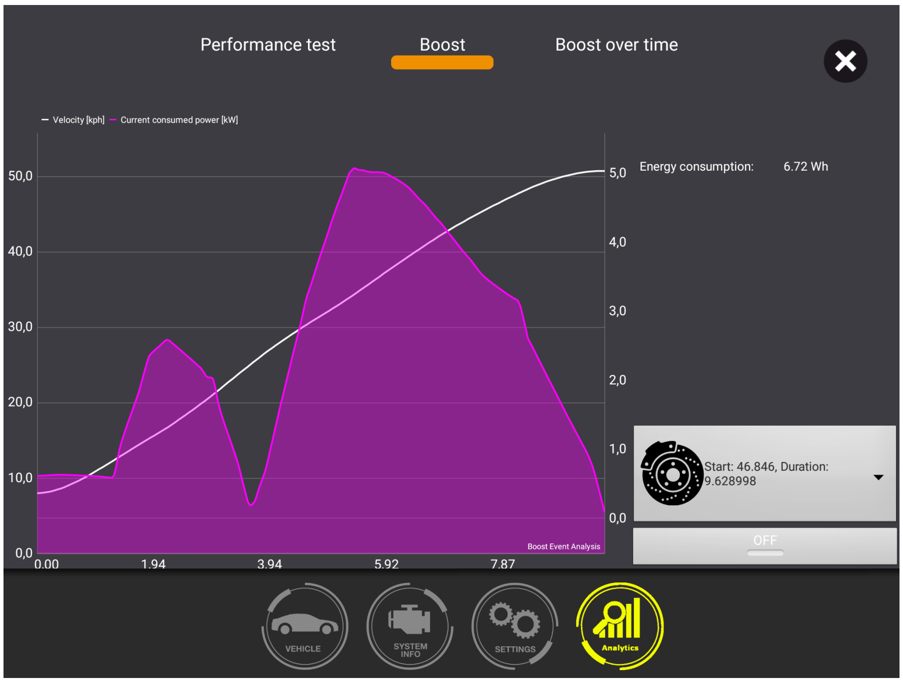
Energy gained while braking: 26,52 Wh

Additional gained energy: 14,04 Wh

Figure 6.8: Representation of the energy behavior of all braking events over time.

While recuperation deals with the vehicles energy gains in multiple states, the *Performance* section analyses the outgoing power consumption when using functions to support the internal combustion engine. It is separated in three different view elements. Firstly, the driver can use the *Performance test* to actively measure how fast he is accelerating from a certain start to end velocity. The velocity can either be one of the possible selections or a manually given one. Once the start velocity is being exceeded, the measurement starts and only ends if the end velocity is reached. While measuring, the power levels from both the electric machine and the over all system are displayed. The power output of the electric motor indicates how much it is supporting the acceleration process, whereas the system power shows how much power needs to be expended to achieve the current propulsion. This includes the change of roll- and air-resistances. Similar to the other layout it is possible to load previously recorded performance tests.
To get further details about the boosting functionality, the user has to navigate to the *Boost* view (Figure 6.9) which displays the power behavior while a boost event is active with regard to the current velocity. It is only possible to view one event at a time. Once

a new event starts, the view is cleared and the animation begins anew. A selection option is also available in this view and will be displayed once the measurement has ended. The following selection of one boost list event will render the specific event onto the graphical area. In the top right corner, the expended amount of energy is displayed for the chosen event.

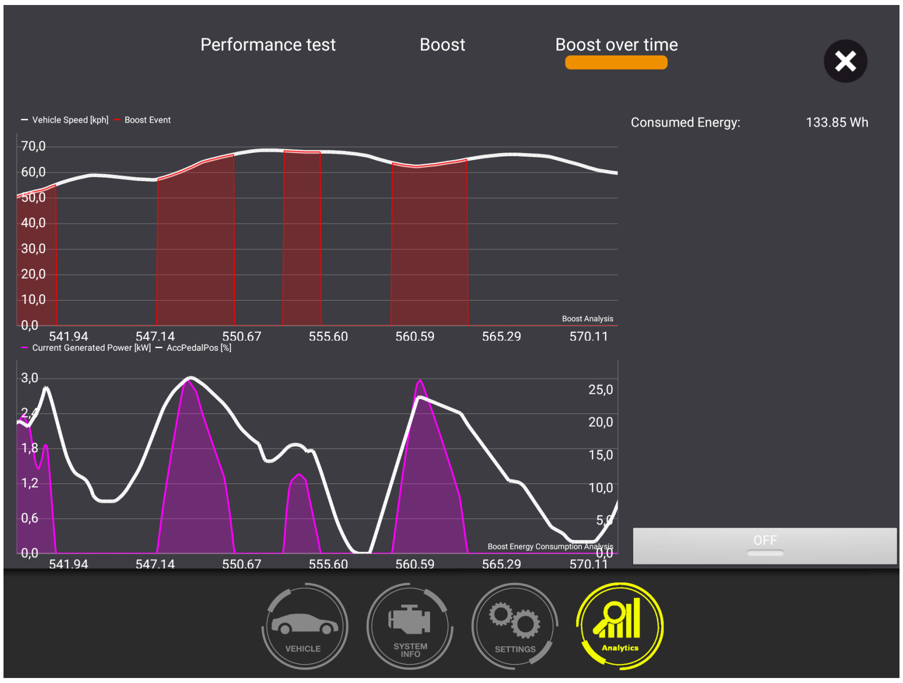


Energy consumption: 6.72 Wh

Start: 46.846, Duration: 9.628998

Figure 6.9: Representation of the energy behavior of a single boost event over time.

Another feasible analysis option is to look at the boost events over time. The application activity view for this is similar to *Coast over time* or *Brake over time*, whereas the boost state is examined in this case. The top chart in Figure 6.10 displays the velocity values over time and indicates when a boost event is happening by drawing a red area under the velocity curve. The bottom chart shows the power consumption with regard to the accelerator pedal position. In this particular view the consumed energy is calculated and shown in the top right corner.
A summary of the vehicles energy behavior is illustrated in figure 6.11. The top chart shows the vehicles state properties for each point in time, while the bottom chart displays a minutely measured energy distribution over the driving cycle. Each bar depicts the energy values for every hybrid function with regard to energy loss and gain. Every minute, the application computes a summary and displays the results on top of the respective bar. The pie chart on the right side of the layout indicates the total time distribution of hybrid function usage.



Consumed Energy: 133.85 Wh

Figure 6.10: Representation of the energy behavior of all occurring boost events over time.

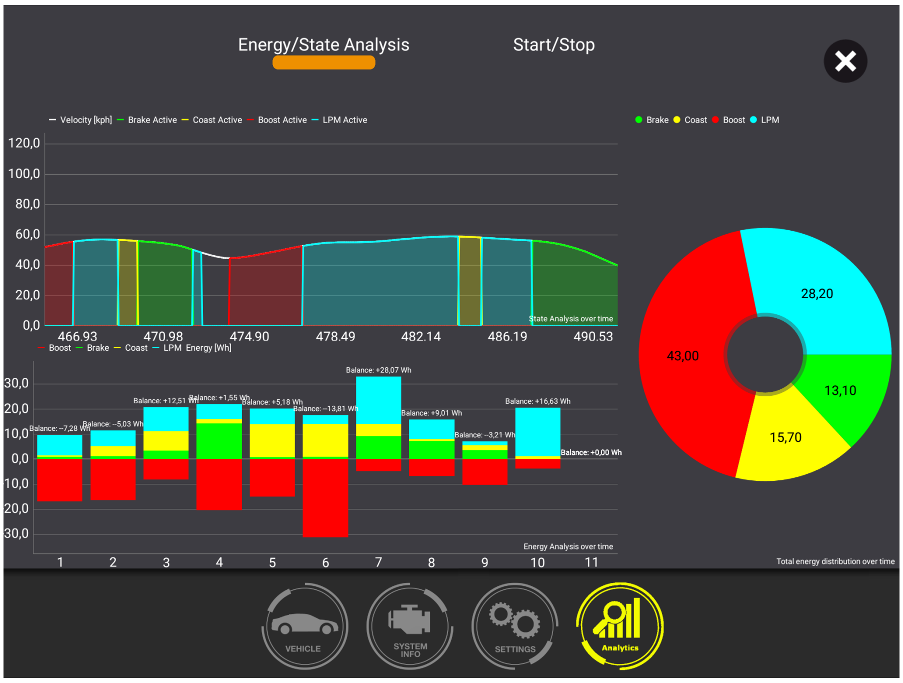


Figure 6.11: Representation of the energy behavior of all states and a distribution over time.

Additionally, the application is able to display data from a specific signal in the oscillator view. Figure 6.12 displays a signal with a constant value of zero. By pressing the option symbol in the top right corner it is possible to change the current viewed signal which refreshes the chart. The text field on the top shows the latest value from the chosen signal, while the start/stop button at the bottom can be used to freeze or unfreeze the chart updates.

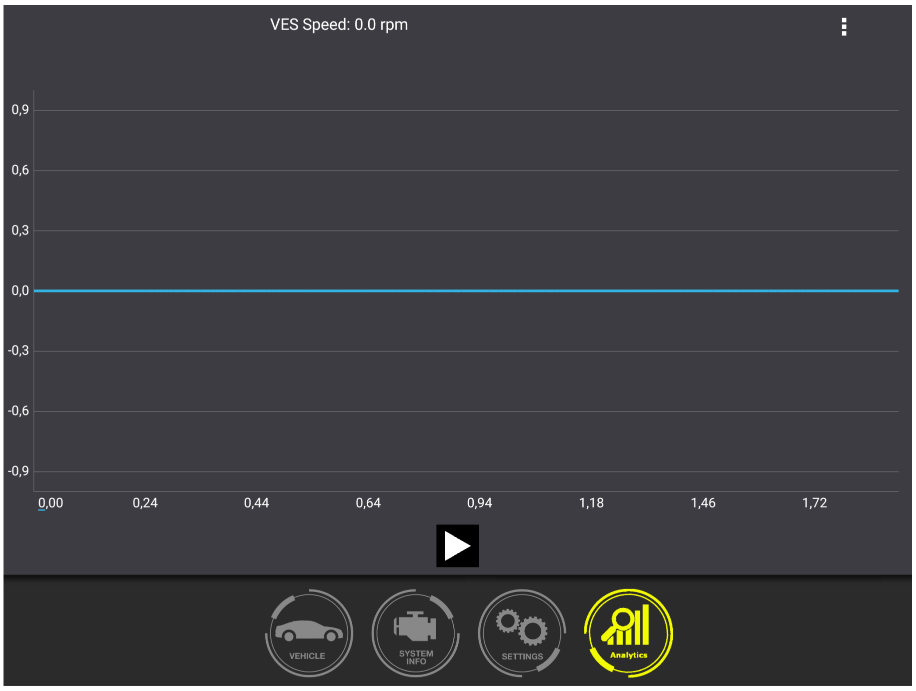


Figure 6.12: The oscillator view shows a chosen signal on the graph over time.

Figure 6.13 shows the engine map view which can present ICE load points and visualize the process of load-point moving. The underlying brake specific fuel consumption map (BSFC) indicates in which regions the ICE is working efficiently. In the green areas the loss of energy is small, whereas the efficiency in the red area is not good. Depending on the engine speed and the current torque given by the ICE, the load-point can be determined. It is desirable to always operate in a green area to maximize the vehicles energy efficiency. In order to achieve that load-point moving is used in hybrid architecture to get the optimum energy result. In this view, the driver can monitor the load-point strategy and see changes to it in an instant.

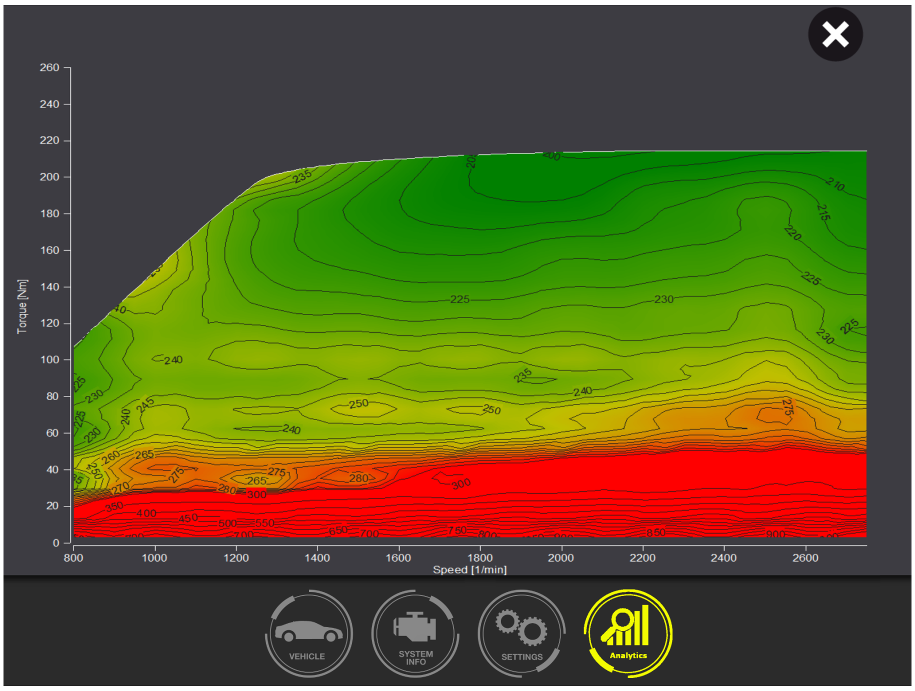


Figure 6.13: Representation of the engine map which illustrates the engines current working point. A green area is an optimal driving point but does not mean low fuel consumption per se. It just indicates that the engine is working in an optimal combustion mode.

Additional features are possible with the developed options menu. It provides functionalities which enables the user to store, load or export the measurement data as well as to open a parameter view. This view contains all the calibration values which are vehicle specific and should be specified before the test cycle begins. In case the user does not specify the information, the application uses certain standard values.

Chapter 7

Conclusion

The main goal of this thesis was to design and implement a solution to acquire telematic vehicle data and to transform the data to represent and analyze the energy behavior of hybrid vehicles. In order to achieve that, a Bluetooth connection was realized between the vehicle and the mobile device on which the graphical representation should be illustrated. CAN messages were transmitted and transformed into usable values for further computation and state matching. The states in which a hybrid vehicle is able to operate were designed and defined having all hybrid architectures in mind. For the selection of suitable CAN messages and signals for the computation, a Java tool was designed and developed. This tool is able to generate Java class files which can easily be integrated into the existing source code of the application, making it possible to be independent of the vehicle model or CAN specification. The computation of these states for each point in time as well as calculations regarding the energy behavior of the vehicle were achieved online on the mobile device. Suitable graphical representations as well as storage mechanism were designed and implemented in order to fully capture the analysis feature for each hybrid state. All operations are preformed in real-time to get the best possible result for an analysis.
The results of this thesis indicate that the desired goal was completely achieved, albeit with certain limitations. A Bluetooth device was chosen in order to manage the communication between the vehicle CAN bus and the mobile device. While the connection is working and data can be transmitted and received, the latency - even if it is sufficient for the current use-case - limits the developer on how much signals can be transmitted for a delay free representation.
Since the CAN signals are not standardized - meaning they are not in the same message and have not the same name and bit position on the CAN bus - and are dependent of the hybrid vehicle structure, the developed Java tool will serve as a base for the selection process. The visualization of CAN messages and signals makes it easy to choose the ones needed for the computation. The exported Java class files just need to be added to the applications source code files. This process works flawlessly, however, a limitation is the

necessity to build the application for each vehicle. Nevertheless, with this approach it is possible to generate a specific application with just a few clicks instead of a slow implementation process for each new vehicle.

The most important limitation lies in the fact that the application is only able to correctly calculate the theoretical power - due to the lack of proper sensor devices - when the vehicle is driving on a flat route without slope. If the road has any kind of slope, the theoretical value is slightly error prone.

This thesis provides a possibility to save all measurement data on the mobile storage device. Each time new measurement data points are incoming they are persisted in the random-access-memory (RAM) of the mobile device. In Androids latest operating system *Marshmallow* an application has up to 512 Megabytes of RAM space. Since the graphical representation and the computation consumes RAM as well, there is a limitation on how long the measurement can be performed until the application begins to stutter because it has to constantly reuse the memory.

As one of the additional features a post-processing algorithm, which polishes the data and presents it after the measurement was loaded or has ended, was developed. Furthermore, two new analysis views have been implemented to enable the user to get further information about the vehicle. Firstly, a brake specific fuel consumption (BSFC) map analysis was added to be able to visualize the current load point the internal combustion engine (ICE) is operating in. Secondly, an oscillator layout was realized which makes it possible to visualize a single signal over time.

This is the first study of a real-time investigation of hybrid vehicles energy behavior in certain driving situations with a mobile device serving as a monitor. Although the conducted tests are based on a small sample of test vehicles, the findings suggest that the chosen design and implementation are sound and applicable, which means that an energy behavior analysis is possible in real-time.

Chapter 8

Future work

This thesis was undertaken to design a way to acquire vehicle specific telematic data and to transform the data for further usage in order to get a glimpse on the energy efficiency. The results of this thesis indicate that the first steps towards an energy analysis for hybrid vehicles are achievable on a mobile android device. However, there are several topics arising from this work which should be pursued for future improvements.
Firstly, the thesis was realized with a Bluetooth connection as the main communication protocol. Additionally, a completed Bluetooth device was used to provide the CAN bus communication. For future use-cases and to reduce the material costs and hardware dimensions, it may be good to design and manufacture an own communication device as well as investigate its transmission latency. In order to reduce the battery consumption of the mobile device, a connection with Bluetooth low energy might be worth considering. Secondly, there is room for improvements in the smoothing algorithms. Only basic smoothing algorithms were implemented which run on the mobile device. If they would compute on a micro-controller chip, the mobile device could reduce the performance load which would be an obvious improvement. The smoothing algorithms could also be replaced with more complex ones if they would be computing externally. Thirdly, the computation of the theoretical energy disparity between two points in time is not optimal yet. It was assumed that the vehicle is always moving on a route without any slope. Therefore, the calculated energy is error prone in cases where the vehicle is running up- or downhill. An improvement would be the use of a gyro sensor with which it is possible to detect rotational motion as well as changes in orientation. With this additional information about the slope and the orientation of the vehicle, it would be possible to better calculate the theoretical energy. Another research area which might be of interest is the usage of Java 8, which is supported with the upcoming Android operating system "Nougat". With the help of lambda expressions and the concept of streams, a performance increase might be possible. Lastly, with the upcoming trend of integrating Android Auto into new vehicles, there is a possibility to integrate the application into that concept as well. The representation is

then done on the integrated monitor, but the system would have an in-build mechanism to receive CAN messages over cable rather than using a network communication. This would prevent the latency issues. Another positive factor would be the internal storage of the data which is not needed on the mobile device anymore.
All in all the application design and implementation is working as intended with some restrictions which were already stated. Future work can give some improvements for the benefit of the current state of the application making it an even better tool to efficiently analyze the energy behavior and other components of a hybrid vehicle.

Appendix A

Additional code

```java
private static double[] pascal(int line) {
  int lines = line;
  //Initialising the first line of the pascal triangle
  BigInteger[] line_before = new BigInteger[2];
  line_before[0] = BigInteger.valueOf(1);
  line_before[1] = BigInteger.valueOf(1);
  //Loop over all lines starting from the second one
  for (int i = 2; i < lines; ++i) {
    BigInteger[] line_current = new BigInteger[i + 1];
    //The left side and right side of the triangle is always one
    line_current[0] = BigInteger.valueOf(1);
    line_current[line_current.length - 1] = BigInteger.valueOf(1);
    //Iterate over all elements in the previous line and sum up neighbours
    for (int j = 1; j < (line_current.length - 1); ++j) {
        line_current[j] = BigInteger.ZERO;
      line_current[j] = line_current[j].add(line_before[j - 1]).add(
      line_before[j]);
    }
    //Set the current line as the old line for the next loop repetition
    line_before = line_current;
  }
  //Calculation of the weights, sum of each element has to equal one
  double[] result = new double[line_before.length];
  for (int i = 0; i < result.length; i++) {
    result[i] = line_before[i].doubleValue() / Math.pow(2, line - 1);
  }
  return result;
}
```

Listing A.1: Algorithm implementation to get the weighting values for the binominal smoothing algorithm. Computes the values with the help of the *Pascal's triangle* formulation. The summation of all weights have to be one.

```java
@Override
protected Boolean doInBackground(Void... params) {
  try {
    //Get the list where all states are persisted
    List<Float> dataState = DataObject.getInstance().getStates();
    StateEvent event;
    int size = dataState.size();

    //Loop over all state elements
    for (int i = 0; i < size-1; i++) {
      //State i
      float s = dataState.get(i);
      //State i+1
      float s1 = dataState.get(i + 1);
      //If states are the same, we are in the same event
      if (s == s1) {
        if (start) {
          startTimeIndex = i;
          start = false;
        } else {
        endTimeIndex = i;
        }
      //Once the state i is not equal with state i+1 the event ended
      } else {
        endTimeIndex = i;
        start = true;
        //Creating a corresponding event object regarding the state
        if (s == 0.0f) {
          event = new NotMatched(startTimeIndex, endTimeIndex);
          noneEvents.add(event);
        } else if (s == 1.0f) {
          event = new Recu(startTimeIndex, endTimeIndex);
          recuEvents.add(event);
        } else if (s == 2.0f) {
          event = new Brake(startTimeIndex, endTimeIndex);
          brakeEvents.add(event);
        } else if (s == 3.0f) {
          event = new Coast(startTimeIndex, endTimeIndex);
          coastEvents.add(event);
        } else if (s == 4.0f) {
          event = new Boost(startTimeIndex, endTimeIndex);
          boostEvents.add(event);
        } else if (s == 5.0f) {
          event = new LPM(startTimeIndex, endTimeIndex);
          lpmEvents.add(event);
        }
      }
    }
    //Populate the DataAnalysisObject with all the computed events
    DataAnalysisObject dataStateObject = DataAnalysisObject.getInstance();
    dataStateObject.setNoneEventList(noneEvents);
    dataStateObject.setRecuEventList(recuEvents);
    dataStateObject.setBrakeEventList(brakeEvents);
    dataStateObject.setCoastEventList(coastEvents);
    dataStateObject.setBoostEventList(boostEvents);
    dataStateObject.setLPMEventList(lpmEvents);
    return true;
  } catch (Exception e) {
    ...
    return false;
  }
}
```

Listing A.2: Post-processing algorithm which computes the start- and end-time of an event as well as stores them in the corresponding state object.

```
public void setValuesWithKey(String key, byte[] msgData) {
  //Get the data object which refers to the signal name hashtag key
  CAN_Data data = canState.get(key);
  //Get LSB
  int lsb = data.getBitPosition();
  //Compute MSB
  int msb = getMSBbitFromSignal(lsb, data.bitsize);
  int numberOfBytes;
  int byteOffsetMSB = getByteOffsetMSB(msb);
  int startByte = data.getByteOffset();
  int bitOffset = data.getBitOffset();
  int msbBitOffset = msb % 8;

  double resultValue = 0;
  //Goto case with matches the nubmer of bytes of the signal specification
  numberOfBytes = Math.abs(-1 * (startByte + 1) + (getByteOffsetMSB(msb)));
  switch (numberOfBytes) {
    case 1:
      //Function getBits() computs the value for one bit, given a mask and a bit offset
      resultValue += ByteCommand.getBits(msgData, 1, startByte,
        data.getBitMaskForLessThan8Bit(false), bitOffset+1);
      break;
    case 2:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
    case 3:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      //Remaining size is 8 bits shorter now, since we already got the value for the first byte
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
    case 4:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 2, 0xFF, 1)
      //Remaining size is 16 bits shorter now, since we already got the value for the first two
     bytes
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 16);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
```

```java
    case 5:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 2, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 16);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 3, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 24);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
    case 6:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 2, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 16);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 3, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 24);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 4, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 32);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
    case 7:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 2, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 16);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 3, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 24);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 4, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 32);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 5, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 40);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffse +1);
      break;
    case 8:
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB, getBiskMaskForMSB(msb), 1)
        * Math.pow(2, remainingBitsize(msbBitOffset, data.getBitSize()));
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 1, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 8);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 2, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 16);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 3, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 24);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 4, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 32);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 5, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 40);
      resultValue += ByteCommand.getBits(msgData, 1, byteOffsetMSB + 6, 0xFF, 1)
        * Math.pow(2, (remainingBitsize(msbBitOffset, data.getBitSize())) - 48);
      resultValue += ByteCommand.getBits(msgData,1,startByte,data.getBitMaskLSB(),bitOffset+1);
      break;
    default:
      break;
  }
  //Setting the value to the CAN_Data object
  data.setValue((int) resultValue);
}
```

Listing A.3: Transforming of the raw data field into an usable value which is then stored into the corresponding *CAN_Data* object.

List of Figures

List of listings

Bibliography

[1] Al-Ani, Tarik: *Android In-Vehicle Infotainment System (AIVI).* Master's thesis, University of Otago, Dunedin, New Zealand, 2011.

[2] Android: *Android Interfaces and Architecture.* `https://source.android.com/devices/`. [last visit 21.07.2016].

[3] Android Developers: *Handler.* `https://developer.android.com/reference/android/os/Handler.html`. [last visit 07.08.2016].

[4] Aubury, Matthew and Wayne Luk: *Binomial Filters.* Journal of VLSI Signal Processing, 12(1):35–50, 1996.

[5] Austin-Morgan, Tom: *Schaeffler's E-Clutch system can cut fuel consumption by up to 8 %.* `http://www.eurekamagazine.co.uk/design-engineering-news/schaefflers-e-clutch-system-can-cut-fuel-consumption-by-up-to-8/116864/`. [last visit 12.08.2016].

[6] Bagschik, Peter: *An Introduction to CAN.* `http://dmi.uib.es/~jantich/Documentos/CAN.pdf`, 2000.

[7] Bayrak, Alparslan: *Topology Considerations in Hybrid Eletric Vehicle Powertrain Architecture Design.* PhD thesis, University of Michigan, USA, 2015.

[8] Beckhoff Automation GmbH & Co. KG: *CANopen Verkabelung.* `http://infosys.beckhoff.com/index.php?content=../content/1031/bk51x0/html/co_inswirbus.htm&id=`. [last visit 21.07.2016].

[9] Berman, Brad: *History of Hybrid Vehicles.* `http://www.hybridcars.com/history-of-hybrid-vehicles/`, 2011.

[10] Bluetooth SIG, Inc.: *Bluetooth core specification.* `https://www.bluetooth.com/specifications/bluetooth-core-specification`. [last visit 20.07.2016].

[11] Brahmachari, Ruprekha: *Hybrid Cars*, Presentation, 9th Indo-German Winter Academy, Pune, India, 2010.

[12] Ceřovský, Zdeněk and Pavel Mindl: *Hybrid Electric Cars, Combustion Engine Driven Cars and their Impact on Environment.* In *International Symposium on Power Electronics, Electrical Drives, Automation and Motion. SPEEDAM 2008*, pages 739–743, Ischia, Italy, 2008.

[13] CONTINENTAL AG: *Grafik electrification.* `http://www.continental-automotive.de/www/download/automotive_de_de/themes/passenger_cars/powertrain/grafik_electrification.pdf`. [last visit 11.07.2016].

[14] Corrigan, Steve: *Introduction to the Controller Area Network (CAN)*, Application Report, Texas Instruments, 2008.

[15] DOK-ING Ltd.: *MyHMI.* `http://www.dok-ing.hr/solutions/myhmi`. [last visit 03.08.2016].

[16] DPA: *Motor ausschalten bei Rot - Kurzer Dreh hilft Spritsparen.* `http://www.n24.de/n24/Nachrichten/Auto-Verkehr/d/496454/kurzer-dreh-hilft-spritsparen.html`, 2008.

[17] Ehsani, Mehrdad, Yimin Gao and Ali Emadi: *Modern Electric, Hybrid Electric, and Fuel Cell Vehicles - Fundamentals, Theory, and Design.* CRC Press, Boca Raton, Florida, USA, 2nd edition, 2009.

[18] Ehsani, Mehrdad, Yimin Gao and John M. Miller: *Hybrid Electric Vehicles: Architecture and Motor Drives.* Proceedings of the IEEE, 95(4):719–728, 2007.

[19] Fischer, Olga, Oskar Kaplun, Thilo Schumann, Holger Zeltwanger and Reiner Zitzmann: *CAN dictionary. Keywords, Technical terms, Standards.* Nuremberg, Germany, 8th edition, 2015.

[20] Göhner, Peter: *CAN-Bus*, Unterlagen zur Vorlesung Automatisierungstechnik I, Universität Stuttgart, Germany, 2014.

[21] Google Inc.: *Android Auto.* `https://developer.android.com/auto/index.html`. [last visit 12.08.2016].

[22] Görke, Daniel: *Grundlagen der Hybridfahrzeuge.* In *Untersuchungen zur kraftstoffoptimalen Betriebsweise von Parallelhybridfahrzeugen und darauf basierende Auslegung regelbasierter Betriebsstrategien*, pages 5–13. Springer Fachmedien Wiesbaden, 2016.

[23] greenrobot: *greenrobot EventBus Documentation.* `http://greenrobot.org/eventbus/documentation/`. [last visit 29.07.2016].

[24] HACKMANN, W.: *Electrical Traction Drives: HEV E-Drive Development and Products*, Presentation Continental AG, Gifhorn, 29.11.2012.

[25] HAWKINS, IAN: *Torque.* `https://torque-bhp.com/wiki/Main_Page`. [last visit 03.08.2016].

[26] HOFMAN, THEO and ROELL VAN DRUTEN: *Energy Analysis of Hybrid Vehicle Powertrains.* IEEE Vehicle Power and Propulsion, 2004.

[27] HTC CORPORATION: *Google Nexus 9 - Technische Daten.* `http://www.htc.com/de/tablets/nexus-9/`. [last visit 05.08.2016].

[28] INFINION TECHNOLOGIES INC.: *CAN - Controller Area Network Tutorial.* `http://microcontroller.com/learn-embedded/CAN1_sie/CAN1big.htm`, 1999.

[29] JAHODA, PHILIPP: *MPAndroidChart.* `https://github.com/PhilJay/MPAndroidChart`. [last visit 22.07.2016].

[30] JUNGINGER, MARKUS: *greenrobot EventBus.* `https://github.com/greenrobot/EventBus`. [last visit 29.07.2016].

[31] MAYER, EUGEN: *Serielle Bussysteme im Automobil - Teil 1:Architektur, Aufgaben und Vorteile.* Elektronik automotive, 7:70–73, 2006.

[32] NAU, ROBERT: *Forecasting with moving averages.* `http://people.duke.edu/~rnau/Notes_on_forecasting_with_moving_averages--Robert_Nau.pdf`, Fuqua School of Business, Duke University, North Carolina, USA, 2014.

[33] OSCH, M. J. P. VAN and SCOTT A. SMOLKA: *Finite-State Analysis of the CAN Bus Protocol.* In *The 6th IEEE International Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking*, pages 42–52, Washington, DC, USA, 2001.

[34] PAWLAN, MONICA: *Reference Objects and Garbage Collection.* `http://www.pawlan.com/monica/articles/refobjs/`. [last visit 04.08.2016].

[35] PREMKUMAR, SAMUEL: *The Future Of Automobile.* `http://www.slideshare.net/spkingsley/the-future-of-automobile`, 2010.

[36] RM MICHAELIDES SOFTWARE & ELEKTRONIK GMBH: *Hardware Manual - RM CANlink Bluetooth 2xxx*, Fulda, Germany, 2006.

[37] ROBERT BOSCH GMBH: *CAN Specification Version 2.0*, Stuttgart, Germany, 1991.

[38] SAMSUNG ELECTRONICS: *Benutzerhandbuch SM-T710 SM-T810*, version 1.0, 2016.

[39] SPELTA, CRISTIANO, VINCENZO MANZONI, ANDREA CORTI, ANDREA GOGGI and SERGIO MATTEO SAVARESI: *Smartphone-Based Vehicle-to-Driver/Environment Interaction System for Motorcycles.* IEEE Embedded Systems Letters, 2(2):39–42, 2010.

[40] SPONÅS, JON GUNNAR: *Things you should know about Bluetooth range.* `http://blog.nordicsemi.com/getconnected/things-you-should-know-about-bluetooth-range`. [last visit 20.07.2016].

[41] TAHAT, ASHRAF, AHMAD SAID, FOUAD JAOUNI and WALEED QADAMANI: *Android-Based Universal Vehicle Diagnostic and Tracking System.* In *2012 IEEE 16th International Symposium on Consumer Electronics (ISCE)*, pages 137–143, 2012.

[42] TECHOTOPIA: *An Overview of the Android Architecture.* `http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture`. [last visit 21.07.2016].

[43] US ENVIRONMENTAL PROTECTION AGENCY: *Sources of Greenhouse Gas Emissions.* `https://www.epa.gov/ghgemissions/sources-greenhouse-gas-emissions`. [last visit 08.07.2016].

[44] VARGAS, OCTAVIO: *eMobility – Passing the baton from fossils to electrons*, Presentation, Schaeffler Symposium 2014.

[45] YI, WON-JAE, WEIDI JIA and JAFAR SANIIE: *Mobile Sensor Data Collector using Android Smartphone.* In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2012.

[46] ZALDIVAR, JORGE, CARLOS T. CALAFATE, JUAN CARLOS CANO and PIETRO MANZONI: *Providing Accident Detection in Vehicular Networks Through OBD-II Devices and Android-Based Smartphones.* In *2011 IEEE 36th Conference on Local Computer Networks*, 2011.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den August 16, 2016

Thomas Richter