Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Conceptualization and Implementation of a Digital Twin for Autonomous Driving

Yannick Liszkowski

Course of Study: Computer Science

Examiner: Prof. Dr.-Ing. Steffen Becker

Supervisor: Patrick Spaney, M.Sc.

Commenced: May 27, 2024

Completed: November 27, 2024

Abstract

This thesis explores the innovative field of Digital Twin (DT) technology, with a particular focus on its applications in Autonomous Driving. A DT is a digital or virtual representation of a physical entity, product, system, process or asset. Data flows between the physical and digital objects are fully integrated in both directions. DTs use techniques such as modelling, shadowing, simulation, Machine Learning and interaction to support in designing, prototyping, monitoring, controlling, decision making and optimizing Cyber-Physical Systems. Engineering and implementing DTs is a time-consuming and complex process. Currently, DT design primarily focuses on a specific application domain, with an abstract perspective. The structure of DTs should integrate more flexibility, composability and extendability with regard to the development and implementation of DTs, regardless of the area of application. This thesis presents a conceptual meta-model for such a DT, based on the findings of research, comprehensive review, systematic categorization and abstraction of previous work in the field. Furthermore, a use case and its architectural framework are presented as well as a prototype DT implementation based on an existing physical entity, i.e. a Physical Twin (Arduino robot cars). A scenario-based evaluation of the architecture for the implementation of the prototype DT for Autonomous Driving was conducted. All the evaluated direct and indirect architecture-scenarios for DT extendability are very well supported. The thesis concludes with suggestions for avenues of future research that could be pursued.

Kurzfassung

Die vorliegende Arbeit widmet sich der Untersuchung des innovativen Feldes der Digitalen Zwilling bzw. Digital Twin (DT)-Technologie. Dabei wird ein besonderer Fokus auf die Anwendungen der DT-Technologie im Kontext des autonomen Fahrens gelegt. Ein DT stellt die digitale bzw. virtuelle Repräsentation eines physischen Objekts, Produkts, Systems, Prozesses oder Assets dar. Der Datenfluss zwischen den physischen und dem digitalen Objekten ist vollständig integriert, in beide Richtungen. DTs nutzen eine Vielzahl von Techniken, darunter Modellierung, Shadowing, Simulation, maschinelles Lernen und Interaktion. Damit unterstützen sie eine Vielzahl von Prozessen, darunter die Gestaltung, Prototypenerstellung, Überwachung, Steuerung, Entscheidungsfindung und Optimierung cyber-physischer Systeme. Die Entwicklung und Implementierung von DTs ist ein zeitintensiver und komplexer Prozess. Derzeit liegt der Fokus beim Design von DTs vorwiegend auf einem bestimmten Anwendungsbereich mit einer abstrakten Perspektive. Die Struktur von DTs sollte mehr Flexibilität, Zusammensetzbarkeit und Erweiterbarkeit im Hinblick auf die Entwicklung und Implementierung von DTs integrieren, und dies unabhängig vom Anwendungsbereich. Diese Arbeit präsentiert ein konzeptionelles Metamodell für ein solches DT, das auf den Ergebnissen der Forschung, einer umfassenden Überprüfung, einer systematischen Kategorisierung und Abstraktion früherer Arbeiten auf diesem Gebiet basiert. Des Weiteren werden ein Anwendungsfall und sein Architekturrahmen sowie eine prototypische DT-Implementierung auf Basis einer vorhandenen physischen Einheit, d.h. eines physischen Zwillings (PT) mit Arduino-Roboterautos, präsentiert. Es wurde eine szenariobasierte Evaluation der Architektur für die Implementierung des Prototyps DT für autonomes Fahren durchgeführt. Alle evaluierten direkten und indirekten Architektur-Szenarien für die DT-Erweiterbarkeit werden sehr gut unterstützt. Die Arbeit schließt mit Vorschlägen für mögliche zukünftige Forschungsansätze.

Contents

1 Introduction **19**
- 1.1 Motivation . . . 19
- 1.2 Objectives . . . 21
- 1.3 Research Questions and problem-solving approach . . . 22
- 1.4 Thesis Structure . . . 23

2 Foundations **25**
- 2.1 Foundational Terms and Definitions . . . 25
- 2.2 Existing Concepts of Digital Twins . . . 28
- 2.3 Existing Digital Twin Architectures . . . 30
- 2.4 Existing Digital Twin Architectures for Autonomous Driving . . . 33
- 2.5 Existing Autonomous Driving Architectures . . . 36
- 2.6 Pertinent Foundations (to **RQ1.1**) . . . 39

3 Related Work **41**
- 3.1 Methodology . . . 41
- 3.2 Related Work for Conceptualization and Architectures of Digital Twins . . . 42
- 3.3 Related Work for Digital Twins for Autonomous Driving . . . 44
- 3.4 Related Work for Autonomous Driving / Vehicle . . . 46

4 Requirements **47**

5 Concept **51**
- 5.1 Key components and connections of a conceptual structure/architecture for a DT (answer to **RQ1.2**) . . . 52
- 5.2 Concept Elaboration . . . 53
- 5.3 Use Case Description and Architecture . . . 57

6 Implementation of a Digital Twin for Autonomous Driving **69**
- 6.1 Essential fundamentals . . . 70
- 6.2 Physical Twin . . . 72
- 6.3 Solution approach, selection of Tools and Languages . . . 77
- 6.4 Technical Realization . . . 83
- 6.5 Illustrative Scenarios with the prototype Digital Twin . . . 116

7 Evaluation **137**
- 7.1 Evaluation Design . . . 137
- 7.2 Results . . . 139
- 7.3 Discussion . . . 153
- 7.4 Threats to Validity . . . 154

8 Conclusion and Outlook **155**
8.1 Summary 155
8.2 Benefits 156
8.3 Limitations 156
8.4 Lessons Learned 157
8.5 Future Work 159

Bibliography **163**

A Supplementary Material for the Robot Cars **171**
A.1 Robot Car Hardware Specification 171
A.2 Robot Car Configuration Data 173
A.3 Communication with Robot Car 178
A.4 Supplementary Images for the Robot Cars 183

B Supplementary Material for the Software Tools **185**

C Structure of the Prototype DT-Entities Implementation coded in Java **187**

D Supplementary Information on Design Decisions for Dashboard **191**

E Summarized Considerations to RQ1.3 **193**

F Summarized Considerations to the Use Case **195**

G Supplementary Material for the Database **197**

List of Figures

5.1 Meta-model for a composable, flexible and extendable DT 56
5.2 The architecture for a DT for Autonomous Driving 57

6.1 Robot Car 73
6.2 Informal illustration for PT connected to DT 78
6.3 Informal illustration of the solution approach for DT (and connection to PT) . . . 78
6.4 Enhanced informal illustration of PT and DT with software tools 79
6.5 Architecture for the Implementation of a DT for Autonomous Driving 83
6.6 Caption of an extract view of a very small part of the Data Manager coded in Node-RED 84
6.7 Screenshot of the DistanceSensorChanges Table from the Digital Twin SQLite Database 85
6.8 Dashboard Sections Physical Twin Vehicle 1 and Digital Twin Vehicle 1 87
6.9 Dashboard Sections Log, Environment and DT Monitor 90
6.10 Handling of Messages in Digital Twin Java Applications 96
6.11 PT Vehicle Configuration Process 98
6.12 Different states of the AdaptiveCruiseControl scenario 106
6.13 The overtaking vehicle, still at a distance, catching up with the lead vehicle 109
6.14 The overtaking vehicle has caught up with the lead vehicle 110
6.15 It is recognised that the lead vehicle has moved aside 110
6.16 The overtaking vehicle did not collide (with the rear of) the lead vehicle 111
6.17 Both vehicles prevent potential collisions using "Pause On Obstacle" functionality 111
6.18 Prototype DT with 2 RCs and Dashboard 116

7.1 Bar chart for Architecture-scenario "insert DT-Entity-Scenario Overtaking" . . . 151
7.2 Bar chart for Architecture-scenario "insert DT-Entity PathPlanning" 152
7.3 Bar chart for Architecture-scenario "insert DT-Entity-Scenario SchoolZone" . . 152
7.4 Bar chart for the 3 Architecture-scenarios of the evaluation 153

A.1 Road Track with both Robot Cars 183
A.2 Sideview of Red Robot Car 184
A.3 Frontview of Red Robot Car 184

C.1 Screenshot of the Package Explorer in Eclipse of the Prototype DT 187

List of Tables

4.1 Requirements 50

5.1 Overview of the Entities with their role classification (Descriptive, Predictive, Prescriptive) according to [WEB+21]. 65
5.2 Overview of the Entities with their functionality classification (DataProcessor, Evaluator, Reasoner, Executor) according to [WBD+20]. 65
5.3 Overview of the Entities and their service classification based on [TLZ+19]. . . . 65

6.1 Functionality States of RC under different MODE Values 88
6.2 Process description for Scenario: driving ON TRACK (shadowing) 118
6.3 Process description for Scenario: driving ON TRACK + PAUSE ON OBSTACLE (shadowing) 121
6.4 Process description for Scenario: driving ON TRACK + STOP ON OBSTACLE 123
6.5 Process description for Scenario: driving ON TRACK + STOP ON VIRTUAL OBSTACLE 126
6.6 Process description for Scenario: driving ON TRACK + ADAPTIVE CRUISE CONTROL + PAUSE ON OBSTACLE for lead car 129
6.7 Process description for Scenario: driving ON TRACK + OVERTAKING + PAUSE ON OBSTACLE 133
6.8 Process description for Scenario: driving ON TRACK + THREE POINT TURN + PAUSE ON OBSTACLE 135

7.1 Overview of Architecture–Scenarios for Evaluation 1/3 141
7.2 Overview of Architecture-scenarios for Evaluation 2/3 142
7.3 Overview of Architecture-scenarios for Evaluation 3/3 143

G.1 SQLite Tables of DB_DT Database 206

List of Listings

6.1 Java code for storing new `TaskObject` instance containing `priority`, `newfutureTask` and JSON object inside `taskMap` 97
6.2 Java code to calculate vehicle speed based on given distance 107
A.1 Regular expression for *set* 179
A.2 Regular expression for a *subscription* 180
A.3 Format to acknowledge a *set* message 181
A.4 Format to acknowledge a *subscription* message 181
A.5 Format of a *get* message 181

Acronyms

ABS anti-lock braking system. 100

ACC Adaptive Cruise Control. 61

ADAS Advanced Driver Assistance System. 27, 50, 57, 72

AGV Automated Guided Vehicle. 41

AI Artificial Intelligence. 19

ALMA Architecture-Level Modifiability Analysis. 137

AMQP Advanced Message Queuing Protocol. 77

API Application Programming Interface. 44, 48, 77, 81, 93, 159

AS Actual System. 28

ASCII American Standard Code for Information Interchange. 79, 81, 178, 179, 180, 181, 182

ATAM Architecture Trade-off Analysis Method. 137

AUTOSAR AUTomotive Open System ARchitecture. 72

CAD Computer-Aided Design. 26

CAN Controller Area Network. 72

CAV Connected and Autonomous Vehicle. 46

CBR Base-Based-Reasoning. 54

CoAP Constrained Application Protocol. 43

CPPS Cyber-Physical Production System. 30, 31, 42, 53

CPS Cyber-Physical System. 3, 19, 27, 44

DC Direct Current. 73, 100, 171, 172, 176, 177

DoE Design of Experiment. 31

DS Data Structure. 19, 28, 30

DSH Dashboard. 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135

DSL Domain-Specific Language. 25

DT Digital Twin. 3, 5, 19, 26, 148, 155

E2E End-to-End. 44

ECU Electronic Control Unit. 72

EEPROM Electrically Erasable Programmable Read-Only Memory. 99, 172

EMB Electro-Mechanical Brakes. 100

FAV Functional Architecture View. 36

HAL Hardware Abstraction Library. 73

HTTP Hypertext Transfer Protocol. 53, 73, 80

IEEE Institute of Electrical and Electronics Engineers. 33

IMU Inertial Measurement Unit. 33

IoT Internet of Things. 19, 41, 43, 52, 70

IoV Internet of Vehicles. 41

ITS Intelligent Transport System. 36

JMS Java Message Service. 93

JSON JavaScript Object Notation. 70

LIDAR Light Detection and Ranging. 27, 33, 35, 46, 50, 72, 156, 161

LWT Last Will and Testament. 71

MAPE-K Monitor-Analyse-Plan-Execute over a shared Knowledge. 41, 43

MODA Models and Data. 29

MQTT Message Queuing Telemetry Transport. 22, 24, 41, 43, 53, 66, 70

NIST National Institute of Standards and Technology, USA. 46

OPC-UA OPC-Unified Architecture. 30

OTA Over-the-Air. 21, 76

PID Principal Ideal Domain. 74, 160

PT Physical Twin. 3, 5, 9, 19, 21, 26

Pub/Sub Publish/Subscribe. 27, 70

QoS Quality of Service. 71

RADAR Radio Detection and Ranging. 27, 33, 35, 50, 61, 63, 72, 156, 161

RC Robot Car. 9, 11, 88, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 131, 132, 133, 134, 135

RCS Realtime-Control-System. 46

REST Representational State Transfer. 159

RPC Remote Procedure Call. 71, 76

SAAM Software Architecture Analysis Method. 137

SAE Society of Automotive Engineers. 27

SQL Structured Query Language. 82, 85, 86

UML Unified Modeling Language. 41

UTC Universal Time Coordinated. 93

VEH Vehicle. 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 131, 132, 133, 134, 135

WLDT White Label Digital Twin. 41, 43

1 Introduction

The term Digital Twin (DT) has steadily evolved over the past few decades and has garnered significant attention due to its transformative potential across industries. A Digital Twin is a virtual representation of an object or system that usually encompasses its complete lifespan. A DT system is continuously updated with real-time data and employs a range of techniques, including simulation, context awareness, Machine Learning, deduction, extrapolation and prediction, to support evidence-based decision-making. DTs are utilized in various disciplines to aid in the engineering, monitoring, controlling, and optimization of Cyber-Physical Systems [WKM+20]. The increasing popularity of Digital Twins can be attributed to advancements in IoT, data analytics, and Artificial Intelligence (AI), which enable more accurate and efficient simulations of complex processes. Engineering DTs is often not tightly integrated with the development of the system [WBD+20].

1.1 Motivation

Digital Twins (DTs) are complex entities. To create precise digital representations of physical or conceptual entities and their properties, the designing, developing, and maintaining of DTs, often requires the integration of various disciplines in order to analyse the properties of the physical element and the associated software domain. As the complexity of the developed systems increases, misunderstandings between stakeholders from different domains complicate system development.

According to Adamenko et al. [AKP+20], there are two principal methods for designing Digital Twins: either by constructing a system model of the physical object or asset, or by creating a Data Structure that organises and links together the PT's data, parameters and other pertinent information and values, and that *"The greatest added value and functionality is achieved through a combination of both approaches"*.

This thesis aims to investigate the feasibility and modelling of DTs that allow the DTs to be composable and to enable minor changes, major modifications and also to integrate new components/elements (e.g. new sensors, etc.) and/or functionalities (e.g. other scenarios, simulations, testing, etc.), as such was not found in the reviewed architectures.

This research focuses on DTs in the application area of Autonomous Driving.

Renee Morad describes the potential quiet well in his article "How digital twins are driving the future of autonomous vehicles"[1]:

Automakers increasingly utilize Digital Twins to precisely test and measure automotive components and features, prior to their real-world deployment, fostering faster, safer, and more cost-effective innovation.

The automotive sector leverages Digital Twins for simulation, design, maintenance, repair, manufacturing, and post-accident analysis, as well as for the development of in-vehicle networks and cybersecurity systems.

With DTs they can also emulate and simulate complex scenarios, i.e. real-world driving conditions in the laboratory, thereby varying factors such as traffic density, speed, distance, and the number of targets. This utilization enables the training of Autonomous Driving algorithms effectively. This approach accelerates development timelines for both typical and edge-case scenarios, as DTs create controlled virtual environments, emulating real-world interactions and behaviours. By harnessing extensive data from self-driving car tests, automakers construct intricate simulations to analyse Artificial Intelligence responses to unpredictable conditions like weather changes or traffic jams, enabling comprehensive and rapid functional testing.

While complete emulation of real-world scenarios is unattainable, the controlled variables in Digital Twin environments enable close approximations. This transformative approach positions Digital Twins as pivotal in driving automotive innovation, opening new avenues for advancements and safety enhancements.

Renee Morad concludes with: *"By essentially bringing the road to the lab, the automotive industry is undergoing a paradigm shift for innovation – and digital twins are the quiet but powerful forces enabling these new possibilities behind the scenes"*.

A report by Future Market Insights[2] projects that the global Digital Twin technology market, currently valued at $9.5 billion, will grow at a compound annual rate of 22.6% to reach $77.65 billion by 2032. Although Digital Twins are utilized across multiple sectors, the transportation and automotive industry holds the largest market share at 15%.

[1] https://www.electronicspecifier.com/pages/es-design-august-2023

[2] https://www.fortunebusinessinsights.com/digital-twin-market-106246

1.2 Objectives

This thesis aims to investigate the creation process of a composable Digital Twin, with respect to its application domains, in order to avoid redundant developments and implementations of DTs.

The objective of this thesis is to research and create a conceptual structure/architecture for a Digital Twin, that utilizes self-adaptation, control, and model-based engineering techniques to specify the structural and behavioural aspects of DTs and enable the evolution of their internal models. This Digital Twin should be composable and extendable (for terminology see [EI90]), i.e. modelled in a way that minor changes, major modifications and also new components/elements (e.g. new sensors, etc.) and/or functionalities (e.g. other scenarios, simulations, testing, etc.) can be easily added and managed. This involves functional decompositions, separation of concerns, and the establishment and maintenance of association relations between the twins, i.e. the DT and the Physical Twin (PT), as described in [RMV+20]. Furthermore, during the conceptualization and design phase of the DT, it is important to consider potential future extensions which also prioritise exchangeability and flexibility (for terminology see [EI90]). Thus, the architecture describes a general DT in the application area of Autonomous Driving which can be assembled and extended in several ways due to the functional composability.

The elaborated conceptual structure/architecture is supplemented with a prototype Digital Twin implementation based on an existing physical entity (PT), considering its design, functionalities, and limitations. The implemented DT should include control and adaptation mechanisms to enable the DT to continuously shadow and react to changes in the physical counterparts.

The first step is to conduct a thorough review and clarification of the physical entity to identify its capabilities, the model requirements, the methods for data transmission (connection, interaction, protocol, etc.) between the Twins, as well as possible limitations and restrictions, such as latency and synchronization issues.

The next step is the implementation of a shadow-DT and subsequently a fully interacting DT, thereby considering possible limitations and restrictions (e.g. on PT side: processor speed, memory capacity, investigating the possibility to reprogram in real-time Over-the-Air, etc.).

After that, the potential of changing and modifying as well as adding and/or subtracting and/or replacing elements and scenarios (also such as simulation) are investigated and implemented as far as possible, thereby considering possible limitations and restrictions.

Finally, an evaluation of the DT is carried out using illustrative scenarios that relate to the implemented prototype.

1.3 Research Questions and problem-solving approach

RQ1 : What Digital Twin could be elaborated for the context of Autonomous Driving, with specific consideration given to flexibility, composability and extendability?

Work Package WP1: Solution exploration

- Researched literature with regard to DT, PT and Autonomous Driving, and software solutions.
- Gained a full understanding of the problem, including its characteristics and properties, and developed an appropriate solution to it, using established knowledge and best practices where appropriate.

Work Package WP2: Conceptual design elaboration

- Designed and developed a conceptual structure/architecture for a DTs in the context of Autonomous Driving, followed by the elaboration of a graphical representation of the developed conceptual structure/architecture.

  RQ1.1 : Which conceptual architectures exist?

  RQ1.2 : What are the key components and connections of a conceptual structure/architecture for a DT?

  RQ1.3 : What concept and meta-model could be elaborated for a Digital Twin in the context of Autonomous Driving, with specific consideration given to the flexibility, composability and extendability of the proposed approach?

  RQ1.4 : Which Use Case can be defined and which architectural framework can be derived from the concept and its meta-model of a Digital Twin for Autonomous Driving, especially with regard to flexibility, composability and expandability?

Work Package WP3: Implementation and evaluation

- Implemented a DT prototype for the PT Robot Cars (researched, analysed and selected appropriate software tools, as well as coding, etc.). This included the elaboration of the design of the DT and the relevant diagrams (e.g. architecture, structure, flow). Therefore, a comprehensive review and clarification of the physical entity (Physical Twin) including its design, capabilities, functionalities and limitations (e.g. memory capacity, processor speed, hardware, software, etc.) had to be conducted. In addition, it was necessary to thoroughly engineer the interface and interconnection between the DT and the Physical Twin (communication hardware, protocols, MQTT, layers, latency issues, etc.).

  RQ1.5 : What prototype implementation and with which solution approach could supplement the Use Case?

- Illustrative application scenarios are devised to demonstrate the prototype implementation of a Digital Twin for Autonomous Driving, with particular consideration given to the aspects of flexibility, composability and extendability.

Work Package WP4: Aggregated the outcome and observations, and supplemented it with a brief outlook on possible future works.

1.4 Thesis Structure

The remainder of this paper is structured as follows:

Chapter 2 – Foundations: The essential theoretical underpinnings pertinent to this thesis are presented. This includes an explanation of key foundational terms and definitions, a presentation and identification of pertinent existing concepts and architectures for Digital Twins, and an examination of existing Autonomous Driving architectures within and beyond the Digital Twins domain, including applications within the automotive sector

Chapter 3 – Related Work This chapter encompasses pertinent works related to Digital Twins and Autonomous Driving

Chapter 4 – Requirements: Thirdly, derived from the insights of the existing works discussed in Chapters 2 and 3, this chapter presents the specific requirements for the concept and the implementation showcased in this thesis

Chapter 5 – Concept: This chapter describes a concept for a composable, flexible, and extendable Digital Twin in Autonomous Driving, emphasizing efficient data exchange, separation of concerns, a unified communication structure, and component adaptability. It also details a use case

Chapter 6 – Implementation of a Digital Twin for Autonomous Driving: Subsequently, this chapter elaborates on the prototype DT implementation for Autonomous Driving, including the analysis of the Physical Twin, selection of development tools, and detailed components and architecture of the DT

Chapter 7 – Evaluation: An evaluation is conducted to determine the validity of the presented concept, Use Case, and implementation

Chapter 8 – Conclusion and Outlook: To conclude, the thesis highlights the principal achievements and limitations, and discusses prospects for future investigations

Additionally, the following appendices provide supplementary material:

Appendix A – Supplementary Material for the Robot Cars: This appendix encompasses the Robot Car Hardware Specifications, Robot Car Configuration Data, and Robot Car Communication Protocols, serving as an essential resource for those aiming to reproduce, enhance, expand, or adapt the implementations presented. Additionally, it includes supplementary images of the Robot Cars

Appendix B – Supplementary Material for the Software Tools: This appendix provides an overview of the software tools and versions used for the implementation of the DT-Entities, DT-Entities-Scenarios, MQTT broker, Data Manager, Dashboard, and Database in the prototype DT

Appendix C – Structure of the Prototype DT-Entities Implementation coded in Java: This appendix provides more details on the structure and implementation of the prototype DT-Entities and DT-Entities-Scenarios coded in Java

Appendix D – Supplementary Information on Design Decisions for Dashboard: This appendix provides supplementary information regarding the design decisions that were made during the implementation of the Dashboard

Appendix E – Summarized Considerations to RQ1.3: This appendix provides a summary of the considerations related to **RQ1.3**

Appendix F – Summarized Considerations to the Use Case: This appendix provides a summary of the considerations related to the Use Case

Appendix G – Supplementary Material for the Database: This appendix provides a table describing the SQLite tables of the `DB_DT` Database

2 Foundations

This chapter consolidates the theoretical foundations pertinent to this thesis. It opens with an explanation of key foundational terms and definitions in Section 2.1. The subsequent sections, specifically 2.2 and 2.3, comprehensively present and identify pertinent existing concepts and architectures for Digital Twins. Additionally, Sections 2.4 and 2.5 explore existing Autonomous Driving architectures, both within and beyond the Digital Twins domain, including some applications within the automotive sector. The Sections 2.3, 2.4 and 2.5 address also the research question, **RQ1.1**: Which conceptual architectures exist?

2.1 Foundational Terms and Definitions

This section presents necessary foundational terms and definitions. For terminology see also [EI90].

2.1.1 Model

This thesis follows Stachowiak's definition of a model as "*a formal representation of entities and relationships in the real world (abstraction) with a certain correspondence (homomorphism) for a certain purpose (pragmatics)*" [Sta73]. Every model is an abstraction of its original, meaning that it excludes details that do not serve its purpose. Therefore, a model does not contain all attributes of its original, and there may even be changed or additional attributes [Sta73]. Statements about model elements apply to real-world entities (homomorphism). A model and its meta-model are created with a goal-driven approach (pragmatics), such that a model can replace the original under certain assumptions or for specific questions [Sta73]. The use of models offers several benefits. Abstraction leads to concise specifications, reducing complexity and increasing understandability, making (automatic) analyses feasible. Models also improve communication efficiency by providing consistent documentation, making them usable by domain experts via Domain-Specific Languages (DSLs) [Sta73].

2.1.2 Meta-Model

A meta-model is a precise definition of the parts and rules required to create valid models. It includes an abstract syntax, at least one concrete syntax, and static and dynamic semantics [Sta73].

2.1.3 Digital Model

A Digital Model, as defined by Kritzinger et al. [KKT+18], describes a digital representation of an existing or planned physical object that does not rely on any form of automated data exchange between the physical and digital objects. A Digital Model could be for example a Computer-Aided Design (CAD) of a vehicle or object. While Digital Models may be developed using digital data from existing physical systems, all data exchange is performed manually, without the use of automated processes. As a result, changes in the state of the physical object have no direct impact on the digital object, and vice versa. Depending on the application domain, a Digital Model may include descriptions of the physical object, simulation models of planned factories, mathematical models of new products, or any other models of a physical object that do not employ automatic data integration.

2.1.4 Digital Shadow

A Digital Shadow, as defined by Kritzinger et al. [KKT+18], represents a Digital Model with the extension of an automated one-way data flow from the state of an existing physical object to the state of a digital object. This means that any change in the state of the physical object will result in a corresponding change in the state of the digital object, but not vice versa.

2.1.5 Digital Twin (DT)

According to the glossary of the Digital Twin Consortium [Dig24] *"A digital twin is an integrated data-driven virtual representation of real-world entities and processes, with synchronized interaction at a specified frequency and fidelity. Digital Twins are motivated by outcomes, driven by use cases, powered by integration, built on data, enhanced by physics, guided by domain knowledge, and implemented in dependable and trustworthy IT/OT/ET systems."*
The Digital Twin Consortium also provides definitions for a Digital Twin system (depicted with a structure illustration), for a Digital Twin platform (defined as a set of integrated services, applications, and other Digital Twin subsystems that are designed to be used to implement DTs), for Digital Twin Services, etc.

A Digital Twin (DT), as defined by Kritzinger et al. [KKT+18], represents a Digital Shadow with the extension of having data flows between an existing physical object (Physical Twin (PT)) and a digital object which are fully integrated in both directions. A change in the state of the physical object directly leads to a change in the state of the digital object and vice versa. The digital object might also act as a controlling instance of the physical object and there might also exist other objects, physical or digital, which induce changes of state in the digital object.

The definition of a Digital Twin used in this paper is based on the one presented in [Wor]:
"*A Digital Twin is a software system that leverages models and data from and about an original (cyber-physical) system, to represent, predict, and prescribe its behaviour for a specific purpose.*"

DTs are built for various purposes, such as developing, analysing, simulating, visualising, and optimizing non-digital systems [WEB+21]. The common goal is to enhance competitiveness, productivity, and efficiency [KKT+18]. This includes single components of the system up to the

entire assembly of the system [KKT+18].

Here are some of the key features and specifications of Digital Twins, according to Wortmann [Wor]: DTs are used to operate, monitor and control the resulting Cyber-Physical Systems in various domains such as for example automotive, avionics, manufacturing, and medicine. They have the potential to significantly reduce costs and time and improve our understanding of the represented systems. Digital Twins are used at different times in relation to the system they represent. For instance, a DT may be used to explore the design space before the system exists, or it can optimize the system's behaviour during its runtime. A DT is not restricted to a specific technology or application domain, and it does not need to be complete, which is often impossible. Furthermore, a DT does not need to be a model, but it may use models and can alter the behaviour of the original system.

2.1.6 Autonomous Driving

An autonomous vehicle has the capability to perceive the surrounding environment and navigate itself. For Autonomous Driving, heterogeneous set of sensors, actuators and computing elements that execute complex algorithms to perform functions such as perception, localization, planning, control, etc. [KJD+15a]. Autonomous Driving refers to the capability of vehicles to operate at levels zero to five of autonomy as defined by the Society of Automotive Engineers (SAE) [Sur]. It is also referenced as Advanced Driver Assistance Systems [GFSA17]. It aims to improve safety and efficiency in the transportation system [YCT+22]. Established classifications and functions are e.g. "Autonomous Emergency Braking", "Frontal Collision Warning", "Lane Departure Warning" or "Lane Keeping" [GFSA17]. It involves the integration of data, RADAR and LIDAR technologies, sensors, cameras, software, algorithms, Advanced Driver Assistance Systems and eventually AI and probably Machine Learning, etc. [Sur].

2.1.7 Messaging

Messaging is a communication method used in distributed systems to exchange information between different components or services. It allows for asynchronous communication, where the sender and receiver do not need to interact with the message at the same time. This decoupling of communication enhances the scalability and flexibility of the system. Messaging systems can be categorized based on their communication patterns, such as point-to-point and Publish/Subscribe (Pub/Sub). The Pub/Sub model, which is discussed in detail in Subsection 6.1.1, supports loose coupling, discussed in Subsection 2.1.8, enhancing the modularity and maintainability of the system.

2.1.8 Loose Coupling

Loose coupling refers to a design principle in which components or services in a system are minimally dependent on each other. This principle enhances the modularity and maintainability of the system, as changes in one component do not significantly impact others. Loose coupling is achieved through well-defined interfaces and communication protocols that abstract the underlying

implementation details. In messaging systems, loose coupling is facilitated by the Pub/Sub model, as presented in Subsection 6.1.1, where publishers and subscribers are decoupled through a central broker. This allows for independent development, deployment, and scaling of different components in the system.

2.2 Existing Concepts of Digital Twins

Concept involves the theoretical understanding and definition of key characteristics and applications. It forms the foundation for practical implementations. Concepts are the building blocks of software systems, representing increments of functionality introduced by designers to serve particular purposes. They are objective features of a system's design, essential for understanding and implementing the system's behaviour. In essence, a concept can be seen as the theoretical underpinning that informs both the architecture and any frameworks used within the system [Jac15]. The conceptualization of Digital Twins involves defining their key characteristics and applications.

2.2.1 Conceptualizing Digital Twins [WEB+21]

Wortmann et al. [WEB+21] provide a conceptualization of DTs, highlighting their potential applications in various domains. While specific tools or technologies for realizing DTs are not discussed, the authors aim to categorize the different roles and relationships of artifacts on a conceptual level. This work emphasizes the importance of a clear and consistent definition of DTs to ensure their effective implementation. Key aspects include the necessity for the DT to accurately represent the physical system and to have access to real-time data from the physical system to provide accurate simulations and predictions.

The presented conceptual framework comprises a set of **models** and **data** associated with an **Actual System (AS)** that enables the creation of various **services**. The AS is the system associated with the DT, producing data related to different aspects of the system. Collecting, storing, calculating, and inferring data relevant to the AS and its environment is essential for the DT. The DT captures this data and uses it to conduct various operations and actions on the AS.

The data component within the DT, referred to as Digital Shadow, involves the storage and representation of current and past data of the AS. According to [WEB+21], *"Digital Shadows are purposefully abstracted and aggregated Data Structures that represent one-way data flow between the state of the AS and its digital representation"*. Data flows include monitoring and sensing data, measured data, and external/historical data from the system or its environment.

The DT of the presented conceptual framework includes one or more models of the system or its parts, addressing different aspects and disciplines such as engineering, software, and scientific models. Models can be of various language types and play three **roles**: **Descriptive Model**, **Predictive Model**, and **Prescriptive Model**.

The Descriptive Model represents current or past aspects of the AS in a descriptive manner, facilitating understanding and analysis. The Predictive Model predicts information not measured, aiding decision-making and trade-off analyses. The Prescriptive Model describes the system to be realized, driving the constructive process and runtime evolution in self-adaptive systems. The

DT framework involves decision support activities, such as what-if analysis, and executing actions on the AS based on the prescriptive model. The DT can use simulation, even in communication with other DTs, to aid decision-making for maintenance or improvement of the AS based on new requirements and available resources. The framework is described using the Models and Data (MODA) framework [CKM+21], which supports the description of data-centric systems in terms of models, data, and transformations. This conceptual framework categorizes the different roles and relationships of artifacts on a conceptual level, extending existing viewpoints in the literature.
The abstract nature of this conceptual framework offers a foundation for further extension, allowing for the incorporation of specific components that a DT could consist of to support the use cases presented in [WEB+21].

2.2.2 Systematization of Digital Twins: Ontology and Conceptual Framework [BEFH20]

Barth et al. [BEFH20] discuss the systematization of Digital Twins, providing a comprehensive framework for understanding and classifying Digital Twins. The authors address the lack of a shared conceptual framework and unambiguous terminology in the field, proposing an ontology and conceptual framework to systematize the main dimensions of DTs. This research aims to represent the entire spectrum of DTs, ensuring universal validity across various domains and applicability in both research and practice. The study is grounded in a systematic literature review and iterative workshops with academic experts, emphasizing the conceptual foundations of Digital Twins. Notably, [BEFH20] presents a Digital Twin ontology, which is elaborated through several key dimensions:

- Data Resources: This dimension encompasses the sources, categories, and formats of data required to create DTs. It highlights the importance of digital data in the creation and functioning of DTs.
- Internal Value Creation: This dimension focuses on the lifecycle phases of products, product management levels, and the different generations of both. It emphasizes how DTs integrate physical and digital worlds to create internal value.
- External Value Creation: This dimension includes the attributes of services, the level of smartness of connected products, and the actors within the ecosystem. It underscores the role of DTs in creating external value through smart services.

The results of this work demonstrate that the proposed ontology and conceptual framework support researchers and practitioners in positioning and structuring their DT activities. It aims to facilitate communication with internal and external stakeholders and provides a holistic view of the data resource dimension, allowing for the deduction of necessary data for specific applications.
This work is a suitable foundation for developing a concept for Digital Twins tailored to the use case of Autonomous Driving, as it offers a structured approach to understanding and leveraging DTs in various contexts.

2.3 Existing Digital Twin Architectures

Architecture refers to the structural design of a system, detailing the components and their interactions. It provides a blueprint for the system's construction and operation. In software engineering, architecture encompasses the high-level structures of a software system and the discipline of creating such structures. It depicts the system's components and their interactions, serving as the blueprint for the entire system. Furthermore, a framework can be seen as part of the architecture, providing reusable design patterns and guidelines to ensure consistency across different implementations [GLP17].
Digital Twin architectures have been extensively explored in various domains, including manufacturing and Autonomous Driving.

2.3.1 Model-Driven Development of a Digital Twin for Injection Molding [WBD+20]

Wortmann et al. [WBD+20] discuss the model-driven development of DTs for injection molding, emphasizing the importance of a robust architecture for accurate and efficient simulations. They propose a systematic approach to engineering DTs that supports domain-specific customizations and automates essential development activities using a model-driven reference architecture. This approach defines reactive CPPS behaviour through a DSL for event specification and connects to the CPPS via a novel DSL for OPC-UA bindings.

The reference architecture is described as a component and connector architecture in MontiArc, which supports refinement from abstract requirements to detailed technical specifications. The architecture consists of four layers:

1. Cyber-Physical Layer: Describes the CPPS controlled by the DT, requiring interfaces for data access and command execution. This layer ensures that the physical system can communicate with the DT.
2. Data Layer: Contains a Data Lake for storing diverse data from the CPPS, annotated with metadata for reusability. The Data Lake stores both unstructured and structured data, supporting data preparation and processing.
3. Connection Layer: Includes a **Data Processor** and an **Executor**.
   The Data Processor links the Data Lake with the application layer, creating Data Structures (DS) required by the application layer. It has two inner components:
   - **Data Processor Logic**: Receives DS queries from the application layer, transforms these into data requests, and creates DS from the results.
   - **Data Processor Adapter**: Transforms data requests into queries for specific databases within the Data Lake.

   The Executor derives and executes solutions at the CPPS. It has two inner components:
   - **Execution Logic**: Derives a solution to be executed at the CPPS and its surrounding systems.

- **Execution Adapter**: Sends commands to specific parts of the CPPS and processes **feedback** about the success of these commands, which is then handed back to the application layer.

4. Application Layer: Contains the smartness of the DT, with an **Evaluator** and **Reasoner**. The Evaluator analyses DS, detects events, and creates goals. It relies on design-time models and the **Knowledge Base** to decide when an event is considered negative and must be handled. The Reasoner receives goals from the Evaluator and uses the knowledge contained in the Knowledge Base to create solutions that realize these goals.

The DT's architecture facilitates the creation of a **Digital Shadow** which allows for continuous monitoring and optimization of the injection molding process.

Wortmann et al. [WBD+20] developed a model-driven methodology that automates the generation of DTs from models describing a CPPS and its domain. This methodology involves creating a domain model, specifying events and actions using a Domain-Specific Language, and enriching the domain model with data retrieval information. The approach supports extension and adaptation to various domains, ensuring flexibility and reusability.
The presented methodology and reference architecture enable the generation of a DT for setting up and executing a DoE on an injection molding machine. Currently, a parameter change within the controlled CPPS requires a new generation of the DT. Future work will focus on operating on interpreted models to avoid redeploying the DT. The DT gathers relevant data, transmits commands to the machine, and reacts to detected events. However, full automation is not yet possible due to the need for a machine operator to supervise the production process. The current implementation is a proof-of-concept, and further integration with additional assets and enhanced automation is needed.

2.3.2 Digital Twin in manufacturing: A categorical literature review and classification [KKT+18]

Similarly, Kritzinger et al. [KKT+18] provide a comprehensive literature review and classification of Digital Twin architectures in manufacturing, emphasizing the necessity for a systematic approach to design and implementation. As previously discussed in Section 2.1, Kritzinger et al. [KKT+18] distinguish **Digital Twins** from **Digital Models** and **Digital Shadows**.

To summarize briefly, Kritzinger et al. [KKT+18] present the following key points:

A **Digital Model** (see 2.1.3) implies the absence of automated data flows between the physical object and the digital object, necessitating manual data exchange.

A **Digital Shadow** (see 2.1.4) extends the concept of a Digital Model by incorporating an automated one-way data flow from the state of the physical object to the state of the digital object. "Shadowing" thus refers to the digital object automatically recognising and adapting to changes in the physical object [KKT+18].

A **Digital Twin** (see 2.1.5) is characterized by the presence of automated two-way data flows, wherein changes in the physical object are recognised and adapted in the digital object and vice versa.

This high-level abstract view and distinction provided by Kritzinger et al. [KKT+18] serves as a foundational clarification of terminology for this thesis. Building upon this work, the elaborated concept presented in the following Chapter 5 aims to describe all necessary data flows in a manner that ensures the desired outcomes, thereby establishing the concept as a true Digital Twin.

2.3.3 Five-Dimension Digital Twin Model [TLZ+19]

Tao et al. [TLZ+19] introduce the five-dimension Digital Twin model and its ten applications, providing a comprehensive overview of the potential applications of Digital Twins in various domains. This model, which builds upon the initial Digital Twin model introduced by Grieves [Gri05], serves as a guideline for the development and architecture of Digital Twins, ensuring their scalability and adaptability to different applications. In particular, the mentioned dimensions consist of:

Physical Entity represents the real-world physical object or system that is being modeled. It includes all the tangible aspects and characteristics of the entity.

Virtual Entity involves the digital representation of the physical entity. It includes the creation of a virtual model that mirrors the physical entity's attributes and behaviours.

Data encompasses the data collected from the physical entity through sensors and other data acquisition methods. It includes real-time data, historical data, and any other relevant information.

Connection refers to the communication and interaction between the physical and virtual entities. It includes the data transmission, synchronization, and integration processes that ensure the virtual model accurately reflects the physical entity.

Service involves the various services and applications that can be derived from the Digital Twin model. It includes analysis, simulation, optimization, and other functionalities that provide value to users.

The model presented by [TLZ+19] offers a model for the development and architecture of Digital Twins, establishing a suitable foundation for the concept of this thesis presented in Chapter 5.

2.4 Existing Digital Twin Architectures for Autonomous Driving

The application of Digital Twins in Autonomous Driving has gained significant attention in recent years.

2.4.1 Development of Autonomous Car—Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture [KJD+15b]

The IEEE Transactions on Industrial Electronics [KJD+15b] present a comprehensive case study from Kichun et al. on the implementation of an Autonomous Driving system based on distributed architecture, underscoring the pivotal role of Digital Twins in the development and testing of autonomous vehicles. Key components include the segmentation of the system into multiple subsystems, each responsible for specific functions, thereby enhancing modularity and scalability, and the provision of a robust simulation environment for testing and validating the Autonomous Driving system, ensuring real-world applicability and reliability.

According to Kichun et al. [KJD+15b], the development of Autonomous Driving systems employs a distributed architecture approach to significantly enhance the efficiency and reliability of autonomous vehicles. This study provides an in-depth analysis of the structural design of Autonomous Driving algorithms, exploring their functional components and interactions, which are crucial for improving real-time decision-making and operational robustness. Specifically, the authors discuss key elements such as Autonomous Driving Algorithm and Behaviour Reasoning.
The Autonomous Driving Algorithm section dissects the comprehensive structure of Autonomous Driving algorithms, focusing on several critical components:

Localization: This component ensures the vehicle knows its precise location within its environment. Using GPS data, Inertial Measurement Unit (IMU), and map information, the localization algorithms refine the vehicle's position, which is crucial for accurate navigation and decision-making.

Perception: This subsystem involves the collection and interpretation of sensor data to understand the vehicle's environment. It includes various sensors like cameras, LIDAR, RADAR, and ultrasonic sensors. The data is processed through algorithms for object detection, classification, and tracking, enabling the vehicle to perceive obstacles, road signs, pedestrians, and other vehicles.

Sensor Fusion: To enhance accuracy, data from multiple sensors are fused. This process involves algorithms that merge information from different sources to create a coherent understanding of the vehicle's surroundings. Multisensor fusion mitigates the limitations of individual sensors by leveraging their complementary strengths.

Planning: Based on the perceived environment and current location, the path planning algorithms determine the optimal route for the vehicle. This involves short-term trajectory planning for immediate manoeuvres and long-term path planning for overall route optimization. The algorithms consider factors like road geometry, traffic rules, and dynamic obstacles.

Vehicle Control: These control systems algorithms convert the planned path into actionable commands for the vehicle's actuators. They manage steering, acceleration, and braking to ensure smooth and safe navigation. Control systems use feedback loops to adjust actions in real-time, accounting for any deviations from the planned path.

The Behaviour Reasoning section delves into the decision-making processes that govern the vehicle's actions in various driving scenarios:

Decision-Making Algorithms: These algorithms determine the appropriate behaviour in response to dynamic driving conditions. They analyse data from the perception and localization subsystems to predict the actions of other road users and make decisions accordingly.

Overtaking: The behaviour reasoning module evaluates when and how to overtake slower vehicles safely. This involves assessing the speed and position of surrounding vehicles, determining safe gaps in traffic, and executing overtaking manoeuvres without violating traffic rules or compromising safety.

Crosswalks and Pedestrian Interaction: The algorithms are designed to recognise crosswalks and anticipate pedestrian behaviour. They ensure the vehicle yields to pedestrians, stops at crosswalks, and navigates through pedestrian-dense areas cautiously.

Intersection Handling: The reasoning processes handle the complexity of navigating intersections. This includes recognising traffic signals, yielding right-of-way, and making decisions about when to proceed, turn, or stop, ensuring compliance with traffic regulations and safety considerations.

Merging and Lane Changes: The system evaluates the surrounding traffic to perform safe merging onto highways and executing lane changes. It calculates the timing and speed adjustments needed to integrate seamlessly into traffic flow.

In summary, Kichun et al. [KJD+15b] offer an examination of distributed architecture and Autonomous Driving algorithms, showcasing significant advancements in autonomous vehicle technology, thereby forming a groundwork for this thesis by identifying key Autonomous Driving components and functionalities to validate the proposed concept.

2.4.2 Fusion: A Safe and Secure Software Platform for Autonomous Driving [MPS20]

Mundhenk et al. [MPS20] delve into the development of a secure and robust software platform for Autonomous Driving, emphasizing the role of Digital Twins in ensuring vehicle safety and reliability. This platform incorporates essential safety mechanisms to prevent accidents and security measures to safeguard against cyber-attacks and ensure data privacy. They provide a comprehensive overview of a distributed architecture for Autonomous Driving, highlighting the Fusion software platform developed by Autonomous Intelligent Driving GmbH as a foundation for deploying self-driving vehicles. Central to the discussion is a high-level view of the self-driving system and its interfaces, outlining the following components and their interactions:

- Visualisation describes tools and methods for visualising the sensor data and system status.
- Simulation describes environments for testing and validating the system in various scenarios.
- Remote Control describes capabilities for remote monitoring and intervention.
- High-level Functionality refers to the core functionalities that ensure the vehicle operates safely and efficiently.
- Sensors (Camera, LIDAR, RADAR, GPS, IMU) are used to detect and recognise objects and environments, which is critical for navigation and decision-making.
- Perception processes sensor data to detect and recognise objects and environments, which is critical for navigation and decision-making.
- Prediction forecasts the future positions and actions of objects in the environment.
- Environment Model acts as a dynamic model to represent the surrounding environment.
- Trajectory is for planning and adjusting the vehicle's path.
- Routing determines the optimal routes for the vehicle to take.
- Control executes decisions by managing steering, acceleration, braking, and other mechanical functions to ensure precise control and safety.
- Motor is a system that manages the vehicle's movement.
- Localization determines the vehicle's position within its environment using GPS and other sensor data.
- Maps are detailed maps of the driving environment for navigation.
- Middleware / Software Platform is the foundational software that integrates all components and functionalities.
- Hardware comprises the physical components and systems that support the software.

The architecture ensures modularity and scalability, allowing for the integration of new technologies. The Digital Twin simulation environment supports testing and validation by replicating real-world scenarios for extensive refinement.
In conclusion, [MPS20] provides valuable insights into a secure and reliable Autonomous Driving system, highlighting significant advancements in the field.

2.5 Existing Autonomous Driving Architectures

This section provides an overview of foundational frameworks developed to advance Autonomous Driving technologies. It highlights significant contributions and methodologies from key research works in the field.

2.5.1 Functional Architecture for Autonomous Driving [BT15]

Behere and Torngren [BT15] present a functional architecture for Autonomous Driving, focusing on a single vehicle's motion-specific subsystems while excluding cooperative driving and other Intelligent Transport System (ITS) scenarios, and contribute to the literature by discussing key elements, proposing a layered division of the architecture, and explaining the distribution of these components within the architecture.
Notably, Behere and Torngren [BT15] emphasize the key elements of a Functional Architecture View (FAV) for an Autonomous Driving system, as defined by the ISO26262 functional safety standard, which describes the *functional concept* as a "specification of the intended functions and their interactions necessary to achieve the desired behaviour" [Int11]. This definition aligns with the term FAV as used in this context.
The principal FAV components are divided into three categories: perception, decisions and control, and vehicle platform manipulation.

Perception involves sensing and interpreting data into high-level concepts relevant to the task.
: Sensing refers to gathering data using sensors, while perception interprets this data to understand and make sense of it in the context of the task.
: Sensing components are categorized into those sensing the ego vehicle's states (internal environment) and those sensing the environment's states (external environment).
: Sensor fusion combines multiple information sources to hypothesize the environment's state, perform object association and tracking, and eliminate superfluous data to reduce computational load.
: Localization determines the vehicle's position relative to a global map using GPS and inertial measurement sensors, aiding in map matching and improving accuracy through visual landmarks.
: Semantic understanding shifts from sensing to perception, including object classification and behaviour prediction, and uses ego vehicle data for motion control and error detection.
: The world model represents the external environment's state as perceived by the ego vehicle, typically separated from the internal states due to technical and qualitative differences.

Decisions and control focus on vehicle behaviour in the external environment.
: Trajectory generation repeatedly creates and selects optimal obstacle-free trajectories, constrained by platform motion limitations, energy availability, and platform state.
: Energy management includes battery management and regenerative braking, with significant energy consumption from sensors and associated fusion and computation.
: Diagnosis and fault management identify the system's state to influence behaviour, manage redundancy, degrade capabilities systematically, and handle transitions to safe states.

Reactive control components respond immediately to unanticipated stimuli, overriding nominal behaviour requests with faster sense-plan-act loops.
The world model helps decision and control components generate potential futures for input actions, selecting the most desirable outcome.

Vehicle platform manipulation includes components responsible for vehicle motion, stability, and occupant protection.
Platform stabilization involves traction control, Electronic Stability Program, and anti-lock braking to maintain vehicle control and safety.
Trajectory execution components implement the generated trajectory through propulsion, steering, and braking, considered traditional in Autonomous Driving development.

[BT15] highlights the superior sensory capabilities of autonomous vehicle sensors compared to human drivers while acknowledging the current limitations in computer-based perception and interpretation. It underscores the challenges in incorporating learning into deployed vehicles due to validation requirements, suggesting advancements in virtual testing and correctness-by-construction methods as solutions. This is crucial for addressing the complexities of Autonomous Driving systems and their integration into Intelligent Transport Systems.
[BT15] contributes by detailing principal functional components and their distribution, providing a suitable foundation by outlining essential Autonomous Driving elements that can be assessed to determine the relevance of the proposed concept of this thesis.

2.5.2 Functional System Architectures towards Fully Automated Driving [TKZS16]

Taş et al. [TKZS16] provide a comprehensive framework for understanding the critical components and interactions within automated driving systems. This work systematically addresses the essential aspects of such systems, making it a valuable reference for ensuring that all relevant factors are considered in the Digital Twin concept. [TKZS16] offers insight into the automated vehicle as a cognitive system. Automated vehicles function as cognitive systems, integrating various subsystems to perceive their environment, make decisions, and execute actions.
Key components include:

- Cognitive System: The vehicle operates as a cognitive system, coordinating multiple subsystems to achieve Autonomous Driving.
- Perception: Utilizing sensors such as GPS, IMU, on-board sensors, and cameras, the vehicle gathers data about its environment. This data is processed to understand the vehicle's position, state, and surroundings.
- Cognitive Decision: The vehicle employs complex algorithms to analyse the perceived data, predict potential scenarios, and determine the optimal course of action.
- Action: The vehicle executes the decided actions, including steering, accelerating, and braking.
- Environment: Continuous interaction with the environment allows the vehicle to adapt to changes and navigate safely.

Additionally, [TKZS16] delves into the information flow between modules of the automated vehicle A1, which won the Korean Autonomous Vehicle Competition. This vehicle exemplifies a system architecture with seamless information flow between its modules:

- Sensors: Equipped with GPS, IMU, on-board sensors, and cameras, the vehicle collects comprehensive data about its position, state, and environment.
- Perception and Scene Understanding: The perception module processes sensor data to estimate the vehicle's position and state, detect moving objects, and understand the scene. Sensor fusion combines data from multiple sources for a holistic understanding.
- Behaviour and Motion Planning: This module determines the vehicle's path and actions based on the perceived data, ensuring safe and efficient navigation.
- System Management: Overseeing the entire system, the system management module detects failures and backs up failed units, ensuring robustness and reliability.

The architecture of the A1 vehicle demonstrates a basic structure for maintaining robustness. By integrating these modules and ensuring smooth information flow, the vehicle can navigate complex environments and handle unexpected situations.
The structured approach presented in [TKZS16] provides a suitable foundation for designing a Digital Twin concept for Autonomous Driving. By outlining essential Autonomous Driving elements, this work allows for a thorough assessment of the relevance of the proposed Digital Twin concept, ensuring a comprehensive and reliable design.

2.6 Pertinent Foundations (to RQ1.1)

To answer this RQ, the author researched concepts of Digital Twins in general, as well as for architectures for Autonomous Driving:

Conceptualizing Digital Twins [WEB+21] by Wortmann et al. provides a framework for understanding the roles and relationships of Digital Twin artefacts conceptually, underscores their potential applications in various fields, and highlights the necessity of consistent definitions for effective implementation, utilizing the MODA framework to support the description of data-centric systems in terms of models, data, and transformations.

Model-Driven Development of a Digital Twin for Injection Molding [WBD+20] by Wortmann et al. details a methodology for generating Digital Twins from CPPS models, designed for domain flexibility and reusability, and highlights the significance of a robust component and connector architecture for efficient simulations.

Digital Twin in manufacturing: A categorical literature review and classification [KKT+18] by Kritzinger et al. offers a comprehensive literature review and classification of Digital Twin architectures in manufacturing, highlighting the need for a systematic approach to design and implementation, while distinctly separating Digital Twins from Digital Models and Digital Shadows at a conceptual level.

Development of Autonomous Car—Part II [TLZ+19] by Tao et al. outlines a case study on the Autonomous Driving system implementation using distributed architecture, accentuating Digital Twins' importance and focusing on modularity, scalability, subsystem segmentation, and reliable simulation environments.

3 Related Work

To investigate the creation process of a composable Digital Twin and to develop an appropriate conceptual structure or architecture, existing works in this field were examined. This chapter also addresses the research question **RQ1.1**: Which conceptual architectures exist?

3.1 Methodology

The search was conducted using extensive combinations of the following search terms and keywords, utilizing the search engine Google Scholar Germany[1] and others:
digital, physical, twin / twins, framework, architecture, concept, development, software, composable, distributed, assistance, systems, loose, coupling, best, practices, autonomous, driving, assistant, car, connected, vehicle, automotive, AGV, IoT, Things, IoV, WhiteLabelDT, WLDT, Java, MQTT, self, adaptive, adaptation, control, functional, evaluation, feedback, loop / loops, MAPE-K, UML, MontiArc, metrics

Additionally, backtracking of references was employed to identify further relevant results.

[1] https://scholar.google.de

3.2 Related Work for Conceptualization and Architectures of Digital Twins

This section provides related work for the conceptualization and architectures of Digital Twins.

3.2.1 Conceptualizing Digital Twins [WEB+21]

Wortmann et al. [WEB+21] propose a conceptual modelling framework for DTs that captures the combined usage of heterogeneous models and their respective evolving data for the twin's entire life cycle. They explain the main components of a DT framework, namely: Actual System and its environment, Data (storage and flows) and Models. They identified three roles (as defined in the MODA framework) that models can play in a DT: Descriptive Model, Predictive Model and Prescriptive Model. They also illustrated a set of DT applications, especially related to Industry 4.0, as instances of their conceptual framework. In their conclusion, they highlight that the framework offers a structured language for discussing Digital Twins, particularly from a software engineering perspective. Furthermore, it facilitates the reuse of concepts across different Digital Twins, as many are expected to rely on hybrid architectures comprising various presented variants. They also emphasize that the proposed framework can be applied to specific cases using concrete modeling techniques and associated tools while noting that addressing specific integration issues is necessary.

3.2.2 Model-Driven Development of a Digital Twin for Injection Molding [WBD+20]

Wortmann et al. [WBD+20] present a model-driven methodology to develop efficient DTs for CPPS as well as a reference architecture (as a component and connector architecture) and a development process that facilitates adaptivity and extensibility. facilitating the exchangeability of components of the DT. Furthermore, they introduced a DSL to specify events that occur in the CPPS and how the DT reacts to these events. Models of this DSL are integrated into the software architecture model. From these models, an integrated, reactive DT is generated. The DT reference architecture presented in this work is tailored for injection molding and may not be optimally suited for the Autonomous Driving domain, primarily due to the potential need for greater interaction between components. Furthermore, parameter changes within the controlled CPPS of this work necessitate a new code generation for the DT. These, as well as other aspects, need to be further investigated.

3.2.3 On the Engineering of IoT-Intensive Digital Twin Software Systems [RMV+20]

Rivera et al. [RMV+20] point to factors impacting DTs such as real-time aggregation, integration, interoperability, fidelity, resolution, completeness, etc. The authors present a self-adaptive model-driven architectural reference model (GEMINIS). The authors explain that it provides explicit functional decomposition and data as well as control flows and they illustrate this with a view of the reference model. It is influenced by external reference architecture (hierarchical orchestration of MAPE-K feedback loops) and reference model (separation of concerns and three-layer structure of feedback loops). They also present a preliminary version of a process for engineering DTs. It targets software engineers and researchers. Rivera et al. emphasize engineering principles over concrete architecture, including detailed components and connections.

3.2.4 WLDT: A general purpose library to build IoT Digital Twins [PMZ21]

Picone et al. [PMZ21] present a new, powerful, modular and flexible Java library, called White Label Digital Twin (WLDT), designed to maximize modularity, re-usability and flexibility. It focuses on the simplification of the design and development of DTs and provides a set of core features and functionalities for utilization in multiple applications. It integrates a multithreading core engine, a set of built-in IoT features and modules for smart objects using both MQTT and/or CoAP. WLDT is positioned to become an enabling building block for the design and development of DT-driven IoT applications. The targeted users are developers and researchers. While Picone et al.'s work on WLDT provides a general-purpose library for building IoT Digital Twins, it does not delve into a detailed architectural framework.

3.3 Related Work for Digital Twins for Autonomous Driving

This section provides related work for Digital Twins for Autonomous Driving.

3.3.1 Digital Twins for Autonomous Driving: A Comprehensive Implementation and Demonstration [WYL+23]

Wang et al. [WYL+23] present a real-time smart mobility DT framework for Autonomous Driving (End-to-End), as well as a comprehensive real-world implementation of this system. The captured real-world traffic information is processed in the cloud to create a DT model, which enables route planning services to avoid heavy traffic. The authors explain that it demonstrates high reliability and low latency. The presented architecture for a smart mobility DT focuses on high-level concepts without delving into the detailed implementation and interactions of the components.

3.3.2 AutoDRIVE: A Comprehensive, Flexible and Integrated Digital Twin Ecosystem for Autonomous Driving Research & Education [SSK+23]

Samak et al. [SSK+23] present a comprehensive, flexible, modular, integrated and expandable cyber-physical ecosystem (AutoDRIVE) for prototyping, simulating and deploying for Autonomous Driving. The authors explain that it features software and hardware-in-the-loop testing interfaces with openly accessible scaled vehicle and infrastructure components, and that it is compatible with a variety of development frameworks, and supports both single- and multi-agent paradigms through local as well as distributed computing. It comprises a hardware platform testbed (PT), a simulator (DT), and a Devkit (software packages, APIs and tools). They showcase exemplary use cases such as autonomous parking, intersections, etc. It targets virtual prototyping of autonomy solutions. While Samak et al. present a comprehensive and flexible Digital Twin ecosystem, their work does not specifically address an explicit architectural framework that encompasses the extendability of the Digital Twin.

3.3.3 Modeling Hardware and Software Integration by an Advanced Digital Twin for Cyber-Physical Systems: Applied to the Automotive Domain [KMG+20]

Kriebel et al. [KMG+20] present a comprehensive integrated modeling approach (MBCPSE, i.e. Model-Based, Cyber-Physical, and Systems Engineering), with a focus on facilitating hardware and software integration by an Advanced System Model. It supplements a function-oriented systems engineering with a solution-oriented model. Kriebel et al. contribute to the understanding of advanced Digital Twin modeling for the automotive domain, however, do not present a general Digital Twin architecture with the core elements.

3.3.4 Digital Twin Architecture for Autonomous Driving Validation and Verification [KKG23]

Kızılırmak et al. [KKG23] present a DT in order to accomplish validation and verification in a virtual environment and thereby reduce cost and development time. To showcase the capabilities they present an experimental setup with a PT robot car driving on a track and a software-based simulator. While Kızılırmak et al. present a framework for Digital Twin architecture in Autonomous Driving validation and verification, their work does not delve into detailed architectural models nor provide comprehensive concepts and insights.

3.4 Related Work for Autonomous Driving / Vehicle

The works presented in this section offer insight into the architectural, methodological and/or tool-related aspects of Autonomous Driving and Autonomous Vehicles. However, they do not focus on the application of Digital Twins.

3.4.1 A Standard Driven Software Architecture for Fully Autonomous Vehicles [SPV18]

Serban et al. [SPV18] present a design of a functional software architecture for fully autonomous vehicles and thereby follow the functional requirements from an automotive standard in terms of definition of the levels of driving automation etc. They support the design with a NIST Realtime-Control-System (RCS) reference architecture.

3.4.2 Tools and Methodologies for Autonomous Driving Systems [BAR20]

Bhat et al. [BAR20] present a design and reference architecture for Connected and Autonomous Vehicle (CAV) and describe the various components with their interactions They also present methodologies and tools for the modeling, design, development, and testing of CAV system.

3.4.3 Autonomous Driving Architectures, Perception and Data Fusion: A Review [VYBW20]

Velasco-Hernandez et al. [VYBW20] give a review on Autonomous Driving architectures and describe certain aspects of self-driving solutions as a component of the architectures (sensors, data-fusion, localisation, mapping, detection, etc.) and round up with an outlook on ongoing developments.

3.4.4 Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms [BS21]

Bachute and Subhedar [BS21] give a review and insights on Machine Learning and Deep Learning aspects for Autonomous Driving Systems, especially with regard to motion planning, localization, detection (pedestrian, traffic signs, road-markings), automated parking, etc.

3.4.5 A Software Architecture for Autonomous Vehicles: Team LRM-B Entry in the First CARLA Autonomous Driving Challenge [RGS+20]

Rosero et al. [RGS+20] present the development of a software architecture for an autonomous vehicle (for an autonomous vehicle challenge). They describe the utilized elements (e.g. LIDAR, GPS, etc.), structure, functionalities, and outcome.

4 Requirements

This chapter consolidates the requirements identified for this thesis.

The set of requirements presented in Table 4.1 was derived from the research questions and literature review, with particular emphasis on composability, extendability and flexibility, as these were deemed relevant to the prototype DT. The hereafter proposed requirements are non-exhaustive an are based on analysis of some contributions in the field of DTs and Autonomous Driving and explicitly apply to the DT of this thesis, as far as relevant.

The content of the table has been streamlined to include only the ID of the requirement with the associated description, with supplementary fields such as the requirement name or priority excluded.

ReqID	**ReqDescription**
Req01	The Digital Model (according to classification from Kritzinger et al. [KKT+18]) must be a digital representation of an existing or planned PT, that does not use any form of automated data exchange between the physical object and the digital representation, which might include a more or less comprehensive description of the physical object. All data exchange is done in a manual way. A change in the state of the physical object has no direct effect on the digital object and vice versa.
Req02	The Digital Shadow (according to classification from Kritzinger et al. [KKT+18]) must be a Digital Model with additionally an automated one-way data flow between the state of an existing physical object and a digital object. A change in the state of the physical object leads to a change of state in the digital object, but not vice versa.
Req03	The Digital Twin (according to classification from Kritzinger et al. [KKT+18]) must be a Digital Shadow with additionally a fully integrated data flow between an existing physical object and a digital object, in both directions. The DT might also act as a controlling instance of the PT. There might also be other objects, physical or digital, which induce changes of state in the DT. A change in the state of the PT object directly leads to a change in the state of the DT object and vice versa.
Req04	Representativeness and Contextualization: the DT must represent the PT in a specific context, i.e. a DT is a software implementation of a model representing the PT in a specific environment. Feature simplifications or generalizations are possible when they do not affect the behaviour in the specific context [MC21].
Req05	Reflection: the DT must mirror the behaviour and the status of the PT. Each change in status and each event faced by the PT is reflected by the DT. Changes that occur to the DT should be reproduced in the PT [MC21].

ReqID	ReqDescription
Req06	Entanglement: the DT and its PT must be constantly "connected" to instantaneously (i.e. in a time period consistent with requirements of applications) register any change in status. Entanglement requires effective, reliable communication between the PT and the DT suitable to the rate of changes [MC21].
Req07	Persistency: the DT must always be available. Its availability exceeds the actual existence of the PT. A DT could be available before the "creation" during malfunctioning and crashes, and after the end of life of the PT [MC21].
Req08	Memorization: the DT must store all the status changes and events that occurred to the PT. A DT represents the status of the PT over time and space [MC21].
Req09	Augmentation: the DT could extend the PT functions and offer them by means of APIs. Augmentation can add new functionalities that the PT does not support or provide access to data in particular formats [MC21].
Req10	Composability: Physical systems are aggregations of subsystems and components. The DT must support the correlation of different elementary DTs into complex organizations and provide views on the aggregated DT and individual components [MC21].
Req11	Replication: Replicas of the same PT must behave consistently, i.e. they cannot have a different status and they cannot exhibit different behaviours. the DT could be replicated to serve the needs of different applications. All the replicas of a DT tracking the actual status of a PT must/should have the same status at the same time t (with time limitations explained for entanglement). DTs used for simulations can diverge depending on the simulation conditions [MC21].
Req12	Accountability/Manageability: the DT must be manageable, i.e. it is possible to determine its status and activities and to optimize its execution in the framework in which it is operating. It must provide information about the usage of the PT by the applications associated with it [MC21].
Req13	Servitization: the DT must enable the transformation of a product/artefact into a set of functionalities offered to users. This capability transforms products into software-controlled and connected entities accessible "on-demand" by users [MC21].
Req14	Predictability: the DT should have the ability to simulate its behaviour over time. Specific instances of DT should simulate the behaviour of the PT in a context at a specific time (in the past or in the future [MC21].
Req15	Data Link: the DT should have a Data Link to interconnect the DT with the PT and it acts as a hub for all information that is related to PT. It connects digital things to each other and leaves the DT connection for the coupling [OPV+21].
Req16	Coupling: the DT must have a two-way interface between the PT and the DT. Through the coupling, the PT delivers data to the DT or the DT may control the PT [OPV+21].
Req17	Data Storage: depending on the use case, the DT should/must have a suitable and appropriate database(s) [OPV+21].
Req18	(High-)Fidelity: the DT must have sufficient fidelity and (adequately) reflect the state and behaviour of the PT for the properties relevant to each of the DT's usages [OPV+21].
Req19	Multiplicity: the DT structure requires that the multiplicity of the relationship between the PT entities and the DT must be explicitly specified to understand what the entities in the PT that the DT is reasoning about and operating on, and how many DTs are present that obtain information on and influence the PT [OPV+21].

ReqID	ReqDescription
Req20	Security: depending on the use case, the security aspects should/must be embedded to the DT, with the appropriate level for each DT [OPV+21].
Req21	User Interface: the DT should/must have an appropriate and case-specific UI [OPV+21].
Req22	Life-cycle: depending on the use case, the DT should cope with the various stages/phases of the PT development and usage [OPV+21].
Req23	Composability: a DT should be structured modularly so that the system that can adapt over time as complexity increases. A composite DT is a system of assembled DTs that allows for scaling. A composable approach allows components and capabilities to be reused for building various applications tailored to specific purposes and end users [Dig22].
Req24	Extendability (in line with the definition in [EI90]): the DT should be structured and built in a way that facilitates modification with regard to extensions, etc.
Req25	Flexibility (in line with the definition in [EI90]): the DT should be structured and built in a manner such that the system and/or components can be modified.
Req26	Interoperability (in line with the definition of International Organization for Standardization [Int15]): the DT should have the capability of two or more functional units in the DT to process data cooperatively [Dig22].
Req27	A DT could provide some services in relation to optimization, decision making, task allocation, and so forth, coming for example from simulation, analysis, artificial intelligence, computation models, etc. [AVVT20].
Req28	The DT and PT should maintain a continuous, reliable connection to ensure instantaneous registration of any changes in status.
Req29	The DT should maintain continuous availability, ensuring its presence and operability before the PT's creation, during any connection losses to the PT, and after the PT has reached the end of its operational life.
Req30	The DT should continuously record and store all status changes and events of the PT, ensuring that historical data is available to represent the PT's status over time and independently of its/their location(s).
Req31	The DT should support the addition of new functionalities and capabilities through a data manager, enabling the execution of features that are not inherently available in the PT.
Req32	The DT should support the integration and correlation of various elementary DTs into complex systems, providing clear views of both the aggregated DT and its individual components.
Req33	The DT should provide comprehensive status and activity reports to enable effective management and optimization of its performance both through code modifications and user interface interactions within the operational framework.
Req34	The DT should correctly handle or react to changes in vehicle configurations, ensuring that the system behaviour adjusts appropriately within the defined context.
Req35	The vehicles' configuration and identification data must/should be transmitted and updated to the DT, depending on the use case.
Req36	Any simplifications or generalizations made in the DT model should not alter the behaviour of the vehicles on the track within the defined context.
Req37	The DT should support the addition and configuration of new vehicles with minimal effort while ensuring the representativeness of the PT in the specific context.

ReqID	ReqDescription
Req38	The DT should reflect timely (ideally real-time) changes and updates to the vehicles' hardware configurations accurately within the virtual environment.
Req39	The DT should accurately reflect any status changes and events occurring in the vehicles within the virtual environment.
Req40	Any modifications made to the DT should be replicated in the PT to ensure consistency between the virtual and physical systems.
Req41	Autonomous Driving basic function Perception: depending on the use case and complexity of the Autonomous Vehicle, and assuming that the relevant technology is available in the PT, the DT for Autonomous Driving must, in the interworking with the PT, i.e. vehicle(s), sense the surrounding environment of the Autonomous Vehicle using various types of sensor techniques such as RADAR, LIDAR and computer vision [KJD+15a].
Req42	Autonomous Driving basic function Localization: depending on the use case and complexity of the Autonomous Vehicle, and assuming that the relevant technology is available in the PT, the DT for Autonomous Driving must, in the interworking with the PT, i.e. vehicle(s), find the position of the Autonomous Vehicle using the techniques of a global positioning system, dead reckoning, and roadway maps [KJD+15a].
Req43	Autonomous Driving basic function Planing: depending on the use case and complexity of the Autonomous Vehicle, and assuming that the relevant technology is available in the PT, the DT for Autonomous Driving must, in the interworking with the PT, i.e. vehicle(s), determine the behaviour and motion of the Autonomous Vehicle based on the information from perception and localization [KJD+15a].
Req44	Autonomous Driving basic function Control: depending on the use case and complexity of the Autonomous Vehicle, and assuming that the relevant technology is available in the PT, the DT for Autonomous Driving must, in the interworking with the PT, i.e. vehicle(s), follow the desired command from the planning function by steering, accelerating, and braking the Autonomous Vehicle [KJD+15a].
Req45	Autonomous Driving basic function System Management: depending on the use case and complexity of the Autonomous Vehicle, and assuming that the relevant technology is available in the PT, the DT for Autonomous Driving must, in the interworking with the PT, i.e. vehicle(s), supervises the overall autonomous driving system (e.g. fault management system, logging system, and human-machine interface) [KJD+15a].

Table 4.1: Requirements

All the pertinent requirements, standards, regulations, etc. pertaining to Autonomous Driving (including Advanced Driver Assistance Systems, etc.) should/must apply to the DT. Given the time constraints of this thesis, these issues have been considered but not subjected to a comprehensive investigation. It is recommended that future work be conducted on this topic.

5 Concept

For the sake of clarity, it is imperative that the DT in this thesis aligns with the established definition of a Digital Twin, as outlined in Subsection 2.1.5. It is not merely a Digital Model (Req01) and more than Digital Shadow (Req02).
Based on the reviewed literature the author identified the following elements that a Digital Twin for Autonomous Driving should include:

(1) the duration of data exchange plays a crucial role. And therefore, the communication pathways should be short;

(2) a complete separation of concerns simplifies composability and flexibility;

(3) a unified communication structure within the Digital Twin helps to ensure that future components can be integrated with minimal additional effort (extendability);

(4) the Digital Twin should be adaptive to a certain degree, meaning that the respective components of the Digital Twin should be able to respond to an input with a specific reaction, driven by a certain logic

(5) and more as listed in the Chapter 4

It should be noted that this thesis does not elaborate on the conceptual approach or implementation of Autonomous Vehicles themselves, but rather focuses solely on the Digital Twin. Consequently, this thesis does not examine aspects such as protocols, elements, algorithms, standards or interfaces of Autonomous Driving. The concept presented in this thesis is for a Digital Twin of a more general nature and at a higher definition level.

The analysis of existing concepts and architectures (including frameworks) for Digital Twins, as discussed in the foundational Sections 2.2, 2.3, and 2.4 reveals that, to the best of the author's knowledge, no existing work meets the requirements. Consequently, this thesis presents in the following a potential conceptual structure for a composable (Req03), flexible (Req25), and extendable (Req24) Digital Twin, which meets the aforementioned considerations and the requirements presented in Chapter 4. It is firmly rooted based on the foundations presented in Chapter 2.

5.1 Key components and connections of a conceptual structure/architecture for a DT (answer to RQ1.2)

Fuller et al. [FFDB20] focus on enabling technologies and challenges for DTs, and present relevant components (or elements of the four enabling technology domains for I/IoT, namely: object, networking, middleware and application)
Sharma et al. [SKZ+20] present a consolidated overview (from different research and industrial works) of the basic components of a DT, which they categorise as elementary and imperative components. They also provide detailed descriptions of the respective requirements, including necessary connections and other pertinent information.

A Digital Twin must be meticulously structured and comprise the following elements:

- the appropriate components for communication and interconnection with the Physical Twin(s), taking into account for example whether both are located in close proximity or at a remote location. Furthermore, a comprehensive investigation must be conducted into the available interfacing and protocol capabilities at the PT level, along with an assessment of its limitations
- it is recommended that a data management layer and tool(s) be implemented as a gateway, in conjunction with the protocol broker, to facilitate the proper distribution of data and to assist with tasks such as data de-/serialization
- it is beneficial, if not imperative, to have a user interface (dashboard, cockpit, etc.) that is tailored to the specific objective and requirements
- all the components necessary to replicate virtually the Physical Twin with its functionalities, as well as to potentially enable simulation and other interesting application cases

5.2 Concept Elaboration

The following answers the research question, **RQ1.3** : What concept and meta-model framework could be elaborated for a Digital Twin in the context of Autonomous Driving, with specific consideration given to the flexibility, composability and extendability of the proposed approach?

The concept, presented as a meta-model and depicted in Figure 5.1, presents a Digital Twin consisting of separate entities, each with its own concern, characterized by multiple attributes and associations.

The design is intended to address key characteristics of composability (Req10, Req23) such as:

- modularity (parts of the system can function on their own and be replaced or upgraded without affecting the entire system)
- flexibility (can be adapted and reconfigured in order to have the ability to respond to new requirements or rearranging or adding new components)
- scalability (components can be scaled separately from one another without impacting other parts of the system)
- reusability (components can be repurposed by different applications or services)
- and that the components can be selected and assembled in various combinations to satisfy specific requirements

An entity can have a connection to the PT, user interface, database, etc., as and if required, via the data layer. Entities can communicate and interact with each other by sending data over directed connections. These directed connections are characterized by knowing their recipient and possessing various attributes. For instance, a communication message can be timed or triggered by a condition. Additionally, the directed connection can be synchronous, implying tight coupling, or asynchronous, implying loose coupling, which is appropriate for maintaining the composability of the Digital Twin. Each directed connection is based on a protocol such as MQTT or HTTP.

The data sent over the directed connection can serve various purposes, such as commands or requests, but can also include simple information like status messages, confirmations, or responses to requests.

Derived from [WEB+21], each entity can have one to three roles: `descriptive`, `predictive` and/or `prescriptive`. These roles broadly describe the functionality of an entity to better understand its purpose and concern.

Additionally, an entity can possess, derived from [WBD+20], one to four functionalities: `DataProcessor`, `Evaluator`, `Reasoner` and/or `Executor`. Since the format of messages sent internally within the Digital Twin needs to be uniform, data arriving in an entity is not necessarily directly usable. The `DataProcessor` functionality refers to the entity's ability to process the data in a way that makes it usable within the entity. In order to maintain generality, this concept does not refine `DataProcessor` into Data Processor Logic and Data Processor Adapter as in [WBD+20], as the required steps for `DataProcessor` can vary from use case to use case, and this concept does not restrict itself to CPPS. Therefore, the internal processing of `DataProcessor` is not elaborated for the sake of generality.

Messages pertinent to a specific entity typically contain information, data, updates, and often require corresponding actions. Evaluator refers to the entity's ability to evaluate received data.

The data is evaluated, following which a reaction or processing is required. The subsequent action is determined by the intelligence of an entity, referred to as the Reasoner. There are multiple potential approaches to reasoning. In a manual system, a user receives the necessary evaluated information and subsequently specifies the appropriate consequence or action. A step towards greater autonomy would involve a predefined system where reasoning is conditionally based on pre-specified possibilities for evaluation results. For full autonomy, the reasoning would be based on a self-adapting system that can operate using, for example, Base-Based-Reasoning (CBR) (see [WBB+21]) or AI (deep learning, etc.).

For the reasoning process, it may also be necessary to use additional knowledge beyond what is provided by a message. Accordingly, each entity incorporates a KnowledgeBase, which may take the form of access to a proper or shared database, or the local runtime memory of the entity in question.

Finally, the Executor delineates the functionality of an entity responsible for executing the actions determined by the Reasoner. Similar to the DataProcessor, this concept maintains generality by not subdividing the Executor into Execution Logic and Execution Adapter, as outlined in [WBD+20].

For the concept in this thesis, it is assumed that the aforementioned four functionalities (namely DataProcessor, Evaluator, Reasoner and Executor) may, but are not required to, reside within a single entity. This is contingent upon the specific use case of the Digital Twin. It is also assumed that for establishing a feedback loop, the three following functionalities Evaluator, Reasoner, and Executor, are present for each entity, either all in one entity or one functionality used as a service or separate entity by the entity, to ensure feedback. This establishes an automatic two-way data flow, leading to the conclusion that this concept, according to [KKT+18], indeed characterizes a Digital Twin, in comparison for example with shadowing only.

Since each entity has its own scope and necessity by being separate from the other entities, each entity can be seen as a Service that might need or use another entity as a Service as well.

Thus, an entity can provide multiple services, including, for example, User Interface, Performance Indication, Monitoring, Simulation and Analytics.

Notably, this concept considers all five dimensions (Physical Entity, Virtual Entity, Data, Connection, Service) as outlined by [TLZ+19] and presented in the foundational Subsection 2.3.3.

Moreover, an entity may depend on another entity or multiple entities as required to ensure its correct functioning.

To ensure that the entire Digital Twin operates correctly, each entity is further defined by its non-functional requirements, which could include aspects such as for example scalability, performance, reliability, availability, interoperability, security, safety, etc. which are not further investigated in this thesis.

Moreover, the status of an active Digital Twin is contingent upon the active status of its constituent entities. Therefore, each entity has a running state, which can be either Running or Not Running, indicating whether an entity is active or available within the Digital Twin.

The class diagram presented in Figure 5.1 outlines the structure and relationships between various entities/components within the composable (Req10, Req23), flexible (Req25) and extendable (Req24, Req37) Digital Twin.

This comprehensive structure ensures a flexible system capable of adapting to various operational requirements. The enumerations presented here and in Figure 5.1 are provided for illustrative purposes only and do not represent a comprehensive or exhaustive list. Additionally, the relevance of these enumerations may vary depending on the specific context.

The content and boundaries of the data layer, as well as its connections, are defined in a subsequent model and/or architecture.

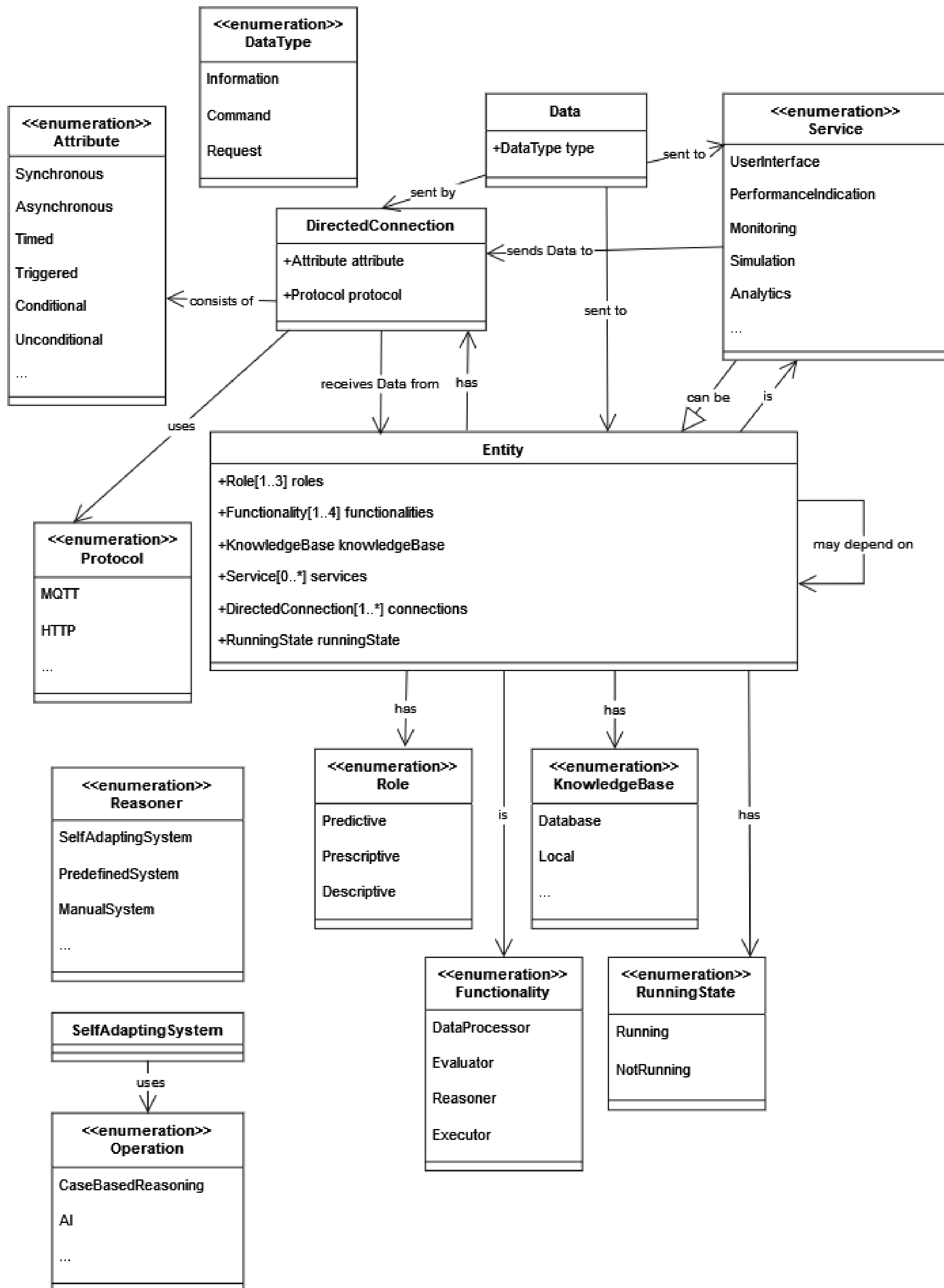

Figure 5.1: Meta-model for a composable, flexible and extendable DT

5.3 Use Case Description and Architecture

RQ1.4 : Which Use Case can be defined and which architectural framework can be derived from the concept and its meta-model of a Digital Twin for Autonomous Driving, especially with regard to flexibility, composability and expandability?

Autonomous Driving encompasses a broad spectrum of use cases, for which Digital Twin(s) could be elaborated based on the above mentioned concept.

Derived from the meta-model concept from Section 5.2 and shown in Figure 5.1, this section details the relevant components for the Digital Twin (Req03) of an Autonomous Driving Use Case, which serves as an instance of the meta-model, i.e. a model. These components are described and elaborated herein and are depicted in Figure 5.2.

This Use Case is for a Digital Twin for Autonomous Driving (specifically for the ADAS capabilities achievable with the available Robot Cars) for multiple vehicles all contained within a single Physical Twin, and considering its specific context (Req04). To determine which elements are relevant and practical for Autonomous Driving, knowledge primarily from foundational Sections 2.4 and 2.5 has been utilized.

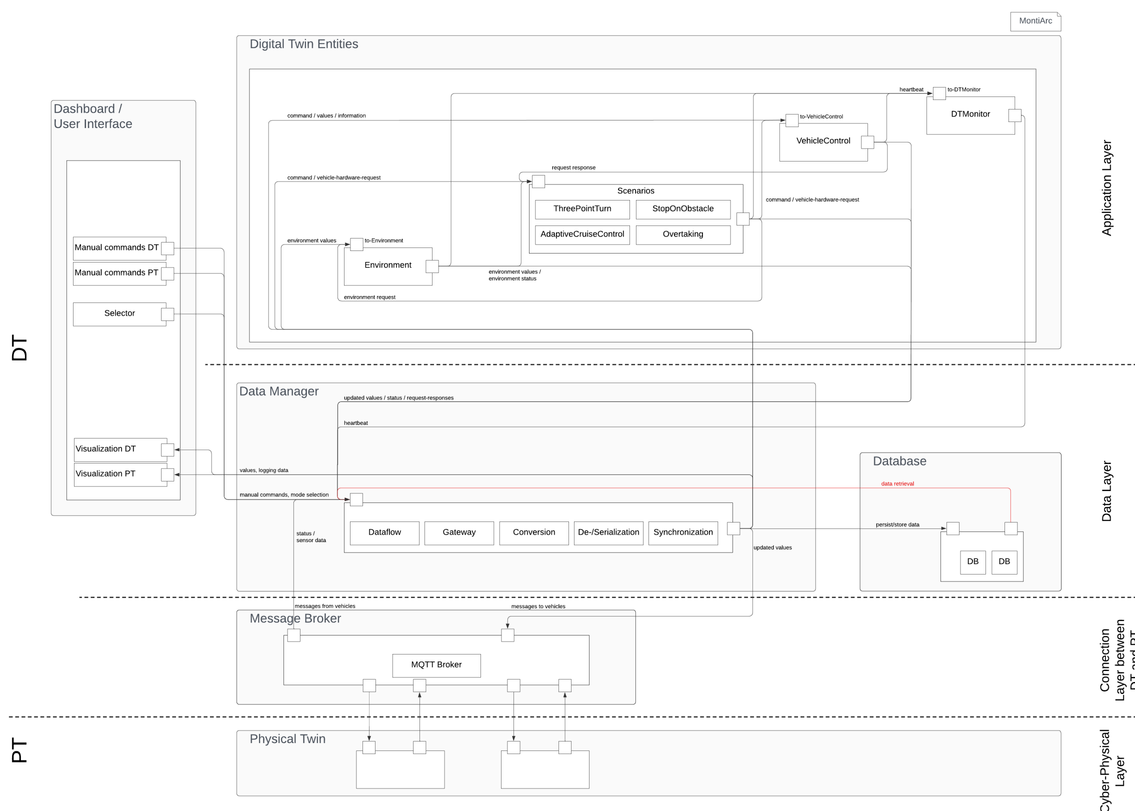

Figure 5.2: The architecture for a DT for Autonomous Driving

5.3.1 Different Denominations of Entities in the Use Case, Implementation, Code

For clarification purposes it is important to note that specific denominations for the Entities are used to differentiate between the Use Case, the implementation description and the code:

- in the use case:
  - the Entities belong to the Digital Twin Entities frame of the application as illustrated in Figure 5.2
  - the Entities of general nature are denominated with their respective names, followed by "Entity", e.g. "VehicleControl Entity"
  - the scenario relevant Entities are denominated with their respective names, followed by "Entity-Scenario", e.g. "AdaptiveCruiseControl Entity-Scenario"
- in the implementation description:
  - the applications of general nature are denominated with "DT-Entity", followed by their respective names, e.g. "DT-Entity VehicleControl"
  - the scenario applications are denominated with "DT-Entity-Scenario", followed by their respective names, e.g. "DT-Entity-Scenario AdaptiveCruiseControl"
- in the code:
  - the Java applications are denominated with their respective names, e.g. "VehicleControl"
  - in the Java code no different demonination is used for general and scenario oriented Entities

5.3.2 Entities

The Entities described herein are consistent with the components outlined in the concept from the preceding Section 5.2 and its meta-model diagram (Figure 5.1).

VehicleControl Entity

To ensure that the Digital Twin can describe multiple vehicles, all Entities are designed to be available for several vehicles in parallel, meaning each Entity exists only once, regardless of the number of vehicles. This Use Case includes the virtual replication of the relevant vehicles' hardware components, which collectively act as the Physical Twin, within the Digital Twin.
This thesis assumes that every vehicle, in the context of Autonomous Driving, at least meets the requirements for basic driving capabilities, which are steering, forward/backward locomotion, acceleration and braking. In order to streamline the communication pathways in the Digital Twin representation, both, instantiation and basic driving capabilities, are incorporated within a single Entity. This Entity, inspired by [KJD+15b] and referred to as "VehicleControl", is responsible for modifying and setting the basic driving capabilities and defining the values and characteristics of the relevant actuators, sensors, etc. of the vehicles (Req03, Req18, Req38, Req39, Req40, Req44). VehicleControl stores the state of the vehicles, i.e. changes in vehicle information, values

and relevant commands go into this Entity (Req05, Req08, Req11, Req18, Req34, Req38, Req39, Req40). All the commands and values from the scenario Entities to the vehicles and/or Database and/or Dashboard go through this Entity. This makes it the core component of the Digital Twin.

According to [WEB+21], presented in Subsection 2.2.1, VehicleControl fulfils in this thesis mainly two roles. It fulfils a `descriptive` role by storing and monitoring the information and state of each vehicle, thereby providing a current status overview. In this thesis it also comprises a vehicle configurator (including the virtual vehicle creator and identifier, detailing its elements e.g. actuators, sensors, etc. and their respective characteristics).

Additionally, it serves a `prescriptive` role in this thesis by issuing commands to adjust and set the vehicle's behaviour and other properties, thereby guiding vehicle actions, based on the respective scenario.

Following [WBD+20], presented in Subsection 2.3.1, VehicleControl operates as both, a `DataProcessor` and `Executor`. The decoding of incoming messages, to understand the data related to the vehicle's state and operational parameters, categorizes it as a `DataProcessor`. The ability to execute commands contained within the decoded messages to adjust the vehicle's behaviour, such as steering, acceleration, and braking, makes it an `Executor`.

VehicleControl offers continuous monitoring and control of vehicle parameters, managing the vehicles' behaviour, making it a vital `Service` (e.g. as Vehicle Management Service) for the overall system's operation, as stated in [TLZ+19] and presented in Subsection 2.3.3.

Deeper aspects of Autonomous Driving and vehicle properties, such as driving safety, robustness under various driving conditions, nonholonomic constraints, vehicle dynamics, and the tradeoff between tracking performance and ride comfort [KJD+15b], are not considered in the Use Case, due to the bounded scope of this thesis. However, these elements can still be represented in a DT, using the presented concept.

Environment Entity

Environmental conditions play a crucial role. These factors include weather and visibility conditions (e.g. fog, rain), as well as vehicle stability and susceptibility to side winds. Road surface and geometry are also relevant factors as well as others such as road condition (e.g. ice, snow, potholes). The "Environment" Entity monitors these by collecting and storing data from the physical Entity(ies), making this information accessible for further analysis (Req04, Req05, Req18, Req43).

According to [WEB+21], presented in Subsection 2.2.1, Environment fulfils in this thesis mainly one role, possibly two. It serves a `descriptive` role by collecting and storing environmental data from the Physical Twin, providing an accurate picture of current conditions. In future extensions, it could also fulfil a `predictive` role by assessing the potential impacts of environmental changes, such as weather or road conditions, on vehicle performance. This `predictive` function, however, is not yet included in the current Use Case.

Following [WBD+20], presented in Subsection 2.3.1, Environment operates as a `DataProcessor` for incoming messages. It collects and stores environmental data from the physical Entity(ies). In future extensions, it could be expanded to evaluate current environmental conditions based on the processed data in order to inform other Entities, which would make it also an `Evaluator`. This evaluation capability has not yet been implemented in the current Use Case.

Environment offers timely critical environmental data, which other Entities rely on to function effectively, enhancing decision-making and adaptability, making it a `Service` (e.g. as Environmental Data Service), as stated in [TLZ+19] and presented in Subsection 2.3.3.

Introduction to the following scenario Entities:

Beyond environmental conditions, numerous other factors must be considered in Autonomous Driving (Req41, Req42, Req43, Req44, Req45). These include road markings (for lane recognition and lane changes), obstacles (requiring evasive manoeuvres to avoid accidents and maintain safety distances), the speed of other vehicles (to adjust the vehicle's own speed and avoid collisions), and traffic density (affecting traffic flow and congestion risks). Construction zones (with changes in the road layout and temporary traffic management), traffic signs (governing traffic rules and speed limits), and infrastructure elements (such as traffic lights and roundabouts) also play significant roles. Additionally, pedestrian density impacts reaction times and stop-and-go movements.

Within the constraints of the given Physical Twin, as introduced in Section 6.2, and the bounded scope of this thesis, the following scenarios have been thoroughly examined and implemented. In building upon existing functionalities of the physical vehicles, and in order to ensure the composable nature of the entire system (Req10, Req23), as per the introduced concept, the solution approach is sequential, with scenarios designed to build on previous ones.

ThreePointTurn Entity-Scenario

The "ThreePointTurn" scenario outlines a vehicle executing a basic three-point turn using fixed steering and locomotion without relying on sensor data for the manoeuvre itself. Once initiated, this scenario follows a hardcoded sequence of time-triggered commands, such as setting specific speeds and steering angles for designated intervals, unless a predefined trigger, such as a sensor detecting an obstacle in the vehicle's path, interrupts and cancels the procedure.

According to [WEB+21], presented in Subsection 2.2.1, ThreePointTurn serves in this thesis a `prescriptive` role by following a hardcoded sequence of commands to execute the manoeuvre, instructing the vehicle on specific actions at designated intervals.

Following [WBD+20], presented in Subsection 2.3.1, ThreePointTurn operates as an `Executor`, performing the three-point turn based on the decoded instructions in the messages.

The ThreePointTurn Entity does not act continuously as a `Service` but rather executes a predefined manoeuvre when triggered, and thus does not fit the continuous `Service` model, as outlined in [TLZ+19] and presented in Subsection 2.3.3. (Req03, Req05, Req18, Req39, Req40)

StopOnObstacle Entity-Scenario

The "StopOnObstacle" scenario describes the process by which a vehicle halts its movement upon detecting an obstacle ahead, based on sensor measurements of the front distance. The threshold distance at which the vehicle reacts is adjustable and not fixed. A notable aspect of this scenario is that, once the vehicle has stopped due to the threshold being set higher than the measured front distance value, it will not resume motion autonomously even after the obstacle has moved out of range. Restarting the vehicle's movement must be explicitly initiated either by a new scenario or manually, either in the DT or PT.

According to [WEB+21], presented in Subsection 2.2.1, StopOnObstacle serves a `prescriptive` role by dictating vehicle behaviour when an obstacle is detected, instructing the vehicle to halt and defining the conditions under which it will resume motion.

Following [WBD+20], presented in Subsection 2.3.1, StopOnObstacle functions in this thesis both, as an `Evaluator` and `Executor`. The assessment of the distance to an obstacle and the decision on whether the vehicle should stop, based on the information decoded from the messages, categorize it as an `Evaluator`. Meanwhile, executing the stop command and managing the conditions for resuming movement, as dictated by the decoded messages, establishes it as an `Executor`.

StopOnObstacle executes specific actions based on sensor data but does not offer continuous functionality that would classify it as a `Service`, as outlined in [TLZ+19] and presented in Subsection 2.3.3 (Req03, Req05, Req18, Req39, Req40).

AdaptiveCruiseControl Entity-Scenario

Traditional Cruise Control, also known as Cruise control, is a system that automatically controls the speed of a vehicle by maintaining a set speed as determined by the driver, regardless of the gradient of the road [LK21].

Adaptive Cruise Control (ACC) is an advanced version of Cruise Control that not only maintains a set speed but adjusts the vehicle's speed to maintain a safe distance from vehicles and obstacles ahead, using RADAR or other sensors to monitor the traffic conditions [LK21] (Req03, Req05, Req18, Req39, Req40).

The "AdaptiveCruiseControl" scenario describes the continuous application of ACC, as defined previously. For clarity within this thesis, the vehicle executing the AdaptiveCruiseControl scenario is referred to as the "following vehicle", a term that effectively conveys its role.

In the event that the lead vehicle reverses, there are two potential ACC response strategies: the following vehicle could either stop completely, even if the lead vehicle continues to reverse and potentially collides with it, or the following vehicle could also reverse once the lead vehicle crosses a certain threshold distance. Alternatively, an obstacle avoidance functionality could also be considered. The appropriate response strategy depends on the characteristics of the Physical Twin and varies case by case.

For ACC, inclines and minor surface irregularities can pose challenges, depending on both the software implementation how effectively these conditions are detected and managed and the hardware capabilities of the vehicles, including the strength and precision of their motors.

According to [WEB+21], presented in Subsection 2.2.1, AdaptiveCruiseControl fulfils in this thesis mainly two roles. It serves a `predictive` role by adjusting the vehicle's speed based on timely (ideally real-time) sensor data to maintain a safe following distance. Additionally, it fulfils a `prescriptive` role by maintaining a set speed and adapting it according to the vehicle or obstacle (e.g. pedestrian, etc.) in front, ensuring safe and efficient travel.

Following [WBD+20], presented in Subsection 2.3.1, AdaptiveCruiseControl operates as an `Evaluator`, a `Reasoner`, and an `Executor`. It functions as an `Evaluator` by continuously monitoring the distance to other vehicles and adjusting speed accordingly based on received information. It acts as a `Reasoner` by interpreting sensor and environmental data to make decisions on speed adjustments, using predefined systems or cases as a reasoning base.

Finally, it serves as an `Executor` by controlling the vehicle's speed based on evaluations and reasoning outcomes, executing the received commands. AdaptiveCruiseControl provides ongoing speed adjustments to maintain safe distances, which other Entities and processes rely on. Its continuous nature and critical role in vehicle safety and efficiency make it a key `Service` (e.g. as Speed and Distance Regulation Service), as stated in [TLZ+19] and presented in Subsection 2.3.3.

Overtaking Entity-Scenario

The "Overtaking" scenario, inspired by the cooperative overtaking scenario presented in [Stü23], describes two autonomously operating vehicles traveling in the same direction on a single lane road track. To maintain clarity in this thesis, the vehicle being overtaken is referred to as the "lead vehicle", and the vehicle performing the overtaking is referred to as the "overtaking vehicle". These terms are straightforward and effectively convey the roles of the two vehicles in the scenario.

In this thesis, the road track offers at some point a bifurcation to a shorter route that merges back to the main road track after a certain distance. The overtaking vehicle uses this bifurcation to overtake the lead vehicle.

The aim of this thesis is to demonstrate the value of a Digital Twin by incorporating the procedural logic of the overtaking scenario into the Digital Twin, as opposed to depending on vehicle cooperation for overtaking. Thus, the cooperative overtaking scenario presented in [Stü23] has been adapted to an overtaking scenario without inter-vehicle communication.

According to [WEB+21], presented in Subsection 2.2.1, Overtaking fulfils in this thesis mainly two roles. It assumes a `predictive` role by assessing the relative speed and position of the vehicles, predicting the optimal moment to initiate the overtaking manoeuvre. Additionally, it fulfils a `prescriptive` role by commanding the overtaking vehicle to accelerate and navigate the bifurcation safely, based on the scenario logic.

Following [WBD+20], presented in Subsection 2.3.1, Overtaking functions as an `Evaluator`, a `Reasoner`, and an `Executor`. It acts as an `Evaluator` by assessing the relative speeds and positions of the vehicles involved in the overtaking manoeuvre, based on the provided information. As a `Reasoner`, it determines the best moment and method to overtake the lead vehicle, using informational requests and predefined systems or cases as a reasoning base (Req03, Req05, Req18, Req39, Req40).

Finally, it serves as an `Executor` by carrying out the necessary commands to perform the overtaking manoeuvre safely. Overtaking offers a complex, coordinated function necessary for safe and effective overtaking manoeuvres. By managing this, it provides an essential `Service` to the driving system's overall operational strategy, making it a key `Service`, (e.g. as Overtaking Coordination Service), as stated in [TLZ+19] and presented in Subsection 2.3.3.

In a real-world overtaking scenario, multiple critical factors and decisions must be considered. A key consideration is the relative speed between the lead and overtaking vehicles. For example, if the lead vehicle accelerates to match the speed of the overtaking vehicle, it may render the overtaking manoeuvre unnecessary or complicate it if the acceleration occurs during the overtaking attempt. Traffic density is another vital factor, as the presence of other vehicles on one or both lanes, or multiple vehicles attempting to overtake consecutively, can significantly impact the manoeuvre. Additionally, various road types, such as multi lane highways, must be considered, especially when vehicles travel in opposite directions. The condition and quality of the road are equally important. Factors such as road damage, unsuitable materials, visibility issues, or insufficient lane width can affect the safety and feasibility of overtaking. Moreover, weather conditions, time of day, and overall visibility of the road ahead are essential considerations. Adherence to traffic laws, including overtaking restrictions and speed limits, is imperative. Maps and navigation systems can provide valuable information on speed limits, the number and length of fast lanes, and other relevant details. The behaviour of other vehicles, encompassing their actions and reactions, must be anticipated and managed. The performance capabilities of the overtaking vehicle, such as acceleration and braking, are critical. Sensor data from RADAR and cameras aid in detecting obstacles and other vehicles, enhancing the safety of the manoeuvre. By taking many to all relevant factors into account and leveraging advanced technologies, safer and more effective overtaking manoeuvres can be achieved through the extensive testing and refinement of simulated real-world driving conditions in the lab using Digital Twins. However, these extended factors are not considered in the specific example introduced here. The more precise procedure depends on the capabilities of both the Digital Twin and the Physical Twin vehicles and will be explained in greater detail in Subsection 6.4.15.

DTMonitor Entity

Some, but not all, of these scenarios can be active simultaneously. The correct interaction of multiple Entities is crucial, as they partially depend on being active at the same time. However, due to their loose coupling, this is not always the default state.

The "DTMonitor" Entity is required in order to periodically check which Entities are active and react accordingly in a predefined manner (Req12, Req18, Req25, Req28, Req29). The DTMonitor performs anomaly detection by monitoring for any irregularities or deviations from expected performance and triggering simple alerts when necessary.

According to [WEB+21], presented in Subsection 2.2.1, DTMonitor fulfils in this thesis two roles. It serves a `descriptive` role by monitoring the active status of all Entities, ensuring they function correctly. Additionally, it assumes a prescriptive role by reacting to Entity status changes with predefined actions to maintain a certain of integrity and performance in coordination with the Data Manager. In future extensions, it could also take on a `predictive` role by detecting anomalies and

anticipating issues that may arise, prompting preventive measures, etc. in order to maintain a more sophisticated level of system integrity and performance. However, this `predictive` function is not yet included in the current Use Case.

Following [WBD+20], presented in Subsection 2.3.1, DTMonitor operates as an `Evaluator`, a `Reasoner`, and an `Executor`. It functions as an `Evaluator` by monitoring the active status of all Entities and detecting anomalies through the evaluation of processed messages. It acts as a `Reasoner` by interpreting the data from these messages to predict potential issues and necessary adjustments, using predefined systems or cases as a reasoning base.

Finally, it serves as an `Executor` by taking predefined actions based on evaluations and reasoning processes, executing and sending the relevant messages. It monitors the health and synchronized functioning of the entire system by monitoring active Entities and reacting to their status, thereby supporting the operational integrity of the Digital Twin. This oversight and management role aligns with the `Service` component (e.g. as System Health Management Service), as outlined in [TLZ+19] and presented in Subsection 2.3.3.

To conclude, in larger use cases, the DTMonitor Entity could evaluate the performance of the PT based on collected data to identify areas for improvement, use historical data and `predictive` algorithms for proactive maintenance scheduling, run simulations to test various scenarios and their impacts on the PT using the DT as a controlled environment, and generate alerts and detailed reports on the performance and condition of the PT. However, these extended functionalities are beyond the scope of the current Use Case.

In the following, tables are presented to provide a comprehensive overview of presented Entities: Table 5.1 illustrates their role classification as per [WEB+21], Table 5.2 outlines their functionality classification according to [WBD+20], and Table 5.3 details their `Service` classification based on [TLZ+19].

Entity \ Role	`Descriptive`	`Predictive`	`Prescriptive`
VehicleControl	✓	X	✓
Environment	✓	possible	X
ThreePointTurn	X	X	✓
StopOnObstacle	X	X	✓
AdaptiveCruiseControl	X	✓	✓
Overtaking	X	✓	✓
DTMonitor	✓	possible	✓

Table 5.1: Overview of the Entities with their role classification (`Descriptive`, `Predictive`, `Prescriptive`) according to [WEB+21].

Entity \ Functionality	`DataProcessor`	`Evaluator`	`Reasoner`	`Executor`
VehicleControl	✓	X	X	✓
Environment	✓	possible	X	X
ThreePointTurn	X	X	X	✓
StopOnObstacle	X	✓	X	✓
AdaptiveCruiseControl	X	✓	✓	✓
Overtaking	X	✓	✓	✓
DTMonitor	X	✓	✓	✓

Table 5.2: Overview of the Entities with their functionality classification (`DataProcessor`, `Evaluator`, `Reasoner`, `Executor`) according to [WBD+20].

Entity \ `Service`	`Service` Classification	Possible `Service` Name
VehicleControl	✓	Vehicle Management Service
Environment	✓	Environmental Data Service
ThreePointTurn	X	-
StopOnObstacle	X	-
AdaptiveCruiseControl	✓	Speed and Distance Regulation Service
Overtaking	✓	Overtaking Coordination Service
DTMonitor	✓	System Health Management Service

Table 5.3: Overview of the Entities and their service classification based on [TLZ+19].

Comments pertaining `Reasoner`, KnowledgeBase, RunningState

Due to the bounded scope and the relatively simple nature of the Physical Twin, which will be examined in detail in Section 6.2, the `Reasoner` for all Entities operates using predefined cases, consisting of numerous if-else conditions that guide decision-making.

Additionally, all Entities utilize the local runtime memory as their KnowledgeBase, i.e. the local memory within each Entity. While changes in the VehicleControl Entity are persistently stored in a Database, this information is not retrieved for reasons described further below. Consequently, the Database cannot be considered as a knowledge base for the presented Entities.

All Entities are in a RunningState, as required, for this Use Case.

5.3.3 Inter-Entities communication

The Entities presented must be capable of inter-Entity communication. This communication is facilitated through directed asynchronous connections, which may be timed, triggered, conditional, or unconditional, depending on the specific message being communicated. MQTT is utilized as the protocol for these interactions.

Each scenario stores relevant information during runtime, correlated to the respective vehicle it concerns. Consequently, each scenario can both request information from VehicleControl regarding specific vehicles and receive responses, as well as issue commands to alter the values of individual vehicles, thereby changing the status of the Digital Twin (Req03, Req05, Req06, Req08, Req11, Req18,Req34,Req38, Req39, Req43, Req44).

As the AdaptiveCruiseControl Entity-Scenario relies heavily on environmental factors, it sends additionally a request to the Environment Entity, which supplies the relevant information in return.

All Entities send a heartbeat signal as a message to the DTMonitor Entity. To ensure the effective and synchronized functioning of all Entities, the DTMonitor Entity regularly checks which Entities are active and send those heartbeats accordingly, as described further in Subsection 6.4.10.

This comprehensive setup enables communication, data exchange and system monitoring, laying a foundation for the Digital Twin's performance and reliability (Req04, Req05, Req06, Req18,Req34,Req38, Req39, Req43, Req44).

The implementation details concerning the Entities, their directed connections, and the data exchanged between them are thoroughly discussed in Section 6.4.

5.3.4 Additional elements

A Data Manager is essential to facilitate communication and interaction between the DT and the PT as well as with the User Interface and Database. This manager acts as a gateway, managing data flows and can be used for conversions, de-/serialization, synchronization, etc. (Req19). It can also be created as an Entity as provided by the presented concept (Req06, Req15, Req16, Req28,Req31,Req38, Req39, Req40).

Externally, the system requires several components: A Physical Twin, which interacts interactively with the DT, and a Message Broker (Req15, Req16) to enable communication and coordination between the two systems. Additionally, a User Interface (Dashboard) is needed to manually control and visualise the states of both the DT and PT, as well as to switch operational modes via selector (shadowing, emulation, simulation, testing, etc.) (Req04, Req05, Req12, Req18, Req21, Req39). The utilization of a database is beneficial for ensuring and maintaining persistent storage of the states of the DT and PT, as well as for the purpose of logging data (Req07, Req08, Req17, Req30).

These components work together to create an integrated system that leverages the capabilities of the DT and PT.

Communication

All Entities send messages to the Data Manager. The Data Manager processes this data and sends it to the appropriate recipients: the Dashboard, Physical Twin and/or Database. Consequently, the Dashboard and Physical Twin receive updated values, while the Database stores data that requires persistent storage (Req08, Req17, Req30).

Furthermore, the Data Manager is responsible for disseminating information such as new status updates, new data (e.g., sensor data), requests, or commands from the Dashboard or Physical Twin to the relevant Entities (Req15, Req16,Req19, Req28, Req38, Req39, Req40). In the event of a change, the Data Manager provides the necessary environmental information to the Environment Entity. Additionally, the Data Manager issues commands and requests to the scenarios ThreePointTurn, StopOnObstacle, AdaptiveCruiseControl, and Overtaking, as well as to VehicleControl.

Given the manageable size of this Digital Twin prototype's data, direct reading from runtime memory proves faster and more efficient than database retrieval. Consequently, only the database storage functionality was implemented, omitting the retrieval aspect for better performance. Nevertheless, adjustments may be required for future expansions of the DT. Thereby, extending this functionality is a straightforward task and thus already presented in as a red connection (labelled: data retrieval) in Figure 5.2.

For larger Digital Twin systems, it is advisable to store raw Physical Twin data in a database for persitence and also to enable extensive data analysis, historical trend tracking, and predictive adaptation and e.g. predictive maintenance, thereby enhancing overall system performance and decision-making (Req14). However, in this instance, direct access from runtime memory was deemed sufficient for the correct functioning of the Digital Twin prototype, given the project's limited scope. This decision ensures efficiency and simplicity, without compromising the system's effectiveness. Nonetheless, for future expansions of the DT, adjustments may be required, but extending this functionality is still a straightforward task.

6 Implementation of a Digital Twin for Autonomous Driving

RQ1.5 : What prototype implementation and with which solution approach could supplement the Use Case?

The elaborated concept introduced in Chapter 5 is supplemented with a prototype DT (Req03) implementation for Autonomous Driving, based on an existing physical entity, henceforth also referred to as the Physical Twin. The integration of a DT aims to enhance the understanding and functionality of autonomous systems by creating a virtual counterpart that mirrors the physical counterpart timely (ideally in real-time).

To achieve this, it is first necessary to comprehend the construct and design of the physical entity, identifying its functionalities and limitations. This foundational understanding is crucial as it informs the subsequent development of the DT. The detailed analysis of the PT will be addressed in the following Section 6.2 after discussing the essential fundamentals required for technical understanding.

Upon gathering sufficient information on the PT, the next step involves selecting the appropriate tools and languages for the development of the DT, as well as for interfacing and database management. The criteria and rationale behind the selection of these tools, along with the decision-making process, will be thoroughly discussed in Section 6.3.

Section 6.4 delves into the comprehensive details of the DT, outlining its components, architecture, and the integration process with the PT. This section will provide an in-depth exploration of how the DT replicates the PT's functionalities and the enhancements it brings to it.

In Section 6.5 a series of illustrative scenarios of the prototype Digital Twin are presented.

6.1 Essential fundamentals

This section presents the essential fundamentals necessary for a comprehensive technical understanding of the PT and DT.

6.1.1 Publish/Subscribe (Pub/Sub)

The Publish/Subscribe (Pub/Sub) model is a messaging pattern where message producers (publishers) send messages to specific topics. Message consumers (subscribers) receive messages from these topics. This model supports asynchronous communication and loose coupling between publishers and subscribers, as they do not need to be aware of each other's existence. In a Pub/Sub-system, a central broker manages the distribution of messages. Publishers send messages to the broker, which then forwards the messages to all subscribers of the relevant topics. This decoupling enhances scalability and flexibility (Req25) in distributed systems, making the Pub/Sub-model suitable for applications such as Internet of Things (IoT), where devices may intermittently connect and disconnect from the network.

6.1.2 Message Queuing Telemetry Transport (MQTT)

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. It follows the Pub/Sub messaging pattern, as presented in Subsection 6.1.1, which allows for loose coupling between message producers (publishers) and message consumers (subscribers). This decoupling enhances scalability and flexibility in distributed systems. In MQTT, a central broker manages the distribution of messages. Publishers send messages to specific topics, and subscribers receive messages from these topics. This model supports asynchronous communication, making it suitable for applications such as IoT, where devices may intermittently connect and disconnect from the network. MQTT messages consist of several components, including a fixed header, variable header, and payload. The fixed header is present in all MQTT messages and contains information such as the message type and flags. The variable header is optional and may include additional information such as the topic name and packet identifier. The payload contains the actual message content and can vary in size depending on the application requirements. The structure of MQTT messages allows for efficient and flexible communication, making it suitable for a wide range of applications.

6.1.3 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a lightweight data-interchange format that is both easy for humans to read and write, and easy for machines to parse and generate. JSON is built on two primary structures: a collection of name/value pairs, which is often realized as an object, record, struct, dictionary, hash table, keyed list, or associative array, and an ordered list of values, which is often realized as an array, vector, list, or sequence [Cro06].
JSON's syntax is derived from JavaScript Object Notation, making it intuitive and easy to understand. Its simplicity allows for quick learning and implementation, while its language independence ensures that it can be used across various programming languages, including Java, Python, and C++. This

makes JSON a versatile choice for data interchange in diverse computing environments.
In the context of messaging protocols such as MQTT (see Subsection 6.1.2), JSON is frequently used to encapsulate the payload of messages. This encapsulation allows for the inclusion of structured information such as the sender, message ID, correlation ID, and other metadata. The use of JSON in MQTT messaging enhances the clarity and structure of the transmitted data, facilitating better communication between distributed applications.
Adopting JSON as a data interchange format in messaging protocols like MQTT provides several advantages. JSON's lightweight nature ensures minimal overhead, making it suitable for low-bandwidth and high-latency environments typical of IoT applications. Additionally, JSON's ability to represent complex Data Structures allows for including detailed metadata, which can be crucial for message routing, correlation, and processing. Furthermore, the widespread adoption of JSON across various platforms and languages ensures seamless data exchange between systems, regardless of the programming language or platform.
For Java, there are currently several fast JSON libraries available which provide efficient serialization and deserialization of JSON data. These libraries are designed to handle large volumes of data quickly and efficiently, making them suitable for high-performance applications. A systematic analysis and comparison of 20 different JSON libraries in Java highlights their performance and efficiency [Pal+21].

6.1.4 Last Will and Testament (LWT)

The Last Will and Testament (LWT) feature in MQTT (see Subsection 6.1.2) is a powerful tool that allows clients to specify a message that will be automatically published by the broker on their behalf if an unexpected disconnection occurs. This ensures that other clients are notified of the disconnection and can take appropriate action.
When a client connects to the broker, it specifies a last-will message. This message includes a topic, payload, Quality of Service (QoS), and a retained flag. The broker stores this message until it detects an ungraceful disconnect from the client. Upon detecting an ungraceful disconnect, the broker broadcasts the last-will message to all subscribed clients of the corresponding topic. If the client disconnects gracefully using the DISCONNECT message, the broker discards the stored LWT message.
The LWT feature is particularly useful in scenarios where it is important to detect and respond to unexpected disconnections, such as in IoT applications. The LWT feature also works with org.eclipse.paho.mqttv5, providing compatibility with various MQTT implementations.

6.1.5 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a protocol that allows a program to request a service from a program located on another computer within a network, without needing to understand the network's details. This abstraction simplifies the development of distributed systems by enabling communication between different systems as if they were local procedure calls. While MQTT is primarily a Pub/Sub system, as discussed in Subsection 6.1.1, and does not inherently support RPC, it can be adapted to fit RPC-style interactions. For instance, a request/response interaction can be implemented over MQTT by publishing the request on one topic and listening for the response on another. This method essentially mimics an RPC interaction.

6.2 Physical Twin

The implementation of the DT in this thesis is a prototype, utilizing the readily available Arduino Robot Cars (of the Robo-Laboratory of the Software Quality and Architecture Group, at the Institute of Software Engineering, University of Stuttgart[1]) as PT, without any major modifications to the hardware and software. It will focus only on basic scenarios, with a shadowing and replication of the relevant functions only. It will not include nor replicate the mechanical parts nor the sensors or actuators as such, but only their functions and parameters. The Arduino Robot Car differs significantly from conventional vehicles in a number of ways. It obviously lacks a complex car architecture, authentic Electronic Control Unit (ECU), bus, protocols (Controller Area Network (CAN), AUTomotive Open System ARchitecture (AUTOSAR), etc.) and a multitude of sensing elements including RADAR, LIDAR, cameras or navigational elements. Additionally, it is devoid of the influence of a driver. Similarly, the road track is not comparable with a real traffic environment.

Accordingly, out of the large scope of Autonomous Driving technologies, and due to the limited capabilities of the Robot Cars, the implementation will focus mainly on the domain of Advanced Driver Assistance Systems (ADAS). It also does not address issues like hazardous events/behaviour nor security or safety issues, etc.

It is also important to consider that the implementation case is largely dependent on the capabilities, but also limitations and restrictions of the Physical Twin (Req04, Req41, Req42, Req43, Req44, Req45) as well as those of the data transmission between the twins, causing eventually latency and synchronization issues.

It is of utmost importance that access to the physical entity(ies) is granted and that timely support by its expert(s) or user(s) is provided. It has to be ensured beforehand and well managed. Conflicting interests should not be underestimated. Modifications to the physical entity during the realization of a Digital Twin could have a detrimental impact on the outcome, or even hinder its realisation/finalisation.

This section provides a thorough review and clarification of the PT, including its functionality to identify the capabilities of the DT, the model requirements, the methods for data transmission (connection, interaction, protocol, etc.) between the twins, as well as limitations and restrictions (e.g. memory capacity, processor speed, hardware, software, etc.).

For the purpose of this thesis, the PT consists of two small Robot Cars based on Arduino microcontrollers and an environment designed to represent Autonomous Vehicles in a laboratory setting. Depending on the specific scenario to be demonstrated with the Robot Cars, the environment is depicted as a road track, henceforth abbreviated to track, with a junction (bifurcation and merge), where a black line surrounded by a white background represents a vehicle lane. The track, which can include both straight sections and curves, is the sole object within the environment. Additional environmental parameters, including but not limited to weather conditions (e.g. reduced visibility, slippery or uneven driving surfaces), temperature, humidity, wind speed, precipitation, and altitude are excluded from the scope of this thesis.

[1] https://github.com/SQA-Robo-Lab

These Robot Cars, composed of multiple ECUs and equipped with driving, sensing, and communication capabilities, are capable of operating and driving autonomously, making them suitable for representing the PT for this thesis.

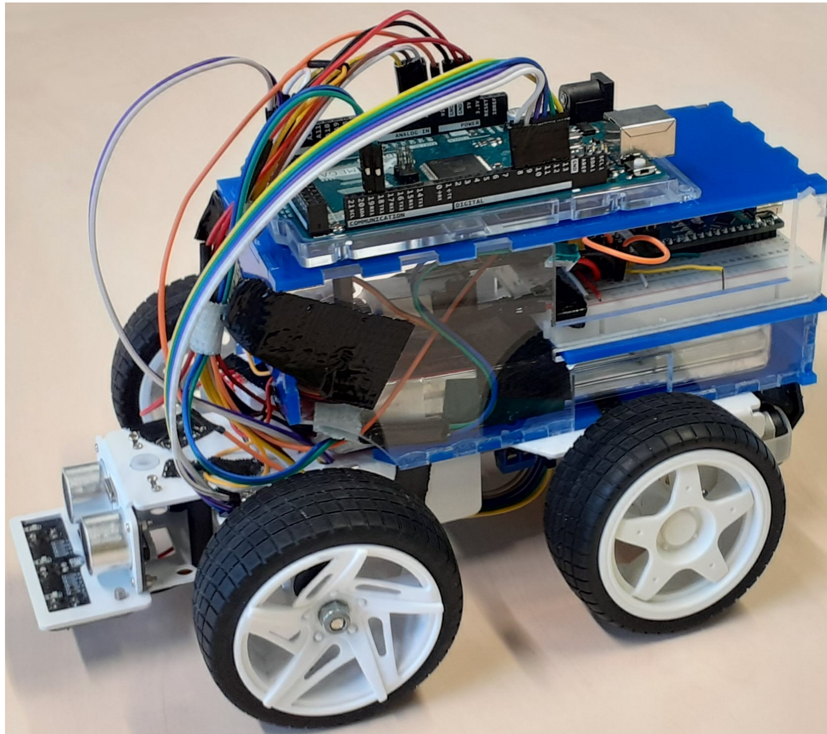

Figure 6.1: Robot Car

Each Robot Car (see Figure 6.1 and Appendix Section A.4) is equipped with two Arduino microcontrollers, an ESP WiFi module for wireless communication (implemented using MQTT, not HTTP or other protocols), an L298N Motor Controller Module for controlling the DC motors, two DC motors for the rear wheels, a steering servo for the front wheels, three combined grayscale sensors for line detection, and a distance sensor for measuring the distance at the front. Notably, these Robot Cars do have a simple on/off switch. A comprehensive list of hardware resources and further details can be found in Appendix A.

To support the utilization of these Robot Cars for short-term, time-limited projects such as this master's thesis, the university designates a contact person, referred to in this thesis as the "Robot Car expert". At the outset of this thesis, he refrained the author of this thesis to make modifications on the Robot Cars, as they are configured for other projects. Consequently, he maintains a Hardware Abstraction Library (HAL)[2] to ensure the basic functionality of the Robot Cars and abstract its complexity to a third party, provides an interface using MQTT for interaction and communication, and offers minor adjustments or functionality extensions if feasible, sensible, and necessary for short-term, time-limited projects.

[2] https://github.com/SQA-Robo-Lab/Sofdcar-HAL

6.2.1 Robot Car PT Limitations and Functionalities

Certain limitations inherent to the hardware, the design and mechanical construction and the Arduino software and the coding of the Robot Cars, impact and restrict the DT implementation possibilities.

The Robot Car will sometimes react abnormally.

The Robot Cars have a problematic center of gravity, particularly the blue Robot Car. The front axle of the blue Robot Car is wobbly and the wheels are misaligned. This results in inadequate traction and sub-optimal steering behaviour, as well as abnormal erratic zigzagging.

The Robot Car speed values can be set within a specified range (minSpeedCmPerSecond and maxSpeedCmPerSecond). The values were determined through testing, as the expert did not indicate precise values. The PT can provide them in the whole 16-bit range and can also be negative. However, the behaviour and performance of the Robot Cars is subject to variation depending on the level of battery charging. Consequently, the speed parameter must be adjusted upwards when the battery level decreases, and so on. The Robot Car does not cope well with road inclines, slopes bumps and uneven surfaces, mainly due to weak motors.

It has been observed that Robot Cars have a tendency to deviate from the intended trajectory on occasion. The Robot Car expert has stated that the issue is caused by the lack of PID implementation in the PTs, as well as calibration problems.

Other limitations are due to the type and/or model of sensors, actuators, etc.

As postulated by the Robot Car expert, the aforementioned issues have yet to be resolved, and some may not be solvable at all.

The aforementioned issues present a number of challenges, the resolution of which is contingent upon a number of factors.

The basic functional capabilities of the Robot Cars include speed control, steering, and obstacle recognition.

At the outset of this thesis, the Hardware Abstraction Library already encompassed abstractions for several functionalities of the Robot Cars, presented hereafter in detail, along with their associated constraints and restrictions:

PT Basic Driving Behaviour: The Robot Car is equipped with a rear-wheel drive system comprising two electric motors, one for each rear wheel. The HAL allows for setting a speed in centimeters per second for the motor and an angle for the servo to control horizontal movement. When steering via the servo, the rear wheels must rotate at different speeds depending on the servo angle. The Robot Car is capable of driving in reverse when a negative speed is set. However, reverse driving can be problematic, especially during steering, as the requisite corrective function (analogous to a differential correction as the inner and outer wheel need a different torque) for navigating reversals properly in turns or road bends has yet to be refined in the Robot Car code.
The Robot Car is devoid of brakes. Consequently, the process of deceleration and braking is achieved through the regulation of speed. Consequently, no authentic braking directive can be conveyed, nor can any pertinent status be observed and inferred, thus precluding the calculation of braking distance.
The driveControllerState provides implicit feedback of the vehicle's driving status

PT Obstacle Detection and Reaction (Pause on Obstacle): The Robot Car can react to distance measurements using a distance sensor (see Appendix Section A.1). Specifically, it stops when a preset distance is breached by setting its speed to zero. This functionality, referred to in this thesis as "Pause on Obstacle", does not account for braking distance. The previous speed value is retained through the driveControllerState, which provides implicit feedback on the vehicle's driving status, enabling the Robot Car to automatically resume driving at the same speed once the obstacle is no longer detected.

PT Line Following (Stay on Track): The Robot Car can follow a line using a three-channel grayscale sensor (see Appendix Section A.1), referred to in this thesis as the "Stay on Track" functionality. During this thesis, the line was black, and the surrounding area was white. This configuration can be inverted since the sensor can be calibrated to distinguish between the line and the surrounding area. Calibration should be performed at the beginning of a session if significant changes in light conditions occur, ensuring the Robot Car can accurately identify white as the non-line area and black as the line. Although semi-automating this calibration was considered, it was not implemented due to time constraints. During this thesis, the Robot Car followed the edge of the line rather than the center, which proved more reliable. This approach has the advantages of accommodating varying line thicknesses and allowing the Robot Car to follow different lines at bifurcations by setting it to follow the left or right edge. If the Robot Car is too fast, it may miss the line and drive over it due to delayed sensor data processing. The default speed to prevent the Robot Car from veering off the line varies and depends on the battery charge level, typically ranging from 60 centimeters per second to 70 centimeters per second.

PT Lane Changing: As previously discussed in the context of Line Following, the HAL allows for lane changes by configuring the line to be followed. Specifically, setting the input to 0 directs the Robot Car to follow the left edge of the line, while setting the input to 1 directs it to follow the right edge. This capability inherently implies that the Robot Car can change lanes by adjusting the line-following parameters.

6.2.2 Robot Car PT Communication and interaction capabilities / limitations

Although these functionalities existed, they could not be remotely accessed right away. The prototype DT should include control and adaptation mechanisms to enable the DT to continuously shadow (see Section 5), react to changes, and send commands to the PT, i.e. the two Robot Cars. However, the absence of remote communication and interaction capabilities in the PT necessitated extending the Hardware Abstraction Library to ensure proper interaction for shadowing, parameter modification, command transmission, scenario configuration, and live testing and simulation.
Given the Robot Cars' support for MQTT Pub/Sub, the Robot Car expert refined the HAL to support Remote Procedure Call (RPC) for request/response communication (see Subsection 6.1.5). Throughout the thesis, refining the RPC clarified the accessible functionalities, which ultimately led to the development of an RPC library (Req35). This library enables querying agreed-upon data and sending commands via MQTT (see Appendix Section A.3).

In consultation with the Robot Car expert, it was decided to use two topics per Robot Car: one for incoming messages and one for outgoing messages. This approach not only aligns with the microcontroller's limitations but also reduces the workload for the Robot Car expert, who has limited time available.

The Arduino microcontroller is constrained by its sequential operation and lack of parallel processing capabilities. Only a limited number of tasks can be processed within a specified time frame. It has been observed that occasionally, especially during complex procedures:

- some commands in the PT are executed (too) late
- information from the PT (e.g. sensor values or mode feedback) is provided late

This may also be attributed to an excess of messages undergoing processing concurrently, which may result in their either incomplete execution, delayed arrival, or complete absence.

The Robot Car expert has identified a potential issue with the Robot Car's Arduino microcontroller, which may manifest as a bug. This results in messages not being properly processed and/or acknowledged.

The program could only be uploaded and/or updated to the Arduino via a physical cable connection, thereby precluding any remote modifications to the Arduino software (e.g., disabling a sensor), that is, no Over-the-Air was implemented in the Robot Cars. However, simulations could be conducted, for instance, by altering subscriptions to sensor data, thereby creating the appearance of a remote modification of the Arduino software to disable the sensor.

For detailed information about the communication with the Robot Cars, including the Robot Car command structure, the types of commands, required formats, corresponding responses, and Robot Car network configuration, refer to Appendix Section A.3.

6.3 Solution approach, selection of Tools and Languages

A review of existing Digital Twin Software solutions was conducted. Among others the paper 'Survey on open-source digital twin frameworks–A casestudy approach' by Gil Arboleda et al. [GGLM24] gives an interesting analysis of some.

For example, Eclipse Ditto provides DT builder in the Industrial IoT a set of tools for a cloud-consistent view across a variety of field devices. At the core of Ditto is a data model, called a Thing, that provides a virtual representation of the PT. The Ditto Thing is accessible through an API that allows DT domain experts to interact with the device(s). Ditto services support interaction (via standard IoT protocol like AMQP, MQTT) with the data model through features like persistence and notification of changes to the device information, payload transformation, authorization policies, etc. [Ecl24a].

While Eclipse Ditto offers several advantages, such as extensive integration capabilities, it was not selected for this thesis as only a local prototype DT was implemented.

However, Eclipse Ditto's strengths should not be overlooked for future work. Its scalability makes it suitable for projects that may expand to handle a larger number of devices or more complex workflows. The extensive integration capabilities with other IoT frameworks and platforms can enhance the composability and extendability of the Digital Twin. Eclipse Ditto remains a viable option for future iterations of the Digital Twin, particularly as the project scales and requires more advanced features.

Additionally, other open-source frameworks such as OpenTwins [Rob23] were also investigated.

It was established that none of the considered options was appropriate for the constrained scope of the basic implementation of this Digital Twin prototype. Some lacked comprehensive documentation and/or required a steep learning curve to understand and utilize it effectively, which could result in delays to the thesis's progress. Moreover, complexity and/or lack of the necessary degree of flexibility or completeness could act as a significant barrier.

In order to achieve the desired research outcomes, it was deemed inappropriate to rely on a complete Digital Twin software suite or framework, as this would not allow for the necessary flexibility and adaptation to the specific requirements.

The decision was taken to develop a new DT from the ground up. This approach was deemed sensible also in order to gain a comprehensive understanding of all the intricacies involved and to examine each component and step in detail.

After having assessed the scope of the Use Case and of the prototype, suitability tools were investigated and evaluated, in order to develop this basic DT. Java and JavaScript (Node-RED) were chosen for rapid development capabilities and alignment with the thesis's immediate needs. This thesis uses explicitly only open-source tools.

Given the Physical Twin described in the previous Section 6.2, this section addresses the appropriate solution approach, tools and languages for the development of the Digital Twin, as well as for interfacing with the Physical Twin and database management.

Figure 6.2 presents an informal illustration depicting the PT and DT.

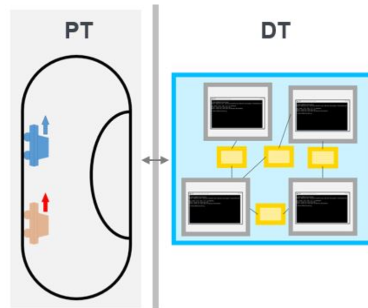

Figure 6.2: Informal illustration for PT connected to DT

On the left side, the Physical Twin is shown, consisting of two small Robot Cars as introduced in the previous Section 6.2 and an environment, depicted as a track with a junction (without further detail). On the right side, the composable and extendable Digital Twin is illustrated according to the concept from Chapter 5, which is divided into individual components that are loosely coupled and asynchronously connected, allowing them to send messages to each other.

Based on the concept presented in Chapter 5, the following solution approach was elaborated and illustrated in Figure 6.3.

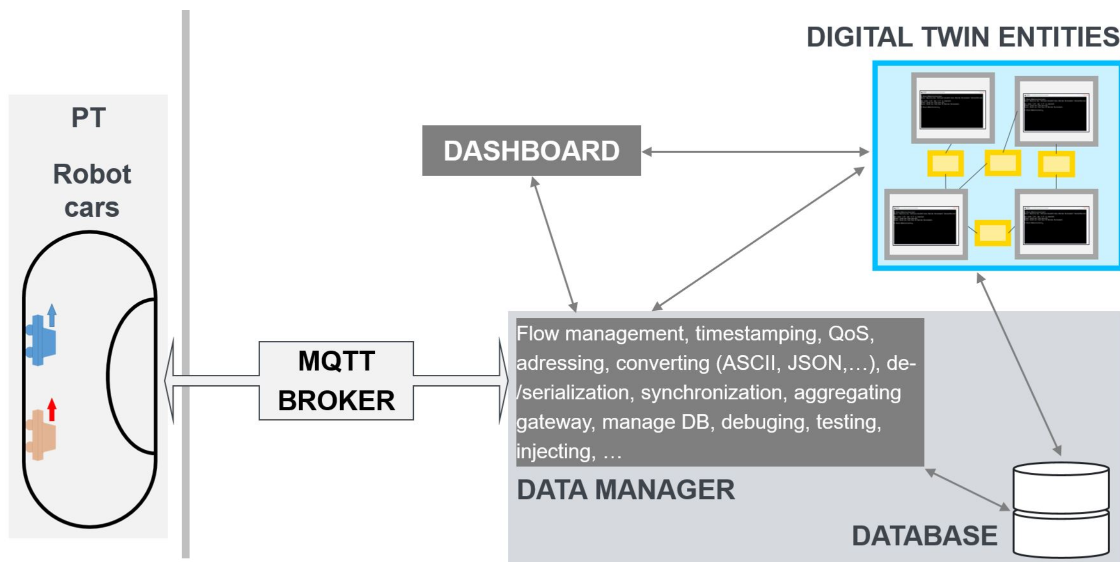

Figure 6.3: Informal illustration of the solution approach for DT (and connection to PT)

This approach was designed with the target to integrate more flexibility (Req25), composability (Req10, Req23) and extendability (Req24, Req37). Therefore the functions are structured in such a way that the gateway services, message-brokering, data flow management, addressing, conversion (of ASCII / JSON, etc.), de-/serialization, synchronization, aggregation, debugging, injection and testing, etc. are externalized of the framework/scenario code lines (Req06, Req15, Req16, Req18, Req28, Req34, Req38, Req39).
It is also advisable to set up a persistent storage of data, with adequate timestamping, etc., in order to ensure a certain level of reliability and durability of data (Req07, Req08, Req17, Req30, Req38).

In a DT environment, dashboards and supervision tools are useful if not essential for visualising and even virtualizing, as well as initiating commands (Req04, Req05, Req12, Req18, Req21, Req39).

After a comprehensive review, the following design, shown in Figure 6.4, was elaborated and the tools, presented hereafter, were selected, in order to obtain a coherent but flexible structure for the entire system (Req25) and in order to also facilitate future extensions and/or modifications and/or even the substitution of components or tools. Appendix B provides supplementary material for the selected software tools.

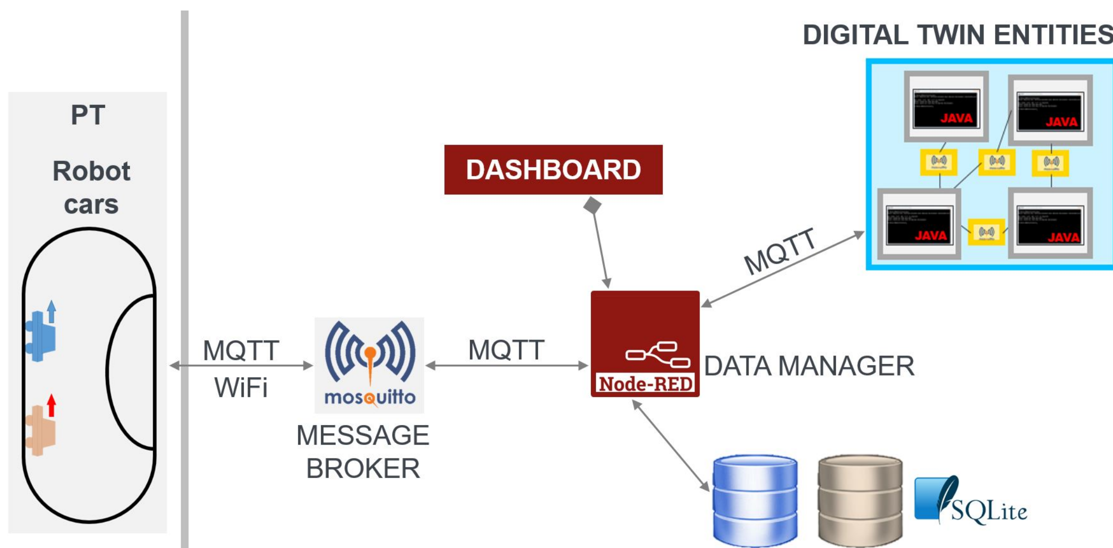

Figure 6.4: Enhanced informal illustration of PT and DT with software tools

MQTT protocol and Eclipse Mosquitto (MQTT broker)

As mentioned in the last Section 6.2, communication with the Robot Cars is only implemented using MQTT. The Robot Car expert had already successfully used Eclipse Mosquitto as the MQTT broker before this thesis. Mosquitto is an open-source message broker that implements the MQTT protocol versions 5.0, 3.1.1, and 3.1. It is lightweight and suitable for use on all devices, from low-power single-board computers to full servers [Ecl24b]. Mosquitto offers the advantages of being easy to install and configure, providing a reliable and efficient solution for vehicle communication [IoT23].

The Robot Cars use MQTT 3.1.1 on the Arduino microcontrollers.

This justifies the decision to use MQTT with Mosquitto as the broker for all further communication pathways, i.e. both within the DT (communication between individual DT components), between PT and DT, and for communication with the database, to maintain uniformity and because MQTT Pub/Sub (see Subsection 6.1.1) has proven advantageous (Req06, Req15, Req16, Req28).

Various communication methods, such as HTTP, RabbitMQ, and others (see possible alternatives [Sta24]), were considered and could have been implemented. However, they were not used for the implementation in this thesis mainly due to the existing setup for communication and interaction with the Robot Cars.

Java

For the development of the Digital Twin Entities and components, Java was selected due to its platform independence, which ensures that the Digital Twin Entities and components can operate seamlessly across various systems, enhancing flexibility (Req25) and deployment ease. The ecosystem of Java libraries and frameworks, such as Eclipse Paho for MQTT communication, simplifies the implementation of messaging protocols, which is crucial for the composability (Req10, Req23) and extendability (Req24, Req37) of the Digital Twin. Additionally, Java's strong community support provides ample resources for troubleshooting and optimization, which is beneficial given the project's tight bounded timeline. Java's object-oriented programming principles further aid in creating modular, reusable, and maintainable code, aligning well with the project's goals. Additionally, the use of object-oriented programming is particularly advantageous for the development of a Digital Twin, as it facilitates the creation of a virtual replica of the Physical Twin, which is inherently suitable to the principles of object-oriented design. Furthermore, the author's experience with Java eliminates the learning curve, allowing for efficient and effective development.

However, there are some considerations to keep in mind. Java's garbage collection mechanism, while generally efficient, can introduce unpredictable pauses, which might be a concern for real-time applications like Autonomous Driving. Additionally, Java applications tend to have higher memory and CPU usage compared to those written in lower-level languages, which could affect performance. The complexity of Java's ecosystem and dependency management can also pose challenges, particularly when integrating multiple libraries and frameworks. Nevertheless, given the scale and careful planning of this project, these cons have not had a visible impact on its success.

Middleware requirements

While Java provides a solid foundation for the Digital Twin Entities and components, additional middleware functionalities are necessary to manage and ensure clear and controlled communication between the DT and the Physical Twin.

Due to the internal structure of the Robot Cars, each message sent to a Robot Car must be a specifically formatted command string, transmitted in the MQTT payload as a buffer, with numerical values encoded as ASCII (for more details, see Appendix Section A.3).

However, this format is not optimal for messages exchanged within the DT-Entities. Converting the message payload into a buffer with numerical values encoded as ASCII is disadvantageous and unsuitable for internal DT communications. Moreover, messages within the DT may require additional information, such as the sender, a message identifier, or a correlation to another message, within the payload. Consequently, a message conversion mechanism between the PT and DT is required.

Additionally, it is essential to monitor both the Physical Twin and the Digital Twin at a glance and interact with each twin as needed. Therefore, a dashboard or a similar interface is necessary to provide these functionalities. Given that the Digital Twin comprises distributed Entities, a dashboard external to the Digital Twin is more suitable to ensure separation of concerns and better structure (Req04, Req05, Req12, Req18, Req21, Req39).

Node-RED

Node-RED was selected for the implementation in this thesis of its data management (flow and much more) and dashboard capabilities.
Implementing the stated middleware functionalities, such as message conversions and monitoring via a dashboard, directly in Java would be time-consuming and complex. A JavaScript tool like Node-RED offers a more streamlined approach, allowing for rapid development and deployment, which is crucial within the project's time-bounded scope. It also comprises comprehensive functions like libraries, gateway services, message-brokering, data flow management, easy conversion, de-/serialization, synchronization, and aggregation, as well as useful options for code debugging and injection for testing and simulations during the coding.

Node-RED is a flow-based development tool for visual programming, originally developed by IBM, that enables the wiring together of hardware devices, APIs, and online services through a browser-based editor [OC13]. This tool simplifies the creation of complex workflows by providing a wide range of nodes that can be easily connected and deployed. Node-RED's low learning curve allows for rapid development and deployment, which is critical within the thesis's time constraints. Its intuitive flow-based development interface enables quick prototyping and iteration, making it easier to manage the communication between the DT and PT.

Node-RED may face challenges when scaling up to handle a large number of devices or high-frequency data streams, potentially leading to performance degradation. The abstraction provided by Node-RED can introduce performance overhead, which might be a concern for real-time applications. For very complex workflows, the visual programming model can become cumbersome and harder to manage compared to traditional coding. Node-RED's reliance on various external nodes and

plugins might introduce compatibility issues or require frequent updates. Additionally, ensuring the security of data transmission and storage is crucial, and Node-RED's default settings might need to be hardened to meet security requirements. Despite these considerations, given the bounded scope of this thesis, Node-RED remains a suitable and effective choice for the implementation Use Case in this thesis.

Node-RED's built-in support for MQTT messaging aligns perfectly with the thesis's requirement for composable (Req10, Req23) and extendable (Req24, Req37) components.

Additionally, since Node-RED provides a user-friendly dashboard for monitoring and manual control, it also fulfils the requirement for an interface to monitor and interact with both the Physical Twin and the Digital Twin, thereby enhancing the overall usability of the system.

Database SQLite

A database is of use in order to cope with the need of persistent storage of data (e.g. sensor values but also set parameters) etc.

SQLite was selected for the implementation in this thesis as it is a good and popular choice for such small to medium-sized applications. SQLite is a lightweight, disk-based database that does not require a separate server process and allows access to the database using a nonstandard variant of the SQL query language [Com24]. SQLite is particularly suitable for embedded systems and applications with low to medium traffic, making it an ideal choice for logging data from the Digital Twin. This data can include sensor data originating from the Physical Twin, as well as actuator data, such as speed adjustments made by the DT, which is then stored in the database. This integration adds another layer of functionality, allowing for efficient data storage and retrieval.

Furthermore, SQLite is compatible with MQTT and easy to integrate with Node-RED.

Other databases such as InfluxDB, MongoDB, MySQL, etc. could be considered for larger projects and/or if scalability and flexibility are paramount.

6.4 Technical Realization

The architecture for this prototype DT, shown in Figure 6.5, is based on the one elaborated for the Use Case, which is described in the chapter Concept, Section 5.3 and shown in Figure 5.2.

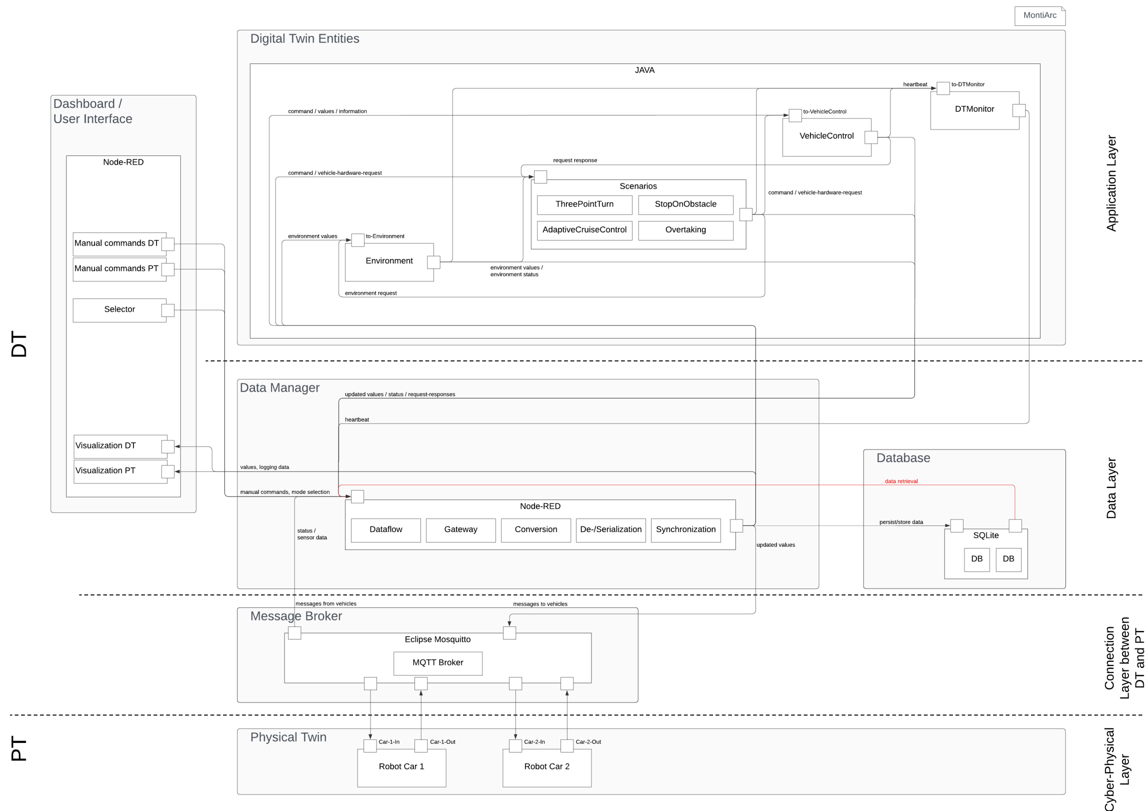

Figure 6.5: Architecture for the Implementation of a DT for Autonomous Driving

The prototype DT's elements are described hereafter in more detail.

6.4.1 Communication between PT and DT

The Robot Cars are connected with the DT via WiFi, through the university's intranet (not Internet for security reasons). The MQTT protocol, utilizing the Mosquitto broker, was employed to connect and communicate between the PT and DT (Req06, Req15, Req16, Req28). Further details can be found in Sections 6.2 and 6.3.

6.4.2 Data Manager

The Data Manager acts as a gateway, ensuring communication and managing the data flow between the various components, such as DT-Entities, Dashboard, database, and Physical Twin (Req15, Req16, Req28,Req31,Req38, Req39, Req40). This encompasses the conversion of data formats to facilitate interoperability (Req26), as well as the deserialization and serialization of data to and from the different components (Req19). Additionally, the Data Manager ensures synchronization between the DT and PT, maintaining consistency and coherence. Its role is critical to the effective and efficient operation of the integrated components within the Digital Twin framework. While the implementation of this Data Manager was a highly demanding and time-consuming task, detailing all aspects would be time-consuming. For more detailed information, the reader is encouraged to refer to Section 6.3 (especially the parts concerning middleware requirement and Node-RED) parts and the illustration in Figure 6.3 as well as to the code.

Figure 6.6 depicts an extract view of the Data Manager, which serves as an illustrative example of its inherent complexity. The Data Manager comprises a multitude of nodes, including, but by far not limited to, handlingVehicleControl, MQTT OUT to-VehicleControl, performDisconnectAction-VehicleControl, handlingAdaptiveCruiseControl, heartbeatMessage. The limitations of the time available for this thesis preclude a comprehensive and detailed account of the extensive structure of the Data Manager, which currently comprises over 1400 nodes.

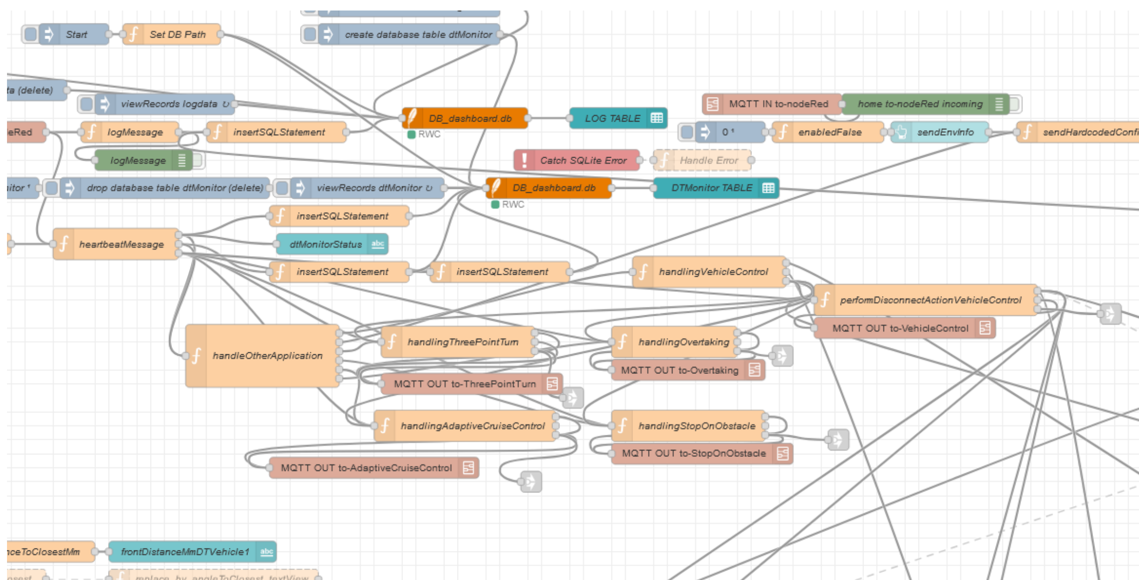

Figure 6.6: Caption of an extract view of a very small part of the Data Manager coded in Node-RED

As Node-RED provides a browser-based object-oriented editor. The development and utilization is a streamlined process. Its visual toolset also simplifies documentation, maintenance and future development.

The Data Manager in this implementation is tailored to this prototype and fulfils the tasks very well. However, fundamental elements of a Data Manager framework as presented in [APT24] such as data modelling, quality and uncertainty, etc. have not been further considered in this thesis and could be addressed in future work when and if relevant.

6.4.3 Database

As outlined in Section 6.3, SQLite is utilized as the database system to assure and preserve persistent storage (Req08, Req17, Req30). In this implementation, two databases must be created prior to the use of the Digital Twin:

1. `DB_DT`: This Database encompasses the replicated vehicles along with their hardware descriptions, structured as an SQLite table for each object (Req19). Each table contains elements that correspond directly to the attributes of the associated object classes, ensuring consistency in data representation. A detailed description of these tables is provided in Table G.1, with additional associated information, in Appendix G.
   For all hardware component objects that change, there are two tables:
   a) A general table where data for each object element is updated (i.e. executed through the SQL `UPDATE` command) in the same row.
   b) A second table (table name concatenated with "Changes") that records the changes by creating a new table entry (i.e. executed through the SQL `INSERT` command) for each change, where only the modified values along with their associations are recorded. As an example, Figure 6.7 displays the DistanceSensorChanges table of the Database dedicated to the DT, highlighting key attributes related to distance sensor data.

	changeID	distanceSensorID	vehicleID	timestamp	name	distanceToClosestMm	changedBy	usagesInVehicle	o
1	1	1	2	2024-10-26 17:54:14.358	UltrasonicSonarDistanceSensor	0	PT / Data Manager	[distance_front]	so
2	2	1	2	2024-10-26 17:54:15.587	NULL	65535	PT / Data Manager	NULL	N
3	3	1	2	2024-10-26 17:55:21.339	NULL	30066	PT / Data Manager	NULL	N
4	4	1	2	2024-10-26 17:55:26.268	NULL	20591	PT / Data Manager	NULL	N
5	5	1	2	2024-10-26 17:55:28.207	NULL	65535	PT / Data Manager	NULL	N
6	6	1	2	2024-10-26 17:56:23.967	NULL	25972	PT / Data Manager	NULL	N
7	7	1	2	2024-10-26 17:56:25.786	NULL	26991	PT / Data Manager	NULL	N
8	8	1	2	2024-10-26 17:56:27.647	NULL	65535	PT / Data Manager	NULL	N
9	9	1	2	2024-10-26 17:56:41.228	NULL	14951	PT / Data Manager	NULL	N
10	10	1	2	2024-10-26 17:56:41.873	NULL	65535	PT / Data Manager	NULL	N
11	11	1	2	2024-10-26 17:56:48.510	NULL	487	PT / Data Manager	NULL	N
12	12	1	2	2024-10-26 17:56:48.742	NULL	426	PT / Data Manager	NULL	N
13	13	1	2	2024-10-26 17:56:49.069	NULL	347	PT / Data Manager	NULL	N
14	14	1	2	2024-10-26 17:56:49.392	NULL	268	PT / Data Manager	NULL	N
15	15	1	2	2024-10-26 17:56:49.692	NULL	72	PT / Data Manager	NULL	N
16	16	1	2	2024-10-26 17:56:49.942	NULL	77	PT / Data Manager	NULL	N

Figure 6.7: Screenshot of the DistanceSensorChanges Table from the Digital Twin SQLite Database

During the configuration of the vehicles, these tables are automatically regenerated. If these tables exist from previous runs, the previous tables are deleted (i.e. executed through the SQL `DROP` command).

2. `DB_dashboard`: This Database is used for visualising relevant information on the Dashboard. It contains two fixed tables that need to be created once:

   a) `dtMonitor`: This table has columns for `DT application` and `status`, and it describes the `status` (e.g., "started", "running", "disconnected") of each Entity, i.e. DT Java application.

   b) `logData`: This table has columns for `timestamp` and `log`, and it records relevant information as logs with the associated `timestamp`. The Dashboard user can utilize this information to monitor the behaviour of the DT.

To ensure that entries for the tables `dtMonitor` and `logData` are written to the Database and conveyed to the Dashboard, it was essential for the Data Manager to be capable of writing to the SQLite Database. Due to the time constraints of this thesis, the Data Manager was also employed to write entries for `DB_DT` into the Database, rather than establishing a direct connection between the Java Entities and the Database. This did not impose any limitations on the thesis, as data is only written to the Database and not queried. In future work, however, depending on the scope of extensions, it may be more practical to implement a direct connection between the DT-Entities and the Database.

Moreover, it would be advisable to consider storing environmental data provided by the Environment Entity (see 6.4.9) in a separate table or database in future work. This would be particularly beneficial should the Environment Entity be enhanced with additional functionalities or factors.

6.4.4 Dashboard

The Dashboard, serving as the User Interface, is designed for both the visualisation of the DT and PT, as well as for executing manual commands (Req04, Req05, Req12, Req18, Req21, Req39, Req44, Req45).

The entire Dashboard, illustrated in Figure 6.18, is divided into several sections:

- Physical Twin Vehicle 1
- Digital Twin Vehicle 1
- Physical Twin Vehicle 2
- Digital Twin Vehicle 2
- Log
- Environment
- DT Monitor

Given that the PT consists of two (2) Robot Cars, as described in Section 6.2, two (2) Dashboard sections are required per Robot Car, namely the DT section and the PT section.

In summary:

- Digital Twin Vehicle 1 and Digital Twin Vehicle 2 pertain to the DT.
- Physical Twin Vehicle 1 and Physical Twin Vehicle 2 pertain to the PT.

Figure 6.8 illustrates the Physical Twin Vehicle 1 and Digital Twin Vehicle 1 Dashboard sections. The Physical Twin Vehicle 1 Dashboard section is responsible for the visualisation and direct control of vehicle 1, i.e. Robot Car 1.

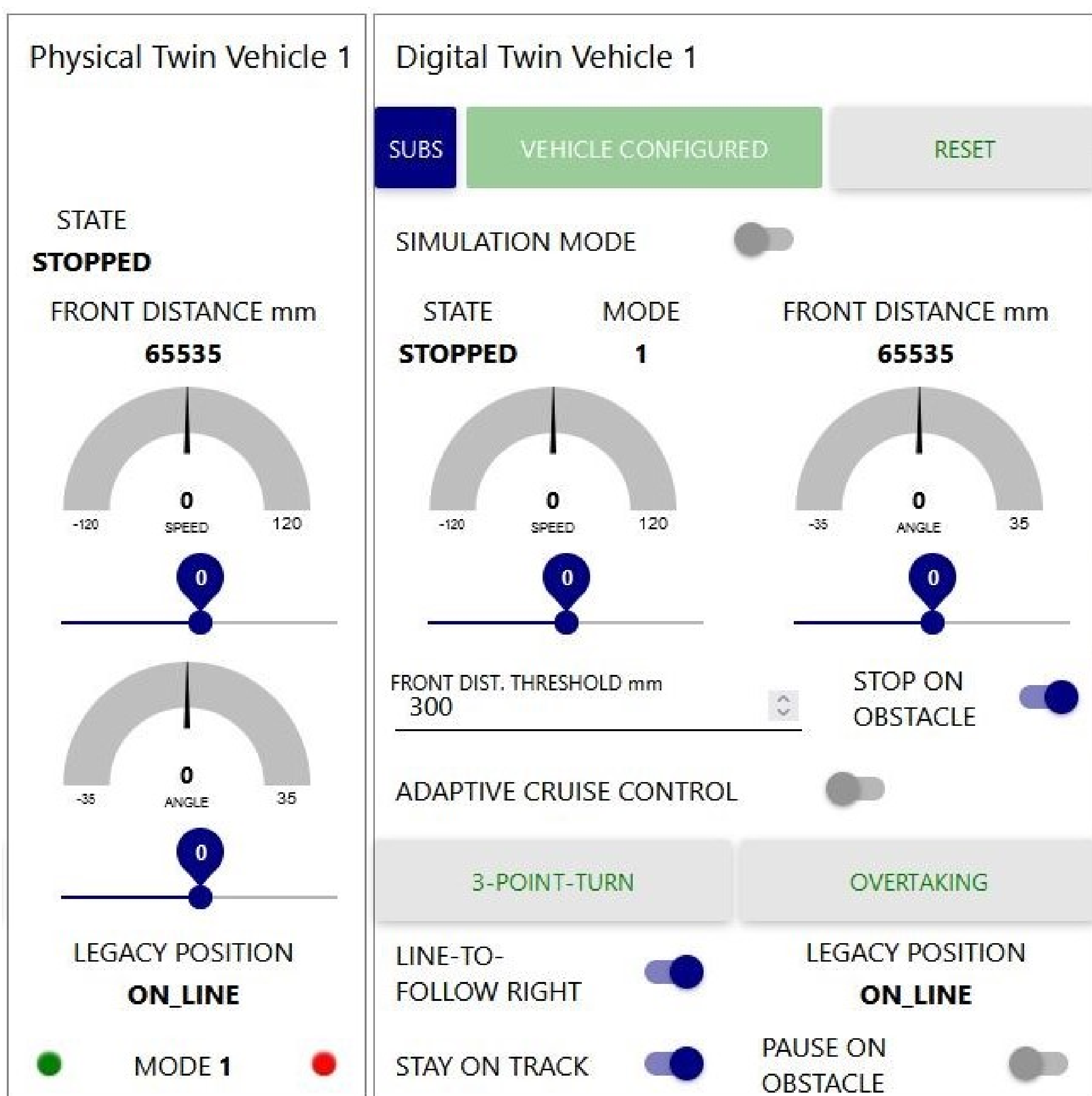

Figure 6.8: Dashboard Sections Physical Twin Vehicle 1 and Digital Twin Vehicle 1

It includes the following features: The text field labelled "STATE" describes the driveControllerState, which can be "STOPPED", "PAUSED", or "DRIVING". The text field labelled "FRONT DISTANCE mm" displays the measured distance to objects in the Robot Car's path in millimeters, using an Ultrasonic Sonar Distance Sensor. The gauge "SPEED" indicates the speed of the Robot Car within a range of -120 to 120 centimeters per second. The slider directly below the "SPEED" gauge corresponds to the speed, displaying the current speed value of the Robot Car and allowing direct speed adjustments. The gauge "ANGLE" shows the steering angle of the Robot Car within an angle range of -35 to 35 degrees. The slider directly beneath it corresponds to the angle, displaying the current angle value of the Robot Car and enabling direct angle adjustments. The text field labelled "LEGACY POSITION" indicates the position of the 3-channel grayscale sensor relative to the line, with possible values being: "LEFT_OF_LINE", "ON_LINE", "RIGHT_OF_LINE", and "LINE_POSITION_UNKNOWN".

As described in Subsection 6.2.1, the Robot Car encompasses the functionalities "Stay on Track" and "Pause on Obstacle".

"MODE" indicates whether these functionalities are individually active or inactive, with the following values:

MODE \ Functionality	"Stay on Track"	"Pause on Obstacle"
MODE 0	inactive	inactive
MODE 1	active	inactive
MODE 2	inactive	active
MODE 3	active	active

Table 6.1: Functionality States of RC under different MODE Values

The left LED from the text field "MODE" indicates whether "Stay on Track" is active in the Robot Car, while the right LED indicates whether "Pause on Obstacle" is active. Red color signifies inactive, and green color signifies active.

The corresponding Digital Twin Dashboard section, Digital Twin Vehicle 1 is responsible for the visualisation and direct control of vehicle 1 in the DT (Req13). Pressing the "SUBS" button sends all relevant subscriptions in a sequential manner with a delay per subscription message to the Robot Car. This ensures that the Physical Twin Vehicle 1 Dashboard section visually displays the current state of the Robot Car 1. Detailed information regarding the subscriptions is provided in Appendix Subsection A.3.1.

The "CONFIG" button configures the Robot Car in the DT, creating its virtual replica based on the selected relevant information. Upon successful configuration, the button label changes from "CONFIG" to "VEHICLE CONFIGURED". The "RESET" button is enabled upon configuration completion and is used to delete the created virtual replica (Req34).

The "SIMULATION MODE" switch is the selector for the simulation mode (Req14). If the execution of a simulation, such as one of the implemented scenarios, is desired to be initiated by the DT, this switch must be enabled.

The text fields labelled "STATE", "MODE", "FRONT DISTANCE mm", and "LEGACY POSITION" display the respective states of the elements of the Robot Car's virtual replica in the DT, analogous to those in the Physical Twin Vehicle 1 Dashboard section.

The gauges "SPEED" and "ANGLE" display the speed and angle of the virtual replica of the Robot Cars in the DT, analogous to the Physical Twin Vehicle 1 Dashboard section. The sliders positioned directly below each gauge are used for visualisation and to execute speed and angle adjustments.

The text input field. labelled "FRONT DIST. THRESHOLD mm", is used to set and display the threshold for the StopOnObstacle scenario (see 6.4.13), provided it is active. This means the StopOnObstacle scenario cannot be activated until the user has defined a threshold that the scenario will utilize.

The switch "ADAPTIVE CRUISE CONTROL" enables the activation of the AdaptiveCruiseControl scenario (see 6.4.14). Pressing the "3-POINT-TURN" button executes the ThreePointTurn scenario (see 6.4.12). Pressing the "OVERTAKING" button executes the Overtaking scenario (see 6.4.15).

The switch "LINE-TO-FOLLOW RIGHT" allows the user to choose which edge of the line the Robot Car's 3-channel grayscale sensor should adjust towards (left or right). Here, 0 signifies left. This element is termed "LINE-TO-FOLLOW" because, in the case of a bifurcation, two single lane road tracks separate, and the Robot Car will follow the edge of the road track it was initially tracking.

The switch "STAY ON TRACK" activates or deactivates the PT's functionality "Stay on Track" in the Robot Car. Similarly, the switch "PAUSE ON OBSTACLE" activates or deactivates PT's functionality "Pause on Obstacle" in the Robot Car.

Due to their nature, not all scenarios can be active concurrently, as different sensor information may necessitate various behaviours from the Robot Car. Consequently, activated scenarios or functionalities may disable or cancel other scenarios or functionalities that are active when triggered. Therefore, the buttons and switches are enabled and disabled based on the conflict management as presented in Subsection 6.4.16.

Ideally, the visualised data from Physical Twin Vehicle 1 and Digital Twin Vehicle 1 should match. In this case, the PT is synchronised with the DT, meaning Digital Twin Vehicle 1 displays the shadow (Req02) of Physical Twin Vehicle 1, i.e. Robot Car 1.

The Dashboard sections for Physical Twin Vehicle 2 and Digital Twin Vehicle 2 are analogous to those for Physical Twin Vehicle 1 and Digital Twin Vehicle 1.

For accurate behaviour, the corresponding buttons and switches to execute or activate the respective Entity or Entity-Scenario are only enabled when:

(1) the corresponding Entity or Entity-Scenario Java application is running and not disconnected or terminated

(2) the corresponding vehicle, to which the Entity or Entity-Scenario applies, exists i.e. is configured and possesses the basic capabilities, and

(3) has the relevant hardware components, such as sensors, necessary for the execution or activation of the respective Entity or Entity-Scenario

This implies that the "CONFIG" button is only enabled if the DT-Entity VehicleControl is recognised/perceived as running, and that each button, slider, switch, or text field for each basic driving capability, PT functionality or Entity-Scenario is only enabled when the corresponding vehicle is configured.

The remaining Dashboard sections Log, Environment, and DT Monitor are shown in Figure 6.9.

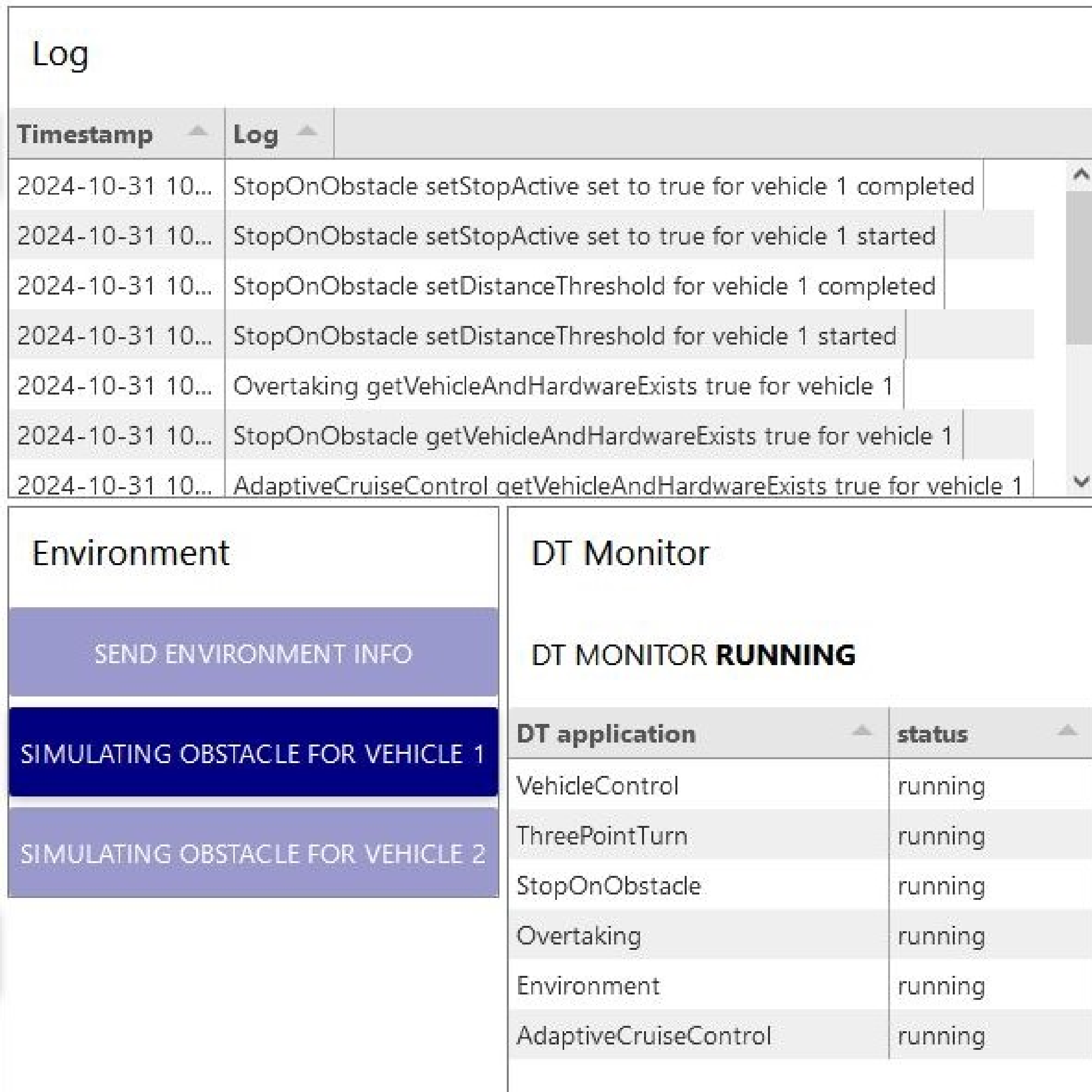

Figure 6.9: Dashboard Sections Log, Environment and DT Monitor

The Log Dashboard section displays log information relevant to the user, along with the corresponding timestamp (Req33). These log messages are also stored in a Database with the timestamp.

The Environment Dashboard section describes the PT's environment. It includes three buttons. Pressing the "SEND ENVIRONMENT INFO" button sends relevant environment information to the DT-Entity Environment. The buttons "SIMULATING OBSTACLE FOR VEHICLE1" and "SIMULATING OBSTACLE FOR VEHICLE2" allow for the simulation of virtual obstacles in front of the respective Robot Cars. Pressing either button sends a single low-distance value (10 millimeters) to the DT for the respective vehicle's distance sensor, which, if the StopOnObstacle scenario is active, results in the Robot Car receiving a command to stop because a (virtual) obstacle has been detected by the distance sensor.

The DT Monitor Dashboard section indicates whether the DT Monitor is active, i.e. "RUNNING" / "STARTED", or "DISCONNECTED". Additionally, this Dashboard section provides an overview of the Entities that have been active at least once while the Dashboard is active. The status of each Entity can be "started", "running", or "disconnected". In the Digital Twin Dashboard sections, only the switches and buttons for Entities marked as active (i.e. "started", "running") and only if the respective vehicle has been configured are available. The "CONFIG" button is only usable/clickable if the VehicleControl Entity is marked as active. Therefore, the DT Monitor must be in the "STARTED" / "RUNNING" state to recognise which other DT-Entities are active/available (Req12, Req18, Req25, Req28, Req29).

6.4.5 Overview of the PT's functionalities used in DT w/o replication

Certain functionalities are already implemented in the Robot Car (PT), herein so-called PT's functionalities 6.2.1. Those are not replicated in the DT but are directly addressed via messages, either at the Dashboard level or where and as needed by the respective Entity(ies).

These are for example:

- PT Basic Driving Behaviour (steering, speed, etc.)
- PT Obstacle Detection and Reaction (Pause on Obstacle)
- PT Line Following (Stay on Track)
- PT Lane Changing

6.4.6 Overview of the Implementation of the Digital Twin Entities

The Digital Twin Entities are coded as Java applications which are loosely coupled and communicate via messaging using MQTT protocol. The structure of the prototype DT-Entities implementation, coded in Java, is shown in Appendix C.

In line with the concept and Use Case, the following general-purpose Entities have been introduced:

- The VehicleControl Entity as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "VehicleControl", and includes the parts:
  - vehicle configurator (comprising the vehicle identifier, its elements e.g. actuators, sensors, etc. and their respective characteristics)
  - storing and monitoring the state of each vehicle
  - issuing commands to adjust and set the vehicle's behaviour and other properties
- The Environment Entity as mentioned in Subsection 5.3.2, has been implemented as an application/class denominated "Environment". In this prototype, the values have been hardcoded as no such inputs are available from the PT
- The DTMonitor Entity as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "DTMonitor"

The functions that are not available in the Robot Car but are implemented in the DT are referred to herein as "DT-Entity-Scenario".

The DT-Entities-Scenarios listed below were selected based on their feasibility within the limitations of the Physical Twin as outlined in Section 6.2, and the constraints and assumptions associated with this thesis:

- The ThreePointTurn Entity-Scenario as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "ThreePointTurn". In this prototype, it does not rely on sensor data. It follows a hardcoded manoeuvre (time-triggered, pre-defined speeds and steering angles, etc.), unless a predefined trigger, such as a sensor detecting an obstacle in the vehicle's path, interrupts and cancels the procedure
- The StopOnObstacle Entity-Scenario as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "StopOnObstacle". In this prototype, the restarting of the movement can only be done on the DT side.
  (Note that in this implementation, the PT's functionality "Pause on Obstacle", contrary to the StopOnObstacle, does not require a restart action but resumes autonomously as soon as the obstacle is out of range.)
- The AdaptiveCruiseControl Entity-Scenario as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "AdaptiveCruiseControl"
- The Overtaking Entity-Scenario as mentioned in Subsection 5.3.2, has been implemented as an application / class denominated "Overtaking"

6.4.7 General information to the coding of the Entities as Java Applications

Message Format

As previously mentioned, the format used to interact with the Robot Cars is disadvantageous and unsuitable for internal Digital Twin communications. These internal messages may require additional information, such as the sender, a message identifier, or a correlation to another message, within the payload.

The Java Message Service (JMS) is a Java API that allows applications to create, send, receive, and read messages. It defines a common set of interfaces and associated semantics that enable programs written in the Java programming language to communicate with other messaging implementations. JMS facilitates the sending and receiving of messages between software systems, supporting both point-to-point and Publish/Subscribe messaging models [Ora24]. JMS properties are key-value pairs used in JMS to provide additional information about a message, aiding in tasks such as message filtering, routing, and attaching custom metadata. They enhance the flexibility and functionality of message handling in JMS [Ora24].

Accordingly, the manner in which the required additional information in the DT can be packed into a message payload was inspired by the following JMS properties/fields:

`publisher`: Analogous to `JMSPublisher` / `JMSSender`, to indicate the publisher of a message.

`method`: To indicate the purpose of a message.

`messageID`: Analogous to `JMSMessageID`, to uniquely identify a message, which can also be used for tracking and referencing messages. Instead of having a supplementary property `Timestamp`, the timestamp is used as the `messageID`. The timestamp, measured in milliseconds, represents the difference between the current time and midnight, January 1, 1970 Universal Time Coordinated (UTC). The granularity of the value depends on the underlying operating system and may vary.

`priority`: Analogous to `JMSPriority`, to specify the priority level of a message. A ten-level priority value is used, with 0 as the lowest priority and 9 as the highest. Priorities 0-4 are considered gradations of normal priority, while priorities 5-9 are considered gradations of expedited priority.

Additionally, if necessary:

`replyTo`: Analogous to `JMSReplyTo`, to indicate the destination to which the message receiver should send a reply message. This element is only included in a message if a reply/response is expected due to the purpose of the message.

`correlationID`: Analogous to `JMSCorrelationID`, to link one message with another. It is used to link a reply/response message with its requesting message by containing the requesting message's `messageID`.

Additional properties, which are not present in JMS and are used only when necessary, include:

`status`: Describes the status concerning the purpose of the message, often used as feedback.

`confirmationRequired`: Indicates that a confirmation is needed after executing the respective purpose of the message.

`booleanValue`: Describes a flag required for some purposes.

`vehicles`: Describes the vehicles with their respective relevant properties involved in a message.

The JMS field `JMSDestination` was substituted using the corresponding receiver topic for the MQTT message. Further properties analogous to `JMSDeliveryMode`, `JMSRedelivered`, `JMSType`, and `JMSExpiration` could be inserted into a message if necessary in future work.

This could have been realized using MQTT message properties similar to JMS. However, this approach is comparatively complex and has the limitation that MQTT 5.0 is required, as the MQTTProperties object is not supported in MQTT v3.1.1.

JSON strings offer several advantages over MQTT properties, providing a flexible foundation for structured communication between distributed applications, and ensuring interoperability and efficient parsing across various programming languages and platforms (see JavaScript Object Notation (JSON)). Moreover, JSON is widely supported across various programming languages and platforms, facilitating interoperability between different systems. For Java, there are currently several fast JSON libraries available, ensuring efficient parsing and serialization of JSON data (see JavaScript Object Notation (JSON)).

Using JSON strings for message payloads also allows for direct application of message payload encryption, enhancing security for future work. This approach ensures that the entire message content can be encrypted, providing a higher level of data protection compared to using individual MQTT properties.

In summary, the adoption of JSON as a data interchange format in messaging protocols like MQTT provides a flexible and secure foundation for structured communication between distributed applications.

For message responses, it is considered best practice to clearly indicate the purpose in the naming. However, since the response messages are treated identically in this thesis the uniform naming "response" has been used to indicate a response for both, a `get` message and a message containing the `confirmationRequired` field. To track and correlate these responses with the original messages, the `correlationID` (`messageID` of the original message received) has been implemented to ensure a consistent identifier throughout the entire process.

Handling of Messages in Digital Twin Java Applications

Figure 6.10 visualises how an Entity, i.e. Java application, handles a received message in general. The processes explained here form the foundational workflow for all Entity Java applications, with additional entity-specific workflows elaborated in subsequent sections.

To simplify the understanding of the workflow, note the following: descriptions that include 'current', pertain to messages that had previously arrived in an entity prior to the depicted sequence, along with their already initiated processes. Consequently, descriptions including 'current' describe processes that are already running when a new message arrives. Conversely, descriptions that include 'new', pertain to messages that arrive subsequently and their associated processes. This distinction clarifies the subject matter. Figure 6.10 and this description are framed from the perspective of the 'new' message and its associated processes.

Given the need for loose coupling, it is essential to ensure that one Entity can send messages specifically to another Entity, and that each Entity can receive messages.

The MQTT protocol with the Mosquitto broker is a suitable solution for this. Each Entity has a dedicated topic, named by concatenating "to-" with the respective Entity Java application's name (e.g. topic "to-VehicleControl" for the Entity Java application VehicleControl). When an Entity wishes to send a message to another Entity, the publisher client within that Java application publishes a new MQTT message to the specified topic. This new message is translated via the Mosquitto broker. The listener client, having subscribed to the topic within the receiving Java application, receives the new message.

After the listener client, of the message-receiving application, receives the new message, this new message, which has a JSON string as payload as previously introduced, is saved and converted into a JSON object for more effective handling of its content. From the content of this JSON object, a string is generated to identify and describe the procedure of the received new message. This is necessary to ensure that arriving messages, which can be processed in parallel (i.e. they do not access or overwrite the same memory resources and thus do not cause race conditions), are indeed processed in parallel. This procedure string includes at least the method, which describes the purpose of the new message, and if present in the new message, additional elements such as the vehicle identifier.

To manage the execution of tasks, `newFutureTask`, an instance of the `FutureTask` class, is generated. The `FutureTask` class provides a cancellable asynchronous computation that allows for targeted cancellation or interruption of an ongoing procedure at any future point. The `newFutureTask` is responsible for executing the specific task defined in the new message using the `processMessage` method. Additionally, it may transmit a `confirmationMessage` after execution if required.

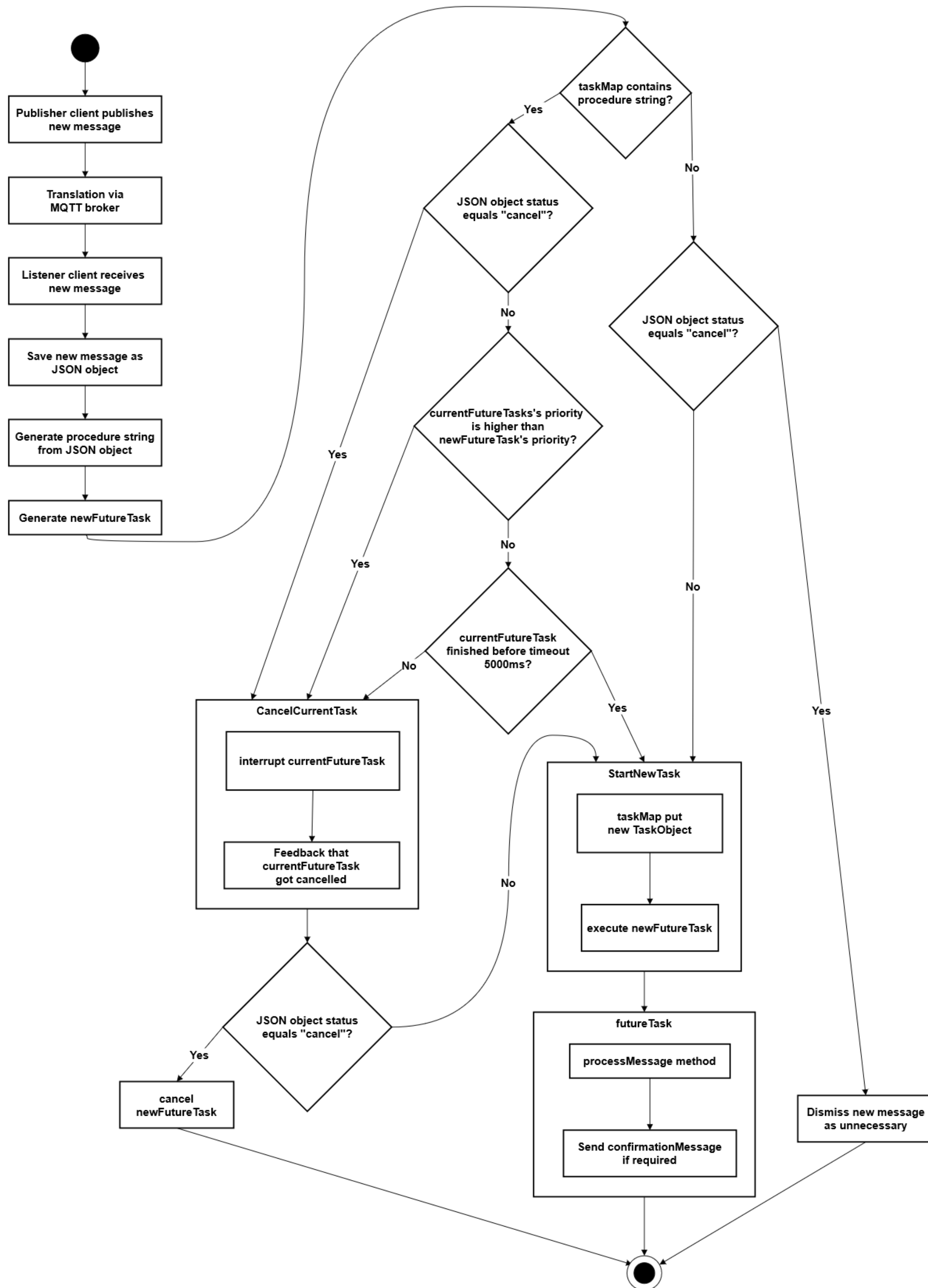

Figure 6.10: Handling of Messages in Digital Twin Java Applications

For effective management and retrieval of tasks, the instance newFutureTask, along with the message priority and the JSON object, is encapsulated within a custom class called TaskObject. This TaskObject is then stored in a ConcurrentHashMap instance known as taskMap, with the corresponding procedure string serving as the key, as shown in Listing 6.1. The ConcurrentHashMap is chosen over a regular HashMap to ensure thread-safe access by multiple threads, maintaining data integrity and preventing concurrency issues.

Listing 6.1 Java code for storing new TaskObject instance containing priority, newfutureTask and JSON object inside taskMap

```
1 taskMap.put(procedureString, new TaskObject(jsonObject.getInt("priority"), futureTask, jsonObject));
```

After creating newFutureTask associated with the new message, it is checked whether the taskMap already contains this identical procedure string. If this is the case, it indicates that a previous task (currentFutureTask) is already running, accessing the same memory resources, and thus newFutureTask cannot run in parallel. If the JSON object additionally contains a status field with the value "cancel", it means that the purpose of the new message is to cancel/interrupt the running currentFutureTask. In this case, currentFutureTask is cancelled. This involves interrupting the running currentFutureTask and sending feedback to the sender indicating successful cancellation. Subsequently, newFutureTask is also cancelled since its purpose was solely to cancel currentFutureTask.

If the taskMap contains the procedure string but the JSON object associated with the new message does not contain the status field, or its value is not "cancel", then newFutureTask must be executed regularly. In this case, the message priorities of currentFutureTask and newFutureTask are compared using the taskMap. If newFutureTask has a higher priority, currentFutureTask is cancelled, and newFutureTask is executed. If the priority of currentFutureTask is equal to or higher, newFutureTask waits for currentFutureTask to complete, checking the taskMap until a timeout of 5000ms is reached. If exceeded, currentFutureTask is cancelled, and newFutureTask is executed.

If the taskMap does not contain the procedure string and the JSON object contains the field status with the value "cancel", it indicates that currentFutureTask is completed or interrupted just before or shortly after the arrival or processing of the new message. In this case, the new message and newFutureTask are dismissed as their purpose is unnecessary.

If the taskMap does not contain the procedure string and the JSON object does not contain the field status with the value "cancel", newFutureTask is initiated. This involves creating a TaskObject with the priority of the new message, newFutureTask, and the JSON object. This TaskObject is then stored in the taskMap with the procedure string as the key and newFutureTask is executed. The processMessage method is then called to handle the new message specifically.

This architecture, leveraging FutureTask, ExecutorService, and ConcurrentHashMap, provides a scalable solution for handling dynamic and parallel message processing from an arbitrary number of senders in the DT's Entity Java applications. It ensures sensible task management, including task cancellation and prioritization, while maintaining the ability to handle a diverse array of message senders and purposes.

6.4.8 DT-Entity VehicleControl

The Java application VehicleControl is responsible for creating and managing the virtual replication of the relevant vehicles' hardware components as well as for modifying and setting the basic driving capabilities of the PT's vehicles (Req03, Req18, Req39, Req40, Req44).

These modified and set basic driving capabilities are integrated in the virtual replication, ensuring they are accurately reflected within the DT.

For the virtual replication, the Java application VehicleControl utilizes dedicated (custom) object classes to instantiate vehicle objects (Req11), each defined with its unique properties through a configuration process (Req19) (elaborated further below). These instantiated vehicle objects are stored in a list, allowing for the creation of an arbitrary number of vehicles. The vehicle objects can be identified and accessed using the Vehicle Identifier, facilitating efficient management and modification.

PT Vehicle Configuration

To receive data on the relevant vehicles' hardware components, a configuration process is required. Herein, VehicleControl acts as the Vehicle Configurator. The sequence diagram in Figure 6.11 elucidates the configuration process.

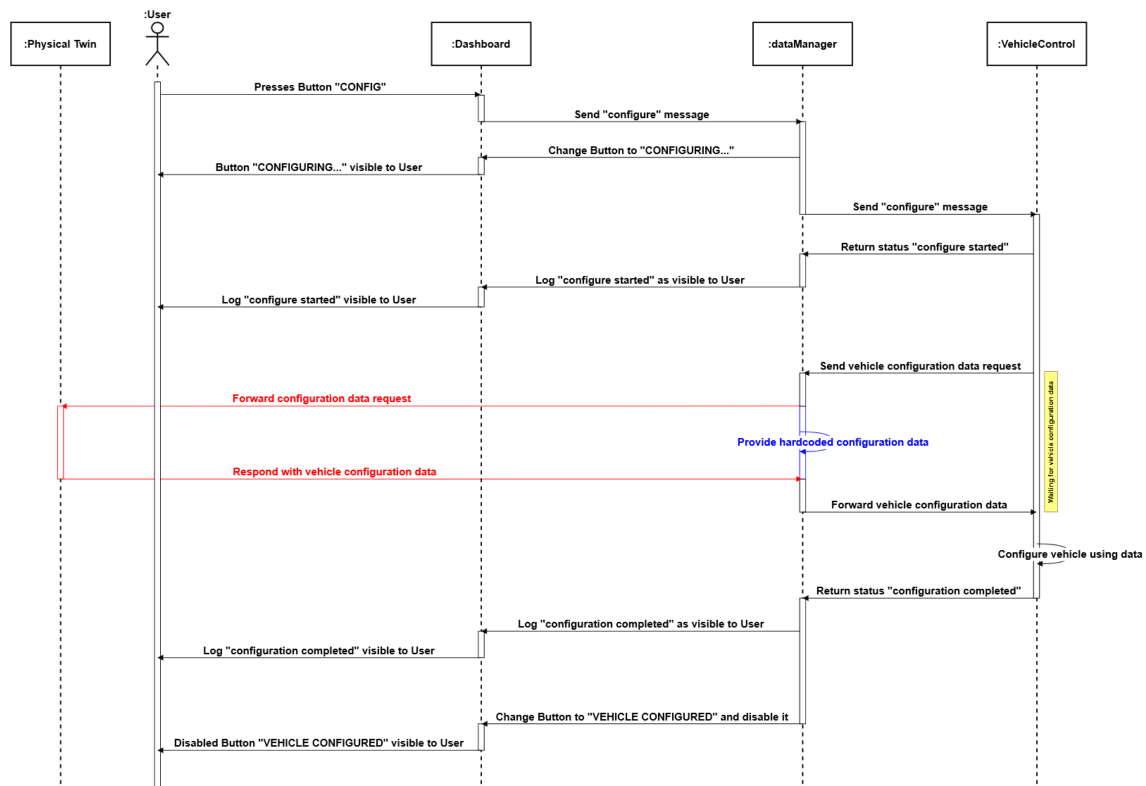

Figure 6.11: PT Vehicle Configuration Process

The configuration of a specific vehicle is manually initiated by a user through the Dashboard. Upon receiving a message regarding that vehicle with "configure" as the value for the JSON field `method`, the Java application VehicleControl transmits a status message to the Data Manager, indicating that the configuration process for the specified vehicle has begun. This information is subsequently recorded in the Dashboard for user awareness. The Java application VehicleControl then requests the configuration data for the specific vehicle from the Data Manager and awaits the response. Ideally, the Data Manager would concurrently forward this request to the PT to obtain the vehicle configuration data, as indicated by the red arrows in Figure 6.11 (Req34, Req37).

However, due to time constraints faced by the Robot Car expert and the limited memory capacity of the Arduino microcontroller, the configuration data of the PT vehicles, i.e. the Robot Cars (serial number, components, software version, hardware components, etc.), was not provided directly by the Robot Cars. The expert planned to transition the storage of configuration data from flash memory to EEPROM, a non-volatile memory that retains data even when the device is reset or powered off. However, this was not feasible within the timeframe of this thesis.

Therefore, since obtaining real configuration data from the Robot Cars was not an automated process, as not made available by the Robot Cars, this step is bypassed by sending the configuration data as hardcoded configuration data from the Data Manager to the Java application VehicleControl, as if it originated from the PT, as indicated in blue in Figure 6.11.

Upon receiving this configuration data, i.e. a message regarding that vehicle with "configData" as the value for the JSON field `method`, the Java application VehicleControl uses this information to configure the respective vehicle (Req35). The data is then stored in the runtime memory of the Java application VehicleControl as well as in the Database.

Finally, the Java application VehicleControl sends a status message to the Data Manager indicating that the configuration has been completed. This status is then logged in the Dashboard for user visibility, and the respective configuration button in the Dashboard is disabled.

All the configuration data were queried from and agreed upon with the Robot Car expert and are detailed with additional information in Appendix Section A.2. Notably, this implementation allows for quick adjustments if, at some point, the PT provides the configuration information. The hardcoded values are set in a manner that ensures the Robot Cars always start in a specific initial state.

Furthermore, the Java application VehicleControl facilitates the deletion of an individual or all vehicle objects that have been configured and thus are virtually replicated. To execute this deletion, a message can be sent to the Java application VehicleControl with "delete" or "deleteAllVehicles" as the value for the JSON field `method`. A user can manually initiate the deletion of a specific vehicle by using the "RESET" button in the Dashboard, which becomes enabled once the respective vehicle has been configured (Req19).

Once a vehicle has been configured, the Java application VehicleControl is responsible for storing the state of the vehicle. This means that any changes in vehicle information, values, and relevant commands are processed and maintained within this DT-Entity Java application (Req34, Req36). Consequently, the respective vehicle's behaviour can be controlled and adjusted using this DT-Entity Java application, either manually or through the use of an Entity-Scenario. Furthermore, all

commands and values from the Entities-Scenarios directed towards the PT's vehicles, i.e. the Robot Cars, as well as the Database and Dashboard, are routed through this central Entity, making it the core component of the DT's set of Entities.

To request information about specific vehicle states/values, a message can be sent to the Java application VehicleControl with the value "get" for the JSON field `method`, along with additional fields specifying the desired information. If the information relates to hardware data, the fields `usagesInVehicle` and `attachmentOnVehicle` must be provided to identify the relevant hardware component and its state/values.

Similarly, to set/adjust the state/values for a particular vehicle, a message can be sent to the Java application VehicleControl with the value "set" for the JSON field `method`, including additional fields that specify the value(s) to be set/adjusted. It is also necessary to specify the source of the adjustment in the message, as this may trigger specific behaviour, such as the need to send cancellation messages (see Subsection 6.4.7). DT-Entity VehicleControl only forwards updates to the Data Manager when an actual change of state has occurred.

Modifying and Setting the Basic Driving Capabilities of the PT's Vehicles

The foundation of the virtual replicas was significantly influenced by the time constraints of this thesis and the PT used within its scope. Consequently, object classes were created solely for the relevant vehicles' hardware components. However, the structure of these object classes is designed to facilitate adjustments and extensions for different PTs, thus ensuring future extensions and flexibility (Req25).

Given that the PT comprises Robot Cars equipped with DC motors, where velocity is regulated by providing a speed value, the processes of acceleration and braking have been abstracted into a single operation of setting a speed value, which implicitly aligns with Traditional Cruise Control, as introduced in 5.3.2. Consequently, the basic driving capabilities of the PT vehicles in this thesis, i.e. the Robot Cars, encompass setting a speed for forward or backward locomotion and controlling steering movement through the angle of the servo.

The PT's functionality braking, if available in the PT, can be integrated in the DT. The way of integrating it in the DT may vary depending on the type of braking system to be emulated, e.g. hydraulic brakes, Electro-Mechanical Brakes (EMB), brake-by-wire systems, anti-lock braking systems (ABS) and anti-skid systems, and Electronic Stability Programs. The implementation of this braking functionality may also affect other Entities, Entities-Scenarios, and components, such as the Data Manager, Dashboard and Database (Req24, Req25, Req34).

PT's Functionalities

Furthermore, the virtual replication structure allows to replicate functionalities which already exist in the respective PT vehicle (Req02).

In Section 6.2, it was explained that the functionalities "Stay On Track" (see 6.2.1), "Change Track at Bifurcation" (implicitly included in 6.2.1 Robot Car PT Limitations and Functionalities) and "Pause On Obstacle" (see 6.2.1) are already integrated within the PT and hence can be activated or deactivated using according messages to the Java application VehicleControl which are then

forwarded to the Data Manager and from there into the PT. The potential interplay and concurrent execution of these PT's functionalities will be addressed in more detail in Subsection 6.4.16.
These PT's functionalities can be set as described above by sending a message to the Java application VehicleControl with the value "set" for the JSON field `method`, including the information on which PT's functionalities should be set active/inactive.

6.4.9 DT-Entity Environment

The Java application Environment is responsible for managing environmental data from the Physical Twin, thereby providing an accurate picture of current conditions for vehicles and making this information accessible for further use. This is crucial as other Entity Java applications, such as DT-Entity-Scenario AdaptiveCruiseControl, rely on critical environmental data to function effectively. Specifically, this Entity Java application handles tasks related to environment management, including setting and retrieving environment values.

The environmental factors considered in this implementation are, due to the scope of this thesis, limited to `roadType`, `groundStability`, and `visibility`, all of which are of type `Double`. These factors are the relevant ones for this prototype implementation and therefore collectively represent, `environmentSafety` as presented in [LK21]. These values are represented on a scale from 0.0 to 1.0, indicating the percentage of optimal quality for the associated factor, with 0.0 representing the worst condition and 1.0 representing the optimal condition.

Upon receiving a message with "setEnvValues" as the value for the JSON field `method`, the Java application Environment, updates the environment values (`roadType`, `groundStability`, and `visibility`) based on the contents of the incoming message. Subsequently, it sends a status message to the Data Manager to indicate that the task is completed, which is then logged in the Dashboard for user visibility.

DTMonitor also provides functionality to request and retrieve the current environment values of `roadType`, `groundStability`, and `visibility` via the `get method`. These values are sent back to the requester as separate JSON fields with their corresponding values.

This Environment Entity Java application manages and updates critical environmental data, ensuring that accurate and up-to-date information is available for other applications to utilize, thereby enhancing decision-making and adaptability (Req03, Req04, Req05, Req18, Req34).

6.4.10 DT-Entity DTMonitor

The correct interaction of multiple Entity Java applications is crucial, as they partially depend on being active simultaneously. Due to their loose coupling, this synchronized activity is not always the default state.
The Java application DTMonitor is responsible for periodically monitoring which Entity Java applications are active and forwarding this information to the Data Manager (Req33). To achieve this, each Entity Java application periodically sends predefined heartbeat messages to the DTMonitor Java application, using the JSON field `status` which can contain either "first heartbeat", "heartbeat", or "LWT disconnected".

Upon receiving the first heartbeat from an Entity application, the DTMonitor stores the sender as a key in `knownApplicationAndConnectedMap`, an instance of the `ConcurrentHashMap` class, along with a boolean value indicating whether the sending Entity Java application is currently active. Notably, `knownApplicationAndConnectedMap` only stores Entity Java applications that have sent at least one heartbeat message to DTMonitor.

DTMonitor utilizes another instance of the `ConcurrentHashMap` class, `lastHeartbeatMap`, to store each sender as a key along with the time of the last received heartbeat as the associated value. Consequently, when a heartbeat message is received, this time is updated in `lastHeartbeatMap`.

To check for missed heartbeats, DTMonitor uses a scheduled task. The `scheduler`, an instance of the `ScheduledExecutorService` class, checks every 10.000 milliseconds for each known heartbeat-sending Entity Java application in `lastHeartbeatMap` to determine if the current time exceeds the last received heartbeat time by more than the retransmission rate check interval (10.000 milliseconds). If this condition is met, DTMonitor sets the corresponding sender's value to false in `knownApplicationAndConnectedMap` and sends a message to the Data Manager. This message uses the field `status` with the value "disconnected" and the field `heartbeat` with the Entity Java application's name as the value, indicating that the message is a heartbeat.

However, it is not sufficient to rely solely on the detection of missed heartbeats to declare an Entity Java application as disconnected. Disconnections need to be detected immediately to ensure the reliability of the overall system. If an Entity Java application terminates and restarts before DTMonitor detects the shutdown (via receiving the "disconnected" message), this incident could be missed, leading to potential issues in the workflow. Therefore, each Entity Java application is configured with a LWT (see 6.1.4 Last Will and Testament (LWT)) to address this problem.

In cases where a Java application terminates unexpectedly and potentially in an uncontrolled manner, the LWT ensures that a message is sent to DTMonitor with the JSON field `status` and the value "LWT disconnected" upon termination. DTMonitor's last will is set to be sent to the Data Manager.

DTMonitor also provides functionality to request an array containing all active Entity Java applications being tracked, i.e. those that have sent at least one heartbeat message and are active at the time of the request.

DTMonitor itself sends a heartbeat to the Data Manager at the same retransmission rate. The initial heartbeat message indicates the start of the DTMonitor Java application (i.e. that it has begun running), while subsequent messages indicate that DTMonitor is actively running.

For this implementation, the retransmission rate of heartbeats for each Entity is set to 8.000 milliseconds, while the `scheduler` inside DTMonitor checks for missed heartbeats every 10.000 milliseconds. These values can be easily adjusted if needed for changes or future extensions.
Each heartbeat received by DTMonitor is converted to a suitable format for the Data Manager and forwarded accordingly. Specifically, the value "first heartbeat", "heartbeat", or "LWT disconnected" from the DTMonitor message field `status` is converted into the corresponding value "started", "running", or "disconnected" for the Data Manager field `status`. The field `heartbeat` in the Data Manager's message indicates that the message is a heartbeat and contains the name of the sending Entity Java application.

The DTMonitor Java application effectively ensures the synchronized and reliable functioning of various loosely coupled Entity Java applications (Req12, Req18, Req25, Req28, Req29).

6.4.11 General information on the DT-Entity-Scenario Java applications

The following scenarios (Req14, Req43) are derived and implemented as explicit separate DT-Entity Scenario Java applications:

- DT-Entity-Scenario ThreePointTurn
- DT-Entity-Scenario StopOnObstacle
- DT-Entity-Scenario AdaptiveCruiseControl
- DT-Entity-Scenario Overtaking

These Entity-Scenario Java applications will be elaborated upon in the following subsections. The following general principles apply to all DT-Entity-Scenario Java applications:

To utilize the hereafter mentioned Entity-Scenario Java applications, it is crucial that the vehicle to which the Entity-Scenario Java application will be applied:

(1) exists, i.e. is configured and possesses the basic capabilities, and

(2) has the relevant hardware components, such as sensors, necessary for the execution of the respective Entity-Scenario

To avoid repeating these checks during each execution to achieve e.g. lower latency times, the Entity-Scenario Java applications store the vehicles that are configured and equipped with the relevant hardware in their local runtime memory. This storage is in the form of an `ArrayList` or `HashMap`, depending on whether additional vehicle-related information needs to be stored with the vehicle identifier.

These checks are implicitly triggered by the Data Manager as soon as the respective Entity-Scenario Java applications are running, i.e. when the Data Manager receives a heartbeat message with the `status` "started" or "running" from the Java application.

If vehicle objects or a specific vehicle, including its configuration, are deleted from DT-Entity VehicleControl and consequently from the DT, the Data Manager instructs all known Entity-Scenario Java applications to remove the affected vehicles from their local runtime memory (`ArrayList` or `HashMap`) to maintain consistency within the DT.

Sensor data relevant to an Entity-Scenario is always sent to the respective Entity-Scenario Java application. The correct sensor(s) can be identified by having an array of sensor usage cases as the value for the JSON field `usagesInVehicle` and a string describing the sensor's attachment position in the vehicle as the value for the JSON field `attachmentOnVehicle`.

To ensure proper data flow, this incoming sensor data is processed only if the respective Entity-Scenario Java application is active, otherwise, the sensor data is discarded.

Special Entity-Scenario Java applications that require other Entity-Scenario Java applications as prerequisites, such as DT-Entity-Scenario Overtaking requiring DT-Entity-Scenario AdaptiveCruiseControl, also check during activation whether the necessary Entity-Scenario Java applications are active. If the required Entity-Scenario Java applications are not active or available, an error message is generated, indicating that the Entity-Scenario Java application cannot be executed.

6.4.12 DT-Entity-Scenario ThreePointTurn

The Java application ThreePointTurn is responsible for executing a basic three-point turn (Req03, Req05, Req18, Req39, Req40).
Upon receiving a message with "setTPTActive" as the value for the JSON field `method` and regarding a vehicle that has been identified as configured and containing the relevant hardware, the Java application ThreePointTurn processes the message.
First, it transmits a status message to the Data Manager to indicate that the task has been initiated, with this information documented in the Dashboard for user awareness. Subsequently, messages are sent out to cancel any potentially ongoing executions that conflict with the correct execution of this scenario (for further details, see Subsection 6.4.16). The actual scenario is executed through a hardcoded series of time-based commands that modify the vehicle's speed and angle at specific intervals. These commands are sent to DT-Entity VehicleControl, which updates the object values of the virtual replica of the PT in the DT and subsequently forwards the commands to the Data Manager (Req34). The Data Manager, in turn, transmits these commands to the PT. Finally, this Entity sends a status message to the Data Manager to indicate that the task is completed, which is then logged in the Dashboard for user visibility.
The aforementioned process will proceed as outlined unless a predefined trigger, such as another Entity detecting an obstacle in the vehicle's path using sensor data, interrupts and cancels the procedure.

6.4.13 DT-Entity-Scenario StopOnObstacle

The Java application StopOnObstacle is responsible for halting a vehicle's movement upon detecting an obstacle ahead (Req03, Req05, Req18, Req39, Req40). The threshold distance at which the vehicle reacts is adjustable and not fixed, and therefore needs to be set before this scenario can be activated. This threshold can be configured by sending a message with "setDistanceThreshold" as the value for the JSON field `method`, along with the respective threshold value, to the Java application StopOnObstacle. The initiation and completion of this threshold-setting task are documented by sending respective status messages to the Data Manager, with this information being recorded in the Dashboard for user visibility. Once the threshold has been established, the StopOnObstacle Entity-Scenario can be activated.

Upon then receiving a message with "setStopActive" as the value for the JSON field `method`, and "true" as the boolean value for the JSON field `booleanValue`, for a message regarding a vehicle that has been identified as configured and equipped with the relevant hardware, the Java application StopOnObstacle sets the state of this scenario to active if it was not already active. First, it transmits a status message to the Data Manager to indicate that the scenario activation has been initiated, with this information documented in the Dashboard for user awareness. Subsequently, messages are sent out to cancel any potentially ongoing executions that conflict with the correct execution of this scenario (for further details, see Subsection 6.4.16).

Next, the active state is stored in the runtime memory of the Java application StopOnObstacle, along with the respective vehicle identifier for the vehicle that this active state refers to. Finally, this Entity sends a status message to the Data Manager to indicate task completion, which is then logged in the Dashboard for user visibility.

From the moment the state for a vehicle has been set to active, incoming messages describing front distance data of the respective vehicle, captured by sensors and sent as messages with "newDistance" as the value for the JSON field `method`, are not discarded. Instead, these messages are processed by comparing each reported distance to the threshold set for the respective vehicle. The process entails evaluating the distance to an obstacle and making a determination on whether the vehicle should halt based on the information decoded from the received messages.

If the incoming distance is smaller than the preset threshold for the respective vehicle, a command to set the vehicle's speed to zero is sent to DT-Entity VehicleControl (Req34). DT-Entity VehicleControl then updates the object values of the virtual replica of the PT in the DT and further forwards the corresponding commands to the Data Manager. The Data Manager then transmits these commands to the PT. It is essential to note that once the vehicle has ceased movement due to the threshold being set higher than the measured front distance value, it will not autonomously resume motion even after the obstacle has moved out of range. The initiation of the vehicle's movement must be explicitly carried out by a DT Scenario, PT's functionality, or manually within either the DT or PT.

This scenario can be deactivated, either manually or by another Entity. In that case, the received message has "setStopActive" as the value for the JSON field `method`, and "false" as the boolean value for the JSON field `booleanValue`. If deactivation occurs, the state stored in the runtime memory of the Java application StopOnObstacle, along with the respective vehicle identifier of the vehicle that this state refers to, will be set to false if it was set to true before.

In case there is a change of state, also the initiation and completion of this task are documented by sending respective status messages to the Data Manager, with this information being recorded in the Dashboard for user visibility.
The cancel message (see 6.4.7) for StopOnObstacle equals such a deactivation message.

6.4.14 DT-Entity-Scenario AdaptiveCruiseControl

As discussed in Subsection 5.3.2, Traditional Cruise Control, also known as Cruise Control, involves automatically controlling the speed of a vehicle by maintaining a set speed determined by the driver [LK21]. Given that DT-Entity VehicleControl abstractly integrates acceleration and braking as setting a speed value, this implicitly covers (Traditional) Cruise Control.

The Java application AdaptiveCruiseControl implements a sophisticated form of Cruise Control, which dynamically adjusts the vehicle's speed to ensure a safe distance from preceding vehicles and obstacles, as illustrated in 6.12 (Req03, Req05, Req18, Req39, Req40).

For clarity within this thesis, the vehicle executing the AdaptiveCruiseControl scenario is referred to as the "following vehicle", a term that effectively conveys its role. In summary, the upper section of Figure 6.12 depicts a scenario in which the subsequent vehicle is still at an excessive distance from the preceding vehicle. Once the distance between the two vehicles falls within the operational range of the ACC system, the subsequent vehicle will then follow in a manner that dynamically adjusts its speed to align with that of the preceding vehicle, while maintaining a safe distance, as illustrated in the middle and lower sections of Figure 6.12.

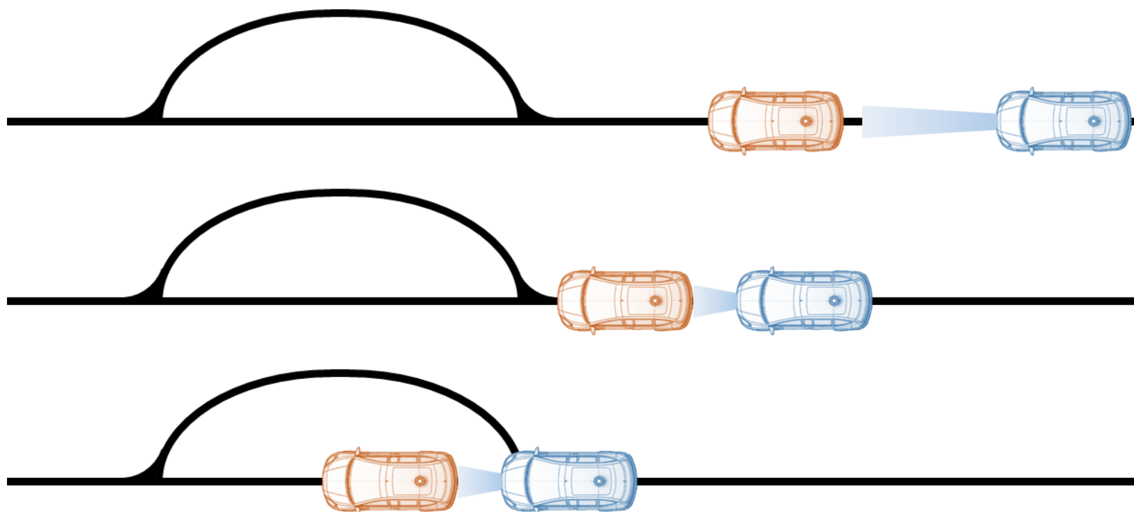

Figure 6.12: Different states of the AdaptiveCruiseControl scenario

After a vehicle has been identified as configured and equipped with the necessary hardware for this Entity-Scenario to function correctly, including the factors `brakeQuality` and `accelerationQuality` which collectively represent `vehicleQuality` as presented in [LK21], this Entity-Scenario further queries environmental information from DT-Entity Environment. This data is then stored along with the respective vehicle identifier in the local runtime memory of this Java application AdaptiveCruiseControl, as it is essential for calculating an appropriate velocity or speed.

Upon then receiving a message with "setAdaptiveCruiseControlActive" as the value for the JSON field `method`, and "true" as the boolean value for the JSON field `booleanValue`, for a message regarding a vehicle that has been identified as configured and equipped with the relevant hardware, the Java application AdaptiveCruiseControl sets the state of this scenario to active if it was not already active. First, it transmits a status message to the Data Manager to indicate that the scenario activation has been initiated, with this information documented in the Dashboard for user awareness. Subsequently, messages are sent out to cancel any potentially ongoing executions that conflict with the correct execution of this scenario (for further details, see Subsection 6.4.16).

Next, the active state is stored in the runtime memory of the Java application AdaptiveCruiseControl, along with the respective vehicle identifier for the vehicle to which this active state refers. Finally, this Entity sends a status message to the Data Manager to indicate task completion, which is then logged in the Dashboard for user visibility.

Once the state for a vehicle has been set to active, any incoming messages describing the front distance data of the respective vehicle, captured by sensors and sent as messages with "newDistance" as the value for the JSON field `method`, are not discarded. Instead, these messages are processed using the reported distance, along with the ranges and `vehicleQuality` factors associated with the respective vehicle and the `environmentSafety` factors, in order to calculate a speed value, as illustrated in Listing 6.2.

Listing 6.2 Java code to calculate vehicle speed based on given distance

```
double relativeDistance = (distanceToClosestMm - vehicleAdaptiveCruiseControlObject.getDistanceRange()
↪ [0]);
int adaptedSpeedInt;
if (relativeDistance < 0) { adaptedSpeedInt = 0; }
else if (relativeDistance == 0) { adaptedSpeedInt = 0; }
else { double normalizedDistance = relativeDistance /
       (vehicleAdaptiveCruiseControlObject.getDistanceRange()[1] -
       vehicleAdaptiveCruiseControlObject.getDistanceRange()[0]);
    if (normalizedDistance > 1.0) { normalizedDistance = 1.0; }
    double relativeMaxSpeed =
        (vehicleAdaptiveCruiseControlObject.getSpeedRange()[1]
        - vehicleAdaptiveCruiseControlObject.getSpeedRange()[0]);
    double relativeSpeed = normalizedDistance * relativeMaxSpeed *
        vehicleAdaptiveCruiseControlObject.getEnvironmentSafety() *
        vehicleAdaptiveCruiseControlObject.getVehicleQuality();
    double adaptedSpeedDouble = vehicleAdaptiveCruiseControlObject.getSpeedRange()[0] + relativeSpeed;
    adaptedSpeedInt = (int) adaptedSpeedDouble;
}
```

After computing the speed based on the received distance, the specified ranges, and the vehicleQuality and environmentSafety factors associated with the respective vehicle, a command to adjust the vehicle's speed accordingly is sent to DT-Entity VehicleControl (Req34). DT-Entity VehicleControl then updates the object values of the virtual replica of the PT in the DT and further forwards the corresponding commands to the Data Manager. The Data Manager then transmits these commands to the PT.

It is essential to note that in the implementation of the Java application AdaptiveCruiseControl presented in this thesis, the respective following vehicle does not reverse if the lead vehicle reverses. This limitation exists because the PT vehicles used in this thesis are not equipped with rear sensors. However, future work could incorporate this functionality to ensure that the following vehicle reverses in response to the lead vehicle's movement, provided that it is detected that the respective PT following vehicle is equipped with rear sensors. The identification of such sensors can be accomplished through JSON fields such as usagesInVehicle and attachmentOnVehicle, as mentioned in 6.4.11. Incorporating such an advancement would enhance both safety and functionality.

This scenario can be deactivated, either manually or by another Entity. In that case, the received message has "setAdaptiveCruiseControlActive" as the value for the JSON field method, and "false" as the boolean value for the JSON field booleanValue. If deactivation occurs, the state stored in the runtime memory of the Java application AdaptiveCruiseControl, along with the respective vehicle identifier of the vehicle that this state refers to, will be set to false if it was set to true before.

In case there is a change of state, also the initiation and completion of this task are documented by sending respective status messages to the Data Manager, with this information being recorded in the Dashboard for user visibility.
The cancel message (see 6.4.7) for AdaptiveCruiseControl equals such a deactivation message.

Furthermore, the values stored for a specific vehicle can be configured separately by sending a message to the Java application StopOnObstacle with "setObjectValues" as the value for the JSON field method. This message could include values for the factors speedRange and distanceRange. It could also include values for roadType, groundStability, and visibility, which collectively represent environmentSafety as mentioned in Subsection 6.4.9. Additionally, the message could include values for brakeQuality and accelerationQuality, which collectively represent vehicleQuality. It is not necessary to set all values, as individual factors can be configured as needed.

It can be observed that the above mentioned implementation of the Java application AdaptiveCruiseControl is, to some extent, covered by the PT functionality "Pause On Obstacle" of the Robot Cars employed as PT in this thesis. It is, however, important to note that the implementation of the Java application AdaptiveCruiseControl, in contrast to the PT's Robot Cars functionality "Pause On Obstacle", performs an actual speed adjustment based on the distance. This approach is significantly more precise than merely pausing and resuming movement.

6.4.15 DT-Entity-Scenario Overtaking

The Java application Overtaking executes an overtaking manoeuvre, as detailed in Subsection 5.3.2, by utilizing DT-Entity-Scenario AdaptiveCruiseControl and a bifurcation of the road track to employ a shorter route. This alternate path subsequently re-merges with the main road track after a specified distance. For clarity within this thesis, the vehicle being overtaken is referred to as the "lead vehicle", while the vehicle performing the overtaking is referred to as the "overtaking vehicle". These terms are straightforward and effectively convey the roles of the two vehicles in the scenario (Req03, Req05, Req18, Req39, Req40).

Considering the constraints of the PT in this thesis, it was necessary to predetermine certain aspects in the implementation. For instance, the road track is designed to include a bifurcation leading to a shorter route, which subsequently rejoins the main track after a specified distance. In addition, to implement the actual overtaking procedure, it was necessary to assign hardcoded speed values for both the overtaking vehicle and the lead vehicle, as well as a hardcoded `distanceRange` for the use of DT-Entity-Scenario AdaptiveCruiseControl, tailored specifically for this route.

Upon receiving a message with "setOvertakingActive" as the value for the JSON field `method` and regarding a vehicle that has been identified as configured and containing the relevant hardware, the Java application Overtaking processes the message. First, it transmits a status message to the Data Manager to indicate that the task has been initiated, with this information documented in the Dashboard for user awareness.

Afterwards, the Java application Overtaking issues a request to DT-Entity DTMonitor to verify that DT-Entity-Scenario AdaptiveCruiseControl is active, as Overtaking relies on it to accurately perform the manoeuvre as intended. If this condition is met, messages are dispatched to cancel any potentially ongoing executions that conflict with the correct execution of this scenario (for further details, see Subsection 6.4.16). Subsequently, the overtaking vehicle is set to follow the shorter road track at the next bifurcation.

At this stage, the overtaking vehicle might still be too far away from the lead vehicle, as illustrated in Figure 6.13. In order that an overtaking makes sense it needs to first catch up and then maintain close proximity before starting the overtaking procedure. Therefore, the Java application Overtaking activates DT-Entity-Scenario AdaptiveCruiseControl, with a predefined `distanceRange`, to catch up with the lead vehicle and if possible, maintain the minimum distance, as determined in this scenario by the Java application Overtaking and communicated in the activation message to DT-Entity-Scenario AdaptiveCruiseControl.

Figure 6.13: The overtaking vehicle, still at a distance, catching up with the lead vehicle

Once the associated DT-Entity-Scenario AdaptiveCruiseControl is activated, any incoming messages in the Java application Overtaking that detail the front distance data of the respective vehicle, captured by sensors and sent as messages with "newDistance" as the value for the JSON field `method`, are not discarded. Instead, these messages are processed using the reported distance to track when the overtaking vehicle has caught up with the lead vehicle.

Once the overtaking vehicle has caught up with the lead vehicle, as illustrated in Figure 6.14, this is flagged in association with the overtaking vehicle in the Java application Overtaking.

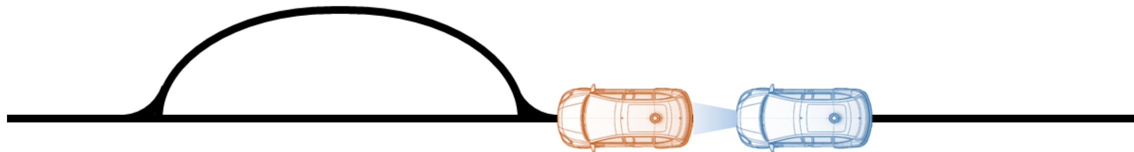

Figure 6.14: The overtaking vehicle has caught up with the lead vehicle

From this point onwards, it becomes evident that several conditions are met: DT-Entity-Scenario AdaptiveCruiseControl is active for the overtaking vehicle, the overtaking vehicle has caught up with the lead vehicle and maintains a short distance, and it is following the shorter road track at the next bifurcation.

When the maximum `distanceRange` value, set for DT-Entity-Scenario AdaptiveCruiseControl, is next exceeded, this indicates that the lead vehicle has moved aside due to the bifurcation, as illustrated in Figure 6.15. Consequently, this situation confirms that the actual overtaking procedure can commence.

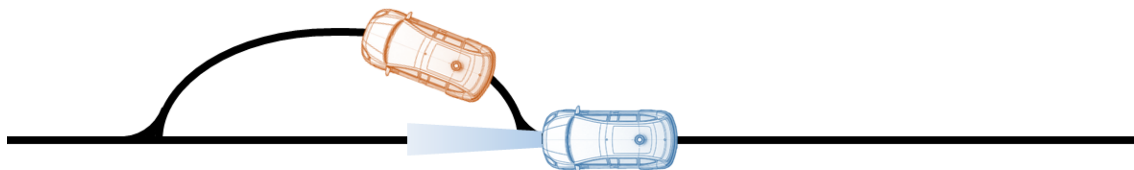

Figure 6.15: It is recognised that the lead vehicle has moved aside

To proceed with the actual overtaking procedure, a short time delay is introduced to ensure that the overtaking vehicle does not collide with the rear of the lead vehicle, as illustrated in Figure 6.16. Following this delay, the Java application Overtaking deactivates DT-Entity-Scenario AdaptiveCruiseControl for the overtaking vehicle and assigns a hardcoded increased speed for a fixed duration. Concurrently, and for the same duration, the Java application Overtaking assigns a reduced speed to the lead vehicle. These speed settings and durations have been tested and tailored to the PT used in this thesis.

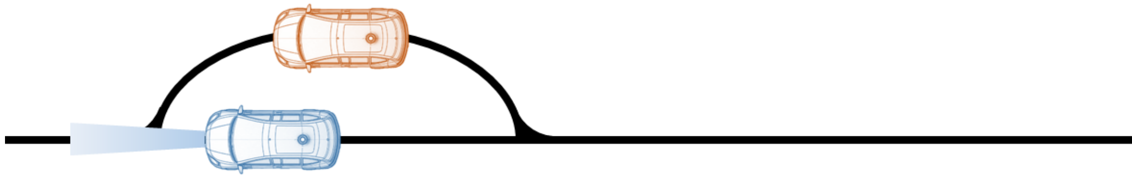

Figure 6.16: The overtaking vehicle did not collide (with the rear of) the lead vehicle

Upon the expiration of this duration, the speed values for both vehicles are reverted to their initial settings, and the overtaking vehicle is directed to follow the longer road track. Additionally, the PT's functionality "Pause On Obstacle" is activated for both vehicles to prevent potential collisions, either with each other or with any obstacles on the road track, as illustrated in Figure 6.17.

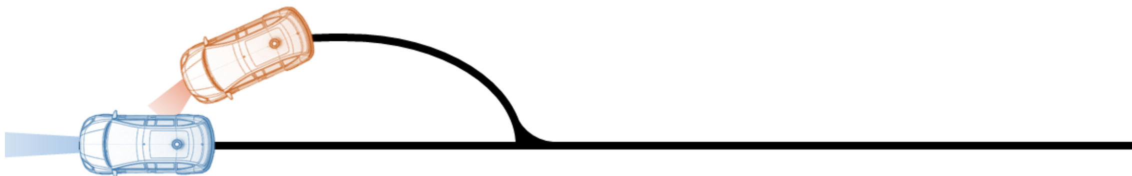

Figure 6.17: Both vehicles prevent potential collisions using "Pause On Obstacle" functionality

The previously mentioned commands of this Overtaking Scenario are sent to DT-Entity Vehicle-Control (Req34), which updates the object values of the virtual replica of the PT in the DT and subsequently forwards the commands to the Data Manager. The Data Manager, in turn, transmits these commands to the PT. Finally, this Entity sends a status message to the Data Manager to indicate that the task is completed, which is then logged in the Dashboard for user visibility.
The aforementioned process will proceed as outlined unless a predefined trigger, such as another Entity detecting an obstacle in the vehicle's path using sensor data, interrupts and cancels the procedure.

It can be observed that the implementation of the Java application Overtaking describes a simplified version of an overtaking process, constrained by the limitations of the PT used and the time bound available for this thesis. In this simplified version, several process steps are represented by hardcoded sequences. Nevertheless, this implementation is structured and designed in such a way that future extensions can be integrated with relative ease. This would reduce the necessity for hardcoded sequences, replacing them with more dynamic incoming data, such as sensor data, to the extent allowed by the hardware and software capabilities of the PT vehicles. For example, additional sensors, as previously mentioned in Subsection 6.4.11, can be identified through JSON fields such as `usagesInVehicle` and `attachmentOnVehicle`.

6.4.16 DT-Entities Conclusion

This prototype DT incorporates a feedback loop, that is, a continuous cycle of data exchange and interaction between the PT and DT. The data monitored from the PT is transmitted to the DT-Entities and DT-Entities-Scenarios, where it is analysed to plan the necessary actions. These actions are subsequently executed on the PT. The local runtime memories, as well as the appropriate databases, are utilized to store historical data and insights with a view to enhancing future decision-making processes (Req03, Req05, Req18, Req38, Req40).

Response and confirmation messages are sent where necessary. Furthermore, within the constraints of this thesis, error handling has been implemented in the prototype DT to capture errors and prevent uncontrolled code flows. Errors and individual steps are recorded on the DT Dashboard to keep the user informed about the DT's activities. This feedback mechanism helps avoid potential user errors when using the Dashboard by providing only the input options that are functional and appropriate at any given time.

Cases such as the sudden disconnection of a DT-Entity or DT-Entity-Scenario, potentially due to unexpected termination, are addressed by the DT-Entity DTMonitor recognising the Last Will and Testament (LWT) message and taking corresponding actions. In this thesis, any instance of an application disconnection is treated as if it was caused by a termination. While various potential causes for disconnections exist, including brief interruptions, these are not differentiated or addressed within this work's scope. Future research may explore these distinctions in greater detail.

Safeguards for the DT have been considered. For instance, if the Data Manager detects the disconnection of DT-Entity DTMonitor, it automatically initiates a disconnection action, triggering each DT-Entity to delete local information about existing vehicles and to disable all related Dashboard controls.

Having a feedback loop in a DT is advantageous because it enables timely monitoring and adaptive responses to changing conditions, thereby enhancing the system's efficiency and reliability. Additionally, the continuous exchange of data facilitates predictive maintenance and optimisation in future work, which are crucial for reducing downtime and operational costs (Req39, Req40).

DT-Entity Conflict Management

In the context of DTs for Autonomous Driving, managing conflicts from mutually exclusive tasks assigned to different DT-Entities or DT-Entities-Scenarios is crucial. Each Entity is designed for specific actions, and concurrent execution can lead to operational conflicts.

There are various approaches to conflict management. In this prototype DT, conflict resolution is implemented within all Entities due to the relatively few DT-Entities and DT-Entities-Scenarios involved. Although this requires each Entity to be aware of others to determine which scenarios to deactivate or interrupt, it reduces communication pathways compared to using a dedicated conflict management entity, which is significant in the context of Autonomous Driving.

The following outlines the conflict management strategies for the DT-Entities and DT-Entities-Scenarios in this prototype DT:

The PT's functionality "Pause on Obstacle" deactivates DT-Entity-Scenario StopOnObstacle and DT-Entity-Scenario AdaptiveCruiseControl, and cancels DT-Entity-Scenario Overtaking.

DT-Entity-Scenario StopOnObstacle halts the vehicle upon detecting an obstacle. Its activation requires the following actions:

- Deactivation of PT's functionality "Pause on Obstacle" to ensure a complete stop rather than a pause
- Deactivation of DT-Entity-Scenario AdaptiveCruiseControl to prevent the vehicle from resuming motion automatically after the obstacle is cleared
- Cancellation of DT-Entity-Scenario ThreePointTurn and DT-Entity-Scenario Overtaking as these manoeuvres cannot proceed with the halting mechanism of DT-Entity-Scenario StopOnObstacle engaged

DT-Entity-Scenario ThreePointTurn is responsible for executing a basic three-point turn. Consequently, its activation necessitates the following deactivations or cancellations:

- VehicleControl to deactivate PT's functionality "Stay on Track" as the vehicle cannot perform a three-point turn if its motion is restricted to the road track
- DT-Entity-Scenario AdaptiveCruiseControl, since the three-point turn is executed through a hardcoded series of time-based commands, which would be interfered with by commands from DT-Entity-Scenario AdaptiveCruiseControl, as those are based on distance values
- DT-Entity-Scenario Overtaking, as the targeted manoeuvres differ

However, the activation of DT-Entity-Scenario ThreePointTurn should not deactivate DT-Entity-Scenario StopOnObstacle and PT's functionality "Pause on Obstacle" if either is active, as the vehicle should halt during the three-point turn manoeuvre if an obstacle is encountered. If this is not desired, the user should manually deactivate DT-Entity-Scenario StopOnObstacle or PT's functionality "Pause on Obstacle" before executing DT-Entity-Scenario ThreePointTurn, depending which of both is active.

DT-Entity-Scenario AdaptiveCruiseControl dynamically adjusts the vehicle's speed to ensure a safe distance from preceding vehicles and obstacles. Consequently, its activation necessitates the following deactivations or cancellations:

- DT-Entity-Scenario StopOnObstacle and VehicleControl to deactivate PT's functionality "Pause on Obstacle" as DT-Entity-Scenario AdaptiveCruiseControl follows its own logic to decide when to halt
- DT-Entity-Scenario ThreePointTurn, as it has a conflicting execution goal
- DT-Entity-Scenario Overtaking, as it has a conflicting execution goal unless the activation of DT-Entity-Scenario AdaptiveCruiseControl was initiated by DT-Entity-Scenario Overtaking, implying that DT-Entity-Scenario Overtaking relies on DT-Entity-Scenario AdaptiveCruiseControl

DT-Entity-Scenario Overtaking necessitates the deactivations/cancellations of DT-Entity-Scenario StopOnObstacle, DT-Entity-Scenario ThreePointTurn, DT-Entity-Scenario AdaptiveCruiseControl and PT's functionality "Pause on Obstacle" to prevent any conflicting actions during the manoeuvre. Nevertheless, DT-Entity-Scenario Overtaking relies on and thus activates DT-Entity-Scenario AdaptiveCruiseControl and PT's functionality "Pause on Obstacle" when necessary.

Manually set speed adjustments by the user in the Dashboard or speed adjustments incoming from the PT vehicle to the DT cancel/deactivate the execution of DT-Entity-Scenario ThreePointTurn, DT-Entity-Scenario AdaptiveCruiseControl, and DT-Entity-Scenario Overtaking as these always have priority. The same applies to a halt initiated by DT-Entity-Scenario StopOnObstacle, i.e. speed value zero.

These conflict management strategies offer a systematic approach to handling interactions and conflicts within the Java applications, ensuring accurate execution and the safe and efficient management of the DT-Entities and DT-Entities-Scenarios.

In future work, as additional DT-Entities and DT-Entities-Scenarios are added, implementing a centralized, dedicated DT-entity for conflict management will be prudent. This was not feasible within the scope of this prototype DT but will enhance scalability and flexibility (Req25) to handle the increasing number of DT-Entities and DT-Entities-Scenarios and their potential conflicts. The conceptual approach already allows for this. As new DT-Entities or scenarios are integrated, the conflict management protocols can be adapted or extended without disrupting the existing operational framework. This modular approach ensures that the Digital Twin can evolve and expand while maintaining cohesive and conflict-free functionality.
A centralized DT-entity for conflict management would serve as an overarching arbiter, harmonizing the actions of various DT-Entities and DT-Entities-Scenarios. This Entity would evaluate potential conflicts, applying predefined rules and possibly optimization algorithms to ensure task execution is coordinated efficiently. This approach minimizes the risk of operational inefficiencies or failures arising from uncoordinated actions. By integrating such a dedicated entity, the overall system's robustness and reliability are enhanced, leading to more seamless and effective autonomous operations. Nonetheless, this approach will inherently extend communication pathways, a consideration that must be acknowledged and accepted.

DT-PT Synchronization

Ensuring synchronisation between the Digital Twin and the Physical Twin is paramount for maintaining temporal accuracy and consistency, which is essential for effective monitoring, analysis, decision-making and timely processing (Req15, Req16, Req18, Req28).

In this prototype DT, the Data Manager oversees synchronisation by using global variables to differentiate between new and old values for elements that can change in both the PT and DT. This approach prevents redundant updates, ensuring only the most recent changes are reflected.

The bidirectional synchronisation logic identifies the source of each new value, whether it originates from the PT, DT, or is manually set in the DT Dashboard. The new value is then propagated and updated accordingly, with newer values always overwriting older ones due to the sequential update method implemented within the scope of this thesis.

Best practices for synchronisation, such as State Management & Versioning and Event Filtering, were considered. However, these methods were found too complex to implement within the given scope using Node-RED as the Data Manager. Event Filtering was unsuitable due to latency issues, as querying the other twin before setting a value is time-consuming. Implementing State Management and Versioning in Node-RED was also complex due to the Dashboard nodes and not feasible within the thesis's scope, though this could be explored in future work.

Non-Functional Properties Applicable to the DT-Entities in this Implementation

It seems that certain non-functional properties are applicable to the prototype DT of this thesis.

The DT-Entities presented in this section should be capable of handling an unspecified (but limited) number of vehicles in parallel. Furthermore, the addition of new DT-Entities in the system should be achievable with a relatively minimal expenditure of effort, indicating a certain degree of scalability. However, due to the bounded scope, this Digital Twin prototype has been initially implemented for the described scenarios and a fixed small number of vehicles. Therefore, no evaluations regarding upper limits on the number of vehicles have been conducted. This could be addressed in future work to ensure the system can support larger fleets and more complex Entities and scenarios over time as it grows.

Autonomous Driving requires a high level of safety and efficiency. Obviously, in this prototype this cannot be fully implemented as it would require high performance for timely (ideally real-time) data processing and decision-making, ensuring that all actions and reactions are executed with acceptable delays. Furthermore, it must be noted that performance is highly dependent on the speed of the algorithms and the platform on which they run.

Additionally, the interactions should be reliable and should operate with a high degree of operational robustness, particularly in critical scenarios like obstacle detection and Adaptive Cruise Control, also to ensure the safety of vehicles and passengers. This seems feasible as long as the Entities and all the elements (e.g. PT, Data Manager, etc.) are running in a duly and timely manner and the communication paths are intact. Moreover, the system needs to be operational at all times, as it involves the continuous monitoring and control of vehicles. In this prototype, this is partially fulfilled, as the system's failure is immediately detectable through monitoring, and also prompting emergency measures. However, due to the bounded scope, automatic restart and backup functionalities were not implemented. This would be a significantly relevant aspect for future work, particularly if the system is scaled up.

Moreover, the system should be straightforward to maintain and update. However, this has not been a point of focus in this thesis. Addressing this, especially with regard to enabling maintenance without downtime, could be a potential avenue for future research.

The system needs to be generic with regard to interoperability with various hardware and software components, including different types of vehicles, sensors, and external systems like dashboards and databases. It should boast good usability, featuring a comprehensive and user-friendly interface. It should also accommodate for future advancements without major rework. The system has been designed with the intention of being flexible (Req25) and extendable (Req24, Req37) and to facilitate the straightforward incorporation of modifications and new features/modules.

6.5 Illustrative Scenarios with the prototype Digital Twin

Given the time frame as well as the capabilities and limitations of the PT, a prototype Digital Twin has been implemented.
Figure 6.18 shows the prototype DT with the short paper track and the two Robot Cars (in the upper part of the picture) and the Dashboard (inserted in the lower part of the picture).

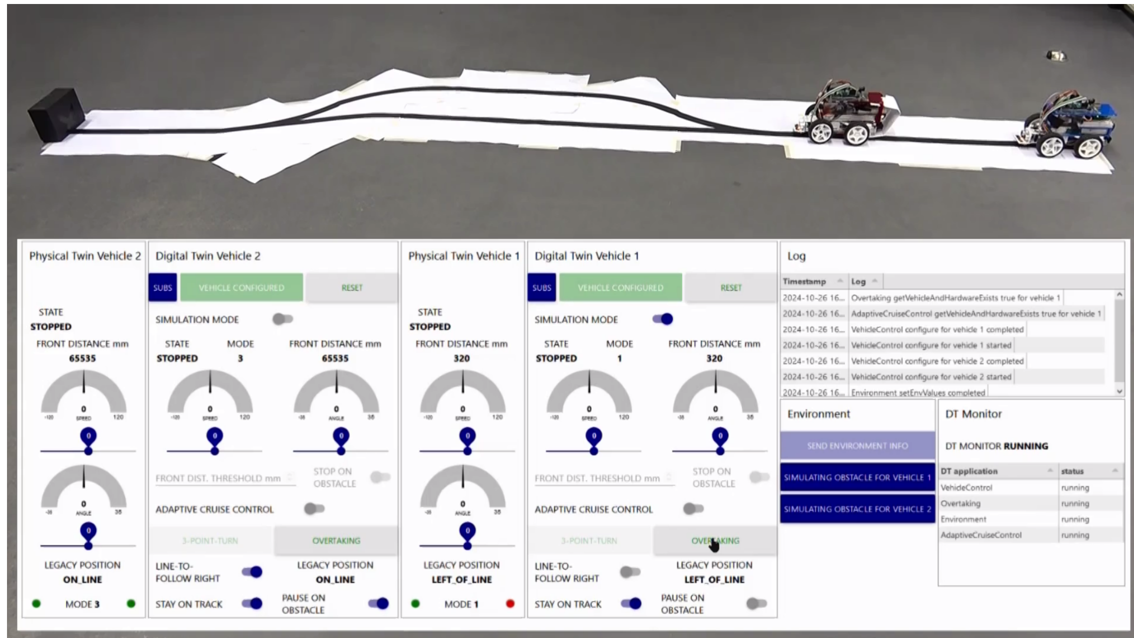

Figure 6.18: Prototype DT with 2 RCs and Dashboard

The following illustrative scenarios have been implemented and tested with the objective of examining the flexibility, composability and extendability of the system, as well as identifying areas requiring further attention, and they have been documented in a video[3]:

6.5.1 driving ON TRACK (shadowing)

6.5.2 driving ON TRACK + PAUSE ON OBSTACLE (shadowing)

6.5.3 driving ON TRACK + STOP ON OBSTACLE

6.5.4 driving ON TRACK + STOP ON VIRTUAL OBSTACLE

6.5.5 driving ON TRACK + ADAPTIVE CRUISE CONTROL + PAUSE ON OBSTACLE

6.5.6 driving ON TRACK + OVERTAKING + PAUSE ON OBSTACLE

6.5.7 driving ON TRACK + THREE POINT TURN + PAUSE ON OBSTACLE

The following subsections present the aforementioned scenarios, each including a respective scenario description, the scope and structure as well as a process description in the form of a table.

[3]Video: Illustrative Scenarios for the Implementation of a Digital Twin for Autonomous Driving

6.5.1 Scenario: driving ON TRACK (shadowing)

Scenario description:
the vehicle drives on the selected track (left or right line). It would inevitably result in a collision with any obstacle that might be encountered, as no preventative measure has been activated.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 1
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC positioned on track and ON
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC	press "SUBS" in DT-DSH VEH
configure RC	press "CONFIG" in DT-DSH VEH
-> reaction in DT-DSH VEH	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH
PAUSE ON OBSTACLE off	deactivate "PAUSE ON OBSTACLE"
SIMULATION MODE off	deactivate "SIMULATION MODE" in DT-DSH VEH
verify that car is on the track	verify in PT-DSH and in DT-DSH that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH

Process description (cont.):	
STAY ON TRACK on	activate "STAY ON TRACK" in DT-DSH VEH
-> reactions of RC	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right or left road path	activate or deactivate "LINE TO FOLLOW RIGHT" in DT-DSH VEH
start and determine speed RC	use speed slider in PT-DSH VEH
-> reactions of RC	drives and follows the selected road path and it does correctly collide with the obstacle at the end
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	shadowing can be seen in the DT-DSH

Table 6.2: Process description for Scenario: driving ON TRACK (shadowing)

6.5.2 Scenario: driving ON TRACK + PAUSE ON OBSTACLE (shadowing)

Scenario description:
both vehicles drive on the selected track (left or right line). In the event of an obstacle being encountered, both Robot Cars are programmed to halt until such time as the obstacle is no longer present. In such a case it/they would automatically resume its/their normal driving pattern.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 2
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
  - pausing when an obstacle occurs (at a preset distance defined in the PT) and resume driving automatically when possible

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC 1 positioned on track and ON and RC 2 positioned on track and ON
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC 1	press "SUBS" in DT-DSH VEH 1
initiate subscription RC 2	press "SUBS" in DT-DSH VEH 2
configure RC 1	press "CONFIG" in DT-DSH VEH 1
-> reaction in DT-DSH VEH 1	wait for "VEHICLE CONFIGURED"
configure RC 2	press "CONFIG" in DT-DSH VEH 2
-> reaction in DT-DSH VEH 2	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH, neither for VEH 1 nor VEH 2
PAUSE ON OBSTACLE off	deactivate "PAUSE ON OBSTACLE"
SIMULATION MODE off RC 1	deactivate "SIMULATION MODE" in DT-DSH VEH 1
SIMULATION MODE off RC 2	deactivate "SIMULATION MODE" in DT-DSH VEH 2

Process description (cont.):	
verify that RC 1 is on the track	verify in PT-DSH VEH 1 and in DT-DSH VEH 1 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify that RC 2 is on the track	verify in PT-DSH VEH 1 and in DT-DSH VEH 2 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH
STAY ON TRACK on RC 1	activate "STAY ON TRACK" in DT-DSH VEH 1
-> reactions of RC 1	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 1	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 1	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
STAY ON TRACK RC 2 on	activate "STAY ON TRACK" in DT-DSH VEH 2
-> reactions of RC 2	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 2	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 2	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right or left road path	activate or deactivate "LINE TO FOLLOW RIGHT" in DT-DSH VEH 1 and in DT-DSH of VEH 2 (road path should be same for both vehicles in this scenario)
PAUSE ON OBSTACLE on RC1	activate "PAUSE ON OBSTACLE" in DT-DSH VEH 1 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH 1	"MODE" changes from "1" to "3"
-> reactions in PT-DSH VEH 1	"MODE 1" changes in "MODE 3" and the MODE color on the left and right sides changes red to green
PAUSE ON OBSTACLE on RC2	activate "PAUSE ON OBSTACLE" in DT-DSH VEH 2 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH 2	"MODE" changes from "1" to "3"
-> reactions in PT-DSH VEH 2	"MODE 1" changes in "MODE 3" and the MODE color on the left and right sides changes red to green

Process description (cont.):	
start and determine speed RC 1	use speed slider in PT-DSH VEH 1
-> reactions of RC 1	VEH 1 drives and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 1	shadowing can be seen in the DT-DSH 1
after RC pauses before colliding with RC 2 the first time:	
start and determine speed RC 2	use speed slider in PT-DSH VEH 2
-> reactions of RC 2	VEH 2 drives and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 2	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 2	shadowing can be seen in the DT-DSH 2
-> reactions of RC 1	resumes driving (as soon as the front vehicle is further away than the threshold) and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 1	shadowing can be seen in the DT-DSH 1

Table 6.3: Process description for Scenario:
driving ON TRACK + PAUSE ON OBSTACLE (shadowing)

6.5.3 Scenario: driving ON TRACK + STOP ON OBSTACLE

Scenario description:
the vehicle drives on the selected track (left or right line). In the event of an obstacle being encountered, the DT of the vehicle would induce it to stop. In consideration of safety concerns, the simulation "STOP ON OBSTACLE" is designed in a manner that precludes the automatic restart after the STOPing, necessitating instead a manual action on the part of the driver.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 1
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
- the simulation case STOPPING on obstacle is implemented in the Digital Twin and the relevant commands for that are set and executed in the DT

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC positioned on track and ON
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC	press "SUBS" in DT-DSH VEH
configure RC	press "CONFIG" in DT-DSH VEH
-> reaction in DT-DSH VEH	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH
PAUSE ON OBSTACLE off	deactivate "PAUSE ON OBSTACLE"
SIMULATION MODE on	activate "SIMULATION MODE" in DT-DSH VEH
verify that RC is on the track	verify in PT-DSH VEH and in DT-DSH VEH that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH

Process description (cont.):	
STAY ON TRACK on	activate "STAY ON TRACK" in DT-DSH VEH
-> reactions of RC	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right or left road path	activate or deactivate "LINE TO FOLLOW RIGHT" in DT-DSH VEH
set FRONT DISTANCE threshold	enter value for "FRONT DIST THRESHOLD mm" in DT-DSH VEH
STOP ON OBSTACLE on	activate "STOP ON OBSTACLE" in DT-DSH VEH (Note: it deactivates automatically "PAUSE ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING" as well as "3-POINT-TURN")
start and determine speed RC	use speed slider in PT-DSH VEH
-> reactions of RC	drives and follows the selected road path and stops before colliding
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position

Table 6.4: Process description for Scenario: driving ON TRACK + STOP ON OBSTACLE

6.5.4 Scenario: driving ON TRACK + STOP ON VIRTUAL OBSTACLE

Scenario description:
the vehicle drives on the selected track (left or right line). In the event of an obstacle being encountered, it is programmed to halt until such time as the obstacle is no longer present. In such a case it would automatically resume its normal driving pattern ("function PAUSE ON OBSTACLE"). In this scenario, the DT demonstrates the potential of simulating with a VIRTUAL OBSTACLE. In consideration of safety concerns, the simulation "STOP ON OBSTACLE" is designed in a manner that precludes the automatic restart after the STOPing, necessitating instead a manual action on the part of the driver.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 1
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
  - pausing when an obstacle occurs (at a preset distance defined in the PT) and resume driving automatically when possible
- the simulation case STOPPING on obstacle is implemented in the Digital Twin and the relevant commands for that are set and executed in the DT
- the VIRTUAL obstacle is simulated in the Digital Twin via a button in the DT's Environment-Dashboard

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC positioned on track and ON
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC	press "SUBS" in DT-DSH VEH
configure RC	press "CONFIG" in DT-DSH VEH
-> reaction in DT-DSH VEH	wait for "VEHICLE CONFIGURED"
PAUSE ON OBSTACLE off	deactivate "PAUSE ON OBSTACLE"
SIMULATION MODE on	activate "SIMULATION MODE" in DT-DSH VEH

Process description (cont.):	
verify that RC is on the track	verify in PT-DSH VEH and in DT-DSH VEH that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH
STAY ON TRACK on	activate "STAY ON TRACK" in DT-DSH VEH
-> reactions of RC	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right or left road path	activate or deactivate "LINE TO FOLLOW RIGHT" in DT-DSH VEH
set FRONT DISTANCE threshold	enter value for "FRONT DIST THRESHOLD mm" in DT-DSH VEH
STOP ON OBSTACLE on	activate "STOP ON OBSTACLE" in DT-DSH VEH (Note: it deactivates automatically "PAUSE ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING" as well as "3-POINT-TURN")
start and determine speed RC	use speed slider in PT-DSH VEH
-> reactions of RC	drives and follows the selected road path and stops before colliding with any (virtual) obstacle when necessary
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
place a virtual obstacle	press "SIMULATING OBSTACLE FOR VEHICLE" (for the relevant VEH) in ENVIRONMENT-DSH
-> reactions of RC	stops before colliding with the virtual obstacle
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position

Process description (cont.):	
after the RC stopped before colliding with the virtual obstacle:	
PAUSE ON OBSTACLE on	activate "PAUSE ON OBSTACLE" in DT-DSH VEH (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH	"MODE" changes from "1" to "3"
-> reactions in PT-DSH VEH	"MODE 1" changes in "MODE 3" and the MODE color on the left and right sides changes red to green
restart and determine speed RC	use speed slider in DT-DSH VEH
-> reactions of RC	drives and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position

Table 6.5: Process description for Scenario:
driving ON TRACK + STOP ON VIRTUAL OBSTACLE

6.5.5 Scenario: driving ON TRACK + ADAPTIVE CRUISE CONTROL + PAUSE ON OBSTACLE for lead car

Scenario description:
both vehicles drive on the selected track (left or right line). In the event of an obstacle being encountered, the front Robot Car is programmed to halt until such time as the obstacle is no longer present. In this case, it automatically resumes its normal driving pattern. The second vehicle maintains a distance from the front vehicle within a specified range and adjusts its speed accordingly, stopping when necessary.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 2
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
  - pausing when an obstacle occurs (at a preset distance defined in the PT) and resume driving automatically when possible
- the simulation case ADAPTIVE CRUISE CONTROL is implemented in the Digital Twin and the relevant commands for that are set and executed in the DT

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC 1 positioned on track and ON and RC 2 positioned on track and ON RC 2 to be positioned at a distance greater than the minimum defined range (in DT). This distance must possibly be longer to offset delays caused by slow process reaction times.
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC 1	press "SUBS" in DT-DSH VEH 1
initiate subscription RC 2	press "SUBS" in DT-DSH VEH 2

Process description (cont.):	
configure RC 1	press "CONFIG" in DT-DSH VEH 1
-> reaction in DT-DSH VEH 1	wait for "VEHICLE CONFIGURED"
configure RC 2	press "CONFIG" in DT-DSH VEH 2
-> reaction in DT-DSH VEH 2	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH, neither for VEH 1 nor VEH 2
SIMULATION MODE on RC 1	activate "SIMULATION MODE" in DT-DSH VEH 1
SIMULATION MODE off RC 2	deactivate "SIMULATION MODE" in DT-DSH VEH 2
verify that RC 1 is on the track	verify in PT-DSH VEH 1 and in DT-DSH VEH 1 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify that RC 2 is on the track	verify in PT-DSH VEH 2 and in DT-DSH VEH 2 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH
verify distance RC1-RC2	RC 2 needs to be positioned far enough in front of RC 1 (for details see initial situation)
STAY ON TRACK on RC 1	activate "STAY ON TRACK" in DT-DSH VEH 1
-> reactions of RC 1	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 1	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 1	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
STAY ON TRACK RC 2 on	activate "STAY ON TRACK" in DT-DSH VEH 2
-> reactions of RC 2	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 2	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 2	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right or left road path	activate or deactivate "LINE TO FOLLOW RIGHT" in DT-DSH VEH 1 and in DT-DSH of VEH 2 (road path should be same for both vehicles in this scenario)

Process description (cont.):	
PAUSE ON OBSTACLE off RC1	deactivate "PAUSE ON OBSTACLE" in DT-DSH VEH 1 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
PAUSE ON OBSTACLE on RC2	activate "PAUSE ON OBSTACLE" in DT-DSH VEH 2 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH 2	"MODE" changes from "1" to "3"
-> reactions in PT-DSH VEH 2	"MODE 1" changes in "MODE 3" and the MODE color on the left and right sides changes red to green
start ADAPTIVE CRUISE CONTROL	activate "ADAPTIVE CRUISE CONTROL" in DT-DSH VEH 1 (Note: it deactivates automatically "PAUSE ON OBSTACLE" and "STOP ON OBSTACLE" and cancels "3-POINT-TURN" as well as "OVERTAKING")
-> reactions of RC 1	drives and on the selected road path and follows the front car (within a defined distance range), thereby adapting its speed (within a defined speed range) accordingly and halts if need be
-> reactions in PT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions of RC 2	drives and on the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 2	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 2	visualisation of state, speed, angle, distance to obstacles and position

Table 6.6: Process description for Scenario:
driving ON TRACK + ADAPTIVE CRUISE CONTROL + PAUSE ON OBSTACLE for lead car

6.5.6 Scenario: driving ON TRACK + OVERTAKING + PAUSE ON OBSTACLE

Scenario description:
in consideration of the configuration of the available track, which features a single lane, the overtaking scenario is designed in a manner that the front vehicle deviates onto a side road section. Consequently, it covers a greater distance than the other vehicle on the direct route. Both vehicles drive and stay on track. The front vehicle deviates onto a side road section (the command "LINE-TO-FOLLOW RIGHT" is active). The second vehicle continues on the main route (the command "LINE-TO-FOLLOW RIGHT" is deactivated). The second vehicle is capable of overtaking the front one, as it covers a shorter distance. At the end of the junction of the two roads, the vehicles need to avoid a collision. In the event of an obstacle being encountered, both Robot Cars are programmed to halt until such time as the obstacle is no longer present. This is achieved through the activation of the "PAUSE ON OBSTACLE" function. Once the obstacle has been removed, the Robot Cars will automatically resume their normal driving pattern.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 2
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
  - pausing when an obstacle occurs (at a preset distance defined in the PT)
    and resume driving automatically when possible
- the simulation case OVERTAKING is implemented in the Digital Twin and the relevant commands for that are set and executed in the DT
- he OVERTAKING makes use of and depends on the ADAPTIVE CRUISE CONTROL (both designed in the DT) to ascertain whether it is possible and safe to accelerate and overtake. Overtaking is based on a continuous assessment of distance measurements with the use of designated flags

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)

Process description (cont.):	
initial situation	RC 1 positioned on track and ON and RC 2 positioned on track and ON RC 2 to be positioned at a distance greater than the minimum defined range (in DT). This distance must possibly be longer to offset delays caused by slow process reaction times.
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC 1	press "SUBS" in DT-DSH VEH 1
initiate subscription RC 2	press "SUBS" in DT-DSH VEH 2
configure RC 1	press "CONFIG" in DT-DSH VEH 1
-> reaction in DT-DSH VEH 1	wait for "VEHICLE CONFIGURED"
configure RC 2	press "CONFIG" in DT-DSH VEH 2
-> reaction in DT-DSH VEH 2	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH, neither for VEH 1 nor VEH 2
SIMULATION MODE on RC 1	activate "SIMULATION MODE" in DT-DSH VEH 1
SIMULATION MODE off RC 2	deactivate "SIMULATION MODE" in DT-DSH VEH 2
verify that RC 1 is on the track	verify in PT-DSH VEH 1 and in DT-DSH VEH 1 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify that RC 2 is on the track	verify in PT-DSH VEH 1 and in DT-DSH VEH 2 that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH
verify distance RC1-RC2	RC 2 needs to be positioned far enough in front of RC 1 (for details see initial situation)

Process description (cont.):	
STAY ON TRACK on RC 1	activate "STAY ON TRACK" in DT-DSH VEH 1
-> reactions of RC 1	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 1	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 1	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
STAY ON TRACK RC 2 on	activate "STAY ON TRACK" in DT-DSH VEH 2
-> reactions of RC 2	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH 2	"MODE" changes from "0" to "1"
-> reactions in PT-DSH VEH 2	"MODE 0" changes to "MODE 1" and the MODE color on the left side changes from red to green and angle adjustments are visualised in the PT-DSH
select right road path VEH 1	activate "LINE TO FOLLOW RIGHT" in DT-DSH VEH 1 and in DT-DSH of VEH 1 (both VEH need to be set at the beginning for the longest road)
select right road path VEH 2	activate "LINE TO FOLLOW RIGHT" in DT-DSH VEH 2 and in DT-DSH of VEH 2 (both VEH need to be set at the beginning for the longest road)
PAUSE ON OBSTACLE off RC1	deactivate "PAUSE ON OBSTACLE" in DT-DSH VEH 1 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
PAUSE ON OBSTACLE on RC2	activate "PAUSE ON OBSTACLE" in DT-DSH VEH 2 (Note: it deactivates automatically in VEH 1 "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH 2	"MODE" changes from "1" to "3"
-> reactions in PT-DSH VEH 2	"MODE 1" changes in "MODE 3" and the MODE color on the left and right sides changes red to green

Process description (cont.):	
start OVERTAKING	press "OVERTAKING" in DT-DSH VEH 1 (Note: it it activates automatically "ADAPTIVE CRUISE CONTROL")
-> reactions of RC 1	drives and follows the selected road path follows the front car (within a defined distance range), thereby adapting its speed (within a defined speed range) accordingly and halts if need be. Once the front/lead vehicle (VEH 2) has left the main road, VEH 1 is capable of overtaking it as it covers a shorter distance. At an adequate point, "PAUSE ON OBSTACLE" in VEH 1 is activated automatically. VEH 1 then is able to pause before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 1	visualisation of state, speed, angle, distance to obstacles and position
-> reactions of RC 2	drives and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH 2	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH 2	visualisation of state, speed, angle, distance to obstacles and position

Table 6.7: Process description for Scenario:
driving ON TRACK + OVERTAKING + PAUSE ON OBSTACLE

6.5.7 Scenario: driving ON TRACK + THREE POINT TURN + PAUSE ON OBSTACLE

Scenario description:
the DT is designed to enable the vehicle to perform a "3-POINT-TURN" automatically. Subsequently, once the road path is selected (left or right line) and "STAY ON TRACK" is activated and speed is set manually, the vehicle will drive accordingly. In the event of an obstacle being encountered, the command "PAUSE ON OBSTACLE" would cause the vehicle to halt until such time as the obstacle is no longer present. Thereafter, the vehicle would automatically resume its normal driving pattern.

Scenario scope and structure:

- amount of vehicle / Robot Car used: 1
- track: single lane, plane, road with a side road section (also single lane)
- physical obstacle on the road
- the DT shadows the PT's functionalities:
  - driving (steering, speed, etc.)
  - staying on track
  - pausing when an obstacle occurs (at a preset distance defined in the PT) and resume driving automatically when possible
- the simulation case THREE POINT TURN is implemented in the Digital Twin and the relevant commands for that are set and executed in the DT

Process description:	
Step	**Actions and Comments**
start the DT	verify in "DT Monitor"-DSH that status is "STARTED" / "RUNNING" and the status of the "DT application" in "DT Monitor"-DSH
log information	Log information can be seen in the Log-DSH in real-time (they are also stored in a database with timestamp)
initial situation	RC positioned on track and ON
establish connection DT to PT	WiFi connection and authentication, etc.
initiate subscription RC	press "SUBS" in DT-DSH VEH
configure RC	press "CONFIG" in DT-DSH VEH
-> reaction in DT-DSH VEH	wait for "VEHICLE CONFIGURED"
SIMULATING OBSTACLE off	do not press "SIMULATING OBSTACLE FOR VEHICLE" in ENVIRONMENT-DSH
SIMULATION MODE on	activate "SIMULATION MODE" in DT-DSH VEH

Process description (cont.):	
verify that RC is on the track	verify in PT-DSH VEH and in DT-DSH VEH that "LEGACY POSITION" is not "LINE_POSITION_UNKNOWN" (ideally it should be "ON_LINE")
verify presence of obstacle	check "FRONT DISTANCE" values shown in the PT-DSH and DT-DSH
PAUSE ON OBSTACLE on	activate "PAUSE ON OBSTACLE" in DT-DSH VEH (Note: it deactivates automatically "STOP ON OBSTACLE" and "ADAPTIVE CRUISE CONTROL" and cancels "OVERTAKING")
-> reactions in DT-DSH VEH	"MODE" changes from "0" to "2"
-> reactions in PT-DSH VEH	"MODE 0" changes in "MODE 2" and the MODE color on the right side changes from red to green
3-POINT-TURN on	press "3-POINT-TURN" in DT-DSH VEH
-> reactions of RC	drives and performs a three-point-turn
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
after the RC made the 3-point-turn, ideally:	
STAY ON TRACK on	activate "STAY ON TRACK" in DT-DSH VEH
-> reactions of RC	the steering wheels adjust accordingly
-> reactions in DT-DSH VEH	"MODE" changes from "2" to "3" visualisation of state, speed, angle, distance to obstacles and position
-> reactions in PT-DSH VEH	"MODE 2" changes to "MODE 3" and the MODE color on the left side changes from red to green visualisation of state, speed, angle, distance to obstacles and position
restart and determine speed	use speed slider in PT-DSH VEH
-> reactions of RC	VEH drives and follows the selected road path and pauses before colliding with any obstacle, until such time as the obstacle is no longer present and it would then automatically resume its normal driving pattern
-> reactions in PT-DSH VEH	visualisation of state, speed, angle, distance to obstacles and position
-> reactions in DT-DSH VEH	shadowing can be seen in the DT-DSH

Table 6.8: Process description for Scenario:
driving ON TRACK + THREE POINT TURN + PAUSE ON OBSTACLE

7 Evaluation

The evaluation phase is concerned with analysing the outcomes of the research presented in this thesis and addresses the research question:

RQ1: What Digital Twin could be elaborated for the context of Autonomous Driving, with specific consideration given to flexibility, composability and extendability?

The evaluation design is presented in Section 7.1. The results are summarized in Section 7.2 and discussed in Section 7.3, while Section 7.4 presents threats to validity.

7.1 Evaluation Design

The evaluation assesses whether and how the requirements are met with a reflective review of the decisions made as well as with regard to the results and methodologies used and identify areas that have not yet been considered and/or that could be improved.

The author is aware of the necessity for an impartial and exhaustive assessment. Given his profound involvement, there is a possibility that unconscious bias may influence his evaluation's objectivity. In order to enhance the credibility and reliability of the findings, research assistants of the Software Quality and Architecture Group at the Institute of Software Engineering, University of Stuttgart were consulted as experts to provide an independent evaluation and fresh insights, and possibly identify areas for improvement that had not been considered.

This section presents a **scenario-based evaluation of the architecture** for the implementation of a DT for Autonomous Driving, especially with regard to extendability.

A scenario-based architecture evaluation is an approach to assessing software architectures that use scenarios to test and evaluate the architecture under various conditions. This is particularly useful in the context of Digital Twins, as they often represent complex and dynamic systems.

This method involves using specific Architecture-scenarios to evaluate how a software architecture performs under different conditions. It helps identify potential issues and areas for improvement by simulating real-world situations [KP00] [Ion05] [CKK+02] [KABC96].

Some typical evaluation methods are:

- Architecture-Level Modifiability Analysis (ALMA) [Ion05] [CKK+02]
- Architecture Trade-off Analysis Method (ATAM) [Ion05] [CKK+02]
- Software Architecture Analysis Method (SAAM) [Ion05] [KBAW94]

While aligned with methodologies like ATAM and SAAM, this evaluation is less elaborate in its approach.

Steps to perform the scenario-based evaluation:

1. Identify and assemble stakeholders:
   Topically the author of this thesis is the Developer, EndUser, Maintainer and SystemAdmin for this prototype. Other stakeholders are the host institution, the supervisor of the thesis, the automotive industry and others as well as DT developer, utilizer and maintainer and possibly the scientific world, research institutions, etc.

   The evaluation of the implementation for a DT especially with regard to extendability, is primarily done from the perspective of stakeholders such as developers and maintainers as well as researchers.

2. Documentation of the current architecture of the DT prototype:
   is described in Chapter 6 and more particularly in Section 6.4 and shown in Figure 6.5

3. Direct Scenarios to test and evaluate the architecture:

   - Insert DT-Entity-Scenario Overtaking as described in Subsection 5.3.2 and 6.4.15 as well as in Section 7.2

4. Indirect Scenarios (not presented in the thesis and requiring modifications or extensions):

   - Insert DT-Entity PathPlanning as described in Section 7.2
   - Insert DT-Entity-Scenario SchoolZone as described in Section 7.2

5. Perform Architecture-scenario Evaluation: analyse how well the architecture supports each scenario and identify potential issues and areas for improvement:
   is described in Section 7.2

6. Reveal Architecture-scenario Interactions: examine how different scenarios interact with each other and identify any conflicts or dependencies:
   is described in Section 7.2

7. Generate Overall Evaluation: Summarize the findings, highlighting strengths and weaknesses of the architecture:
   is described in Sections 7.2 and 7.3

The author prepared the following questions and discussion topics:

- how can the conceptualization and prototype implementation be evaluated?
- assess if and how the requirements are met, credibility, etc. (for implementation)
- reflective review of the decisions made (sensible, optimal, insights, etc.)
- identify areas that have not yet been considered and/or for improvement
- evaluate / validate the results and methodologies used in the thesis

For this evaluation two meetings were set up with the above mentioned experts with following agenda:

- presentation of the thesis scope, objective, use case, prototype implementation
- presentation of the videos with the illustrative scenarios

In the second meeting the elaborated Architecture-scenarios for the evaluation, with one direct and two indirect scenarios, were presented in form of a step-by-step walk-through and in giving detailed insights. The evaluation design, scope and results were discussed extensively.

7.2 Results

The objective of the evaluation was to ascertain whether the stipulated requirements and research questions are addressed in the devised concept and in the Use Case. This involves offering a reflective critique of the decisions made, evaluating their rationality and optimality, and identifying potential avenues for improvement and expansion in future endeavours.

A Digital Twin can be defined as a virtual model of a physical object or system that reflects its behaviour and state. In the scenario-based architecture evaluation of the prototype DT, a series of Architecture-scenarios were considered in order to assess the validity of the Digital Twin's concept and architecture, especially with regard to extendability (Req24, Req37).

Tables 7.1, 7.2 and 7.3 provide an overview of selected and evaluated Architecture-scenarios (inserted as direct and indirect), indicating whether modifications to existing components and/or Entities are necessary.

A more detailed examination of the nature of the changes (such as modification, adjustment, or extension) has not yet been conducted due to time constraints and could be addressed in future work.

Preliminary remarks to the Tables 7.1, 7.2 and 7.3 (legend):

- "no": No modification is required in the implemented prototype DT.
- "(no)": A minimal modification is required in the implemented prototype DT as it stands topically. However, it would not be required if the relevant part of the code would be externalized, out of the application entity, into a dedicated centralized entity (e.g. entity for conflict resolution). The conceptual approach already allows that. This was not implemented in the prototype due to the scope and time constraints of the thesis
- "(yes)": A modification is required in the implemented prototype DT as it stands topically. However, it would not be required if hardcoded object classes would be replaced with generic object classes for virtual replication. The conceptual approach already allows that. This was not implemented in the prototype due to the scope and time constraints of the thesis. The same applies to the creation of hardcoded database tables, which could be replaced by the use of generic database table creation
- "yes": A major modification is required in the implemented prototype DT
- "possible": modifications can be made if desired or necessary
- "not possible": cannot be implemented

General explanation with regard to the different insert Architecture-scenarios:

- in the case of "with changes in PT (shadowing)":
  It is assumed that all the necessary set of functionalities is fully implemented in the PT, i.e. for the DT-Entity-Scenario Overtaking, the DT-Entity-Scenario AdaptiveCruiseControl must also be available in the PT as functionality
- in the case of "with changes in PT":
  It is assumed that all the necessary physical elements like sensors, actuators, etc. are available in the PT. Furthermore these sensors and especially the cameras, have to be intelligent (in-build or via additional function) in a way that they are able for example to recognise road signs. However the functionality itself like School Zone (see below) is not implemented in the PT
- in the case of "without changes in PT":
  It is assumed that nothing is modified or added in the PT

Due to time constraints, this evaluation exclusively considers the cases presented herein, with no other mix or grading being taken into account.

Architecture-scenario	Classification	Modification required in DT-Entities		
		VehicleControl	Environment	DTMonitor
insert DT-Entity-Scenario Overtaking	direct			
-> with changes in PT (shadowing)		(no)	no	no
-> without changes in PT		(no)	no	no
insert DT-Entity PathPlanning	indirect			
-> with changes in PT		(yes)	no	no
-> without changes in PT		no	no	no
insert DT-Entity-Scenario SchoolZone	indirect			
-> with changes in PT		(yes)	no	no
-> without changes in PT		no	no	no

Table 7.1: Overview of Architecture–Scenarios for Evaluation 1/3

Architecture-scenario	Classification	Modification required in DT-Entities-Scenarios			
		ThreePointTurn	StopOnObstacle	AdaptiveCruiseControl	(Overtaking)
insert DT-Entity-Scenario Overtaking	direct				
-> with changes in PT (shadowing)		(no)	(no)	(no)	
-> without changes in PT		(no)	(no)	(no)	
insert DT-Entity PathPlanning	indirect				
-> with changes in PT		no	no	no	no
-> without changes in PT		no	no	no	no
insert DT-Entity-Scenario SchoolZone	indirect				
-> with changes in PT		no	no	no	no
-> without changes in PT		no	no	no	no

Table 7.2: Overview of Architecture-scenarios for Evaluation 2/3

Architecture-scenario	Classification	Modification required in DT Components		
		Dashboard	Data Manager	Database
insert DT-Entity-Scenario Overtaking	direct			
-> with changes in PT (shadowing)		yes	yes	no
-> without changes in PT		yes	yes	no
insert DT-Entity PathPlanning	indirect			
-> with changes in PT		possible	yes	yes
-> without changes in PT		possible	yes	yes
insert DT-Entity-Scenario SchoolZone	indirect			
-> with changes in PT		possible	yes	(yes)
-> without changes in PT		yes	yes	(yes)

Table 7.3: Overview of Architecture-scenarios for Evaluation 3/3

Detailed explanation for the evaluation Architecture-scenarios:

Insert DT-Entity-Scenario Overtaking

Description: The DT-Entity-Scenario Overtaking is detailed in Subsections 5.3.2, 6.4.15, and 6.5.6.

Constraint(s) and scope:

- in consideration of the configuration of the available track, which features a single lane, the DT-Entity-Scenario Overtaking is designed for the case in which the front vehicle deviates on a longer side road section situated on the right. Consequently, this vehicle covers a greater distance than the one overtaking it via the direct shorter route on the left
- DT-Entity-Scenario Overtaking uses PT's sensor data (Front distance sensor) and sends commands to the PT (speed regulation) as well as to the relevant PT's functionalities, namely "Stay On Track", "Line to follow" and "Pause on Obstacle".
- DT-Entity-Scenario Overtaking depends on the DT-Entity-Scenario AdaptiveCruiseControl to accurately perform the overtaking manoeuvre as intended

Note that while DT-Entity VehicleControl and DT-Entity-Scenario AdaptiveCruiseControl do not need to know the inserted DT-Entity-Scenario Overtaking, the DT-Entity-Scenario Overtaking does need to know both. It needs to know DT-Entity VehicleControl to adapt parameters and it needs to know DT-Entity-Scenario AdaptiveCruiseControl since it depends on it.

Modifications required in DT components for the case "with changes in the PT (shadowing)":

DT-Entity VehicleControl
: minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance (only valid if DT-Entity-Scenario Overtaking is not the issuer)

DT-Entity Environment
: no modification is needed as no correlation

DT-Entity DTMonitor
: no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
: minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance

DT-Entity-Scenario StopOnObstacle
: minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance

DT-Entity-Scenario AdaptiveCruiseControl
minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance (only valid if DT-Entity-Scenario Overtaking is not the issuer)

Data Manager
major modification:
integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
major modification:
create new adequate visualisation and command elements, etc.

Database
no modification is needed as there are no additional virtual replication aspects inserted

Modifications required in DT components for the case "without changes in the PT":

DT-Entity VehicleControl
minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance (only valid if DT-Entity-Scenario Overtaking is not the issuer)

DT-Entity Environment
no modification is needed as no correlation

DT-Entity DTMonitor
no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance

DT-Entity-Scenario StopOnObstacle
minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance

DT-Entity-Scenario AdaptiveCruiseControl
minimal modification:
insert the possibility to send a cancel message (see 6.4.7) to the DT-Entity-Scenario Overtaking for conflict avoidance (only valid if DT-Entity-Scenario Overtaking is not the issuer)

Data Manager
major modification:
integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
: major modification:
: create new adequate visualisation and command elements, etc.

Database
: no modification is needed as there are no additional virtual replication aspects inserted

Inspired from *Development of Autonomous Car—Part II* [TLZ+19] by Tao et al. the following two indirect Entities:

Insert DT-Entity PathPlanning

Description: The DT-Entity PathPlanning is responsible for determining the optimal route for the vehicle to reach a given destination. This process is based on received information about the perceived environment, which is stored in the DT-Entity Environment, and the current location. The current location can be determined either by receiving localization data (GPS, etc.) data from the PT or, in a laboratory experimentation, by simulating that information within the DT. Additionally, a specified destination is necessary for the DT-Entity PathPlanning to function.
The DT-Entity PathPlanning involves both short-term trajectory planning for immediate manoeuvres and long-term path planning for overall route optimization. In determining the optimal route, it could also consider other factors, such as road geometry and traffic rules, which are queried by external services.

Constraint(s) and scope:

- it is important to note that the DT-Entity PathPlanning only provides the determined optimal route as information (only descriptive and predictive role). It does not prescribe actions (not prescriptive role)
- it is assumed that DT-Entity PathPlanning does only determine the optimal route and does not consider nor present alternative selection choices (longest route, avoid tall, etc.)
- furthermore, it does not consider dynamic obstacles, as they are assumed to appear arbitrarily and, therefore, cannot be planned for in route planning
- in the case "with changes in the PT" it is assumed that the vehicle location sensor data, map data, etc. are available in the PT and published to the DT. There is no functionality implemented in the PT for Path Planning (this is done in the DT only)
- in the case "without changes in the PT" it is assumed that no location data nor data, etc. are available in the PT. In this laboratory experimentation, this information is assumed to be made available to the DT by other means (simulation case). There is no functionality implemented in the PT for Path Planning (this is done in the DT only)

Modifications required in DT components for the case "with changes in the PT":

DT-Entity VehicleControl
: modification:
: as new data is published by the PT, additional virtual replication elements need to be inserted

DT-Entity Environment
: no modification is needed as no correlation

DT-Entity DTMonitor
: no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
: no modification is needed as no correlation

DT-Entity-Scenario StopOnObstacle
: no modification is needed as no correlation

DT-Entity-Scenario AdaptiveCruiseControl
: no modification is needed as no correlation

DT-Entity-Scenario Overtaking
: no modification is needed as no correlation

Data Manager
: major modification:
integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
: possible if needed:
e.g. creating new adequate visualisation

Database
: major modification
as map data, etc. need to be stored, and additional / external databases might be needed

Modifications required in DT components for the case "without changes in the PT":

DT-Entity VehicleControl
: no modification is needed as no new data published by the PT

DT-Entity Environment
: no modification is needed as no correlation

DT-Entity DTMonitor
: no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
: no modification is needed as no correlation

DT-Entity-Scenario StopOnObstacle
: no modification is needed as no correlation

DT-Entity-Scenario AdaptiveCruiseControl
: no modification is needed as no correlation

DT-Entity-Scenario Overtaking
: no modification is needed as no correlation

Data Manager
: major modification:
  integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
: possible if needed:
  e.g. creating new adequate visualisation

Database
: major modification
  as map data, etc. need to be stored, and additional / external databases might be needed

Optional consideration with regard to navigation: Obviously, in the context of Autonomous Driving the DT-Entity PathPlanning will potentially be combined with an execution application. This would require a new prescriptive Entity-Scenario in the DT, typically called "Navigation", which would be dependent on the DT-Entity PathPlanning. However, this is not further evaluated due to the scope and time constraints of this thesis.

Insert DT-Entity-Scenario SchoolZone

Description: School Zones are highly dynamic areas with frequent pedestrian activity, especially involving children. These zones are generally speed-restricted to ensure the safety of pedestrians. Autonomous vehicles must strictly adhere to these speed limits and exercise heightened caution. The unpredictable nature of pedestrian movements, such as children unexpectedly crossing the street, necessitates that the Autonomous Driving system be highly responsive and accurate in detecting and reacting to these movements.

Constraint(s) and Scope:

- certain Entities-Scenarios in this Digital Twin (DT) are responsible for detecting obstacles and avoiding collisions
- the DT-Entity-Scenario SchoolZone is specifically designed to recognise School Zones and ensure that the vehicle's speed remains within the defined speed limit
- it also interferes if necessary to ensure that the speed is reduced to the maximum allowed
- it analyses (compares speed restriction with the actual speed of the vehicle) and issues corrective measures
- it is assumed that if the driver exceeds the speed limit then the DT-Entity-Scenario SchoolZone can overwrite it in a way of sending deceleration commands
- it is assumed that if DT-Entity-Scenario AdaptiveCruiseControl is active, the DT-Entity-Scenario SchoolZone can overwrite it in a way of sending corrective speed ranges to the DT-Entity-Scenario AdaptiveCruiseControl

- same applies for DT-Entity-Scenario ThreePointTurn and DT-Entity-Scenario Overtaking
- the design has to be fine-tuned in order to prevent and cope with conflicting situations in ways of prioritizing (re-)actions of each Entity
- obviously, the Entities-Scenarios responsible for collision avoidance (e.g. DT-Entity-Scenario StopOnObstacle) shall have the highest priority
- however, not all of these aspects are further evaluated due to the scope and time constraints of this thesis
- in the case "with changes in the PT" it is assumed that a camera and relevant functionality capable of recognising the school zones and speed limit indication is available in the PT and that information is published to the DT. There is no functionality implemented in the PT for School Zone (this is done in the DT only)
- in the case "without changes in the PT" it is assumed that no such camera is available in the PT. In this laboratory experimentation, the necessary information is virtually simulated in the DT. There is no functionality implemented in the PT for School Zone (this is done in the DT only)

Modifications required in DT components for the case "with changes in the PT":

DT-Entity VehicleControl
: modification:
as new data is published by the PT, additional virtual replication elements need to be inserted

DT-Entity Environment
: no modification is needed as no correlation

DT-Entity DTMonitor
: no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
: no modification is needed as no correlation

DT-Entity-Scenario StopOnObstacle
: no modification is needed as no correlation

DT-Entity-Scenario AdaptiveCruiseControl
: no modification is needed as no correlation

DT-Entity-Scenario Overtaking
: no modification is needed as no correlation

Data Manager
: major modification:
integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
: possible if needed:
e.g. creating new adequate visualisation

Database
: modification
  additional virtual replication elements might require additional tables, etc.

Modifications required in DT components for the case "without changes in the PT":

DT-Entity VehicleControl
: no modification is needed as no new data published by the PT

DT-Entity Environment
: no modification is needed as no correlation

DT-Entity DTMonitor
: no modification is needed as DT-Entity DTMonitor receives heartbeats implicitly from the DT-Entity-Scenario Overtaking

DT-Entity-Scenario ThreePointTurn
: no modification is needed as no correlation

DT-Entity-Scenario StopOnObstacle
: no modification is needed as no correlation

DT-Entity-Scenario AdaptiveCruiseControl
: no modification is needed as no correlation

DT-Entity-Scenario Overtaking
: no modification is needed as no correlation

Data Manager
: major modification:
  integrate it in the Data Manager with all the connection points, flow structure, de-serialization, etc.

Dashboard
: major modification:
  create new adequate visualisation and e.g. virtual School Zone (de-)activation, etc.

Database
: modification
  additional virtual replication elements might require additional tables, etc.

Result of the evaluation of Architecture-scenarios (how well does the architecture support each scenario?):

The bar charts presented herewith delineate the frequency of required modifications across various components in the DT for the Architecture-scenarios under discussion.

The evaluation of the Architecture-scenario **"insert DT-Entity-Scenario Overtaking"**, presented in Figure 7.1, shows that in 78% of cases, no or only minimal modifications are required. This demonstrates that this direct Architecture-scenario is very well supported.

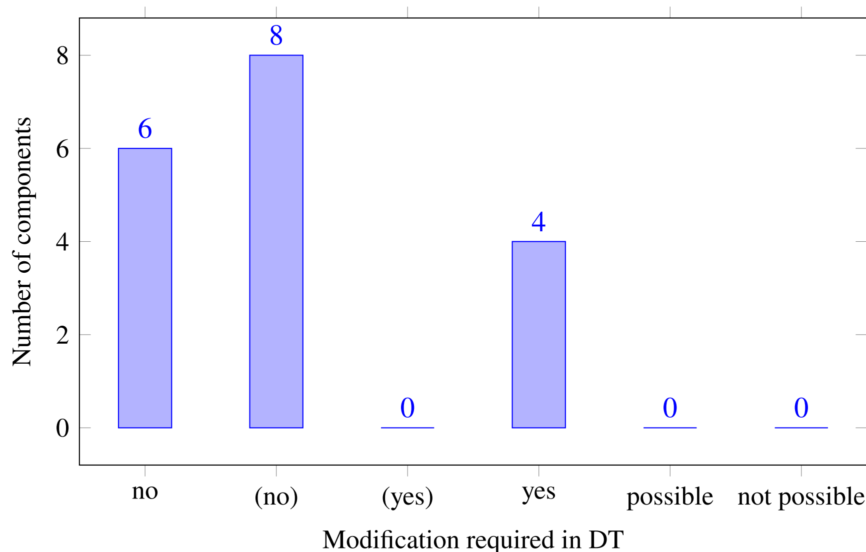

Figure 7.1: Bar chart for Architecture-scenario "insert DT-Entity-Scenario Overtaking"

The evaluation of the Architecture-scenario **"insert DT-Entity PathPlanning"**, presented in Figure 7.2, shows that in 65% of cases, no modifications are required. This demonstrates that this indirect Architecture-scenario is well supported.

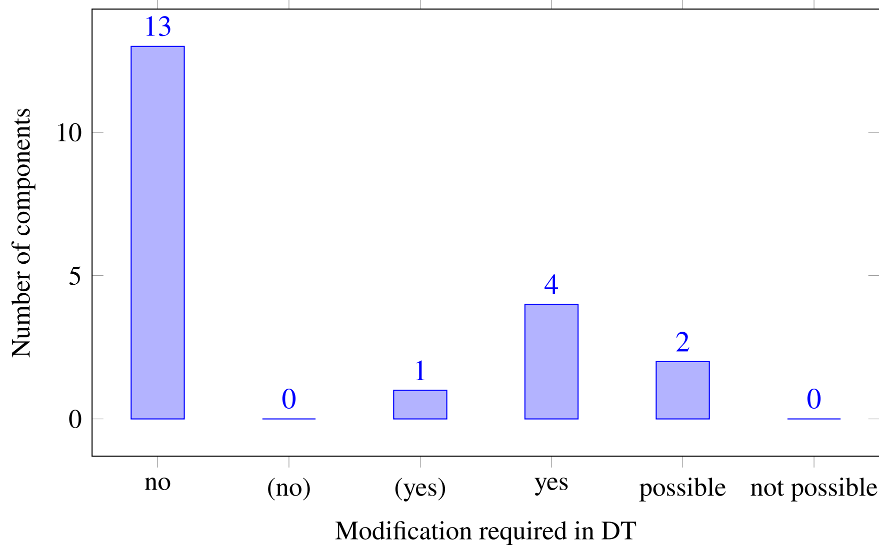

Figure 7.2: Bar chart for Architecture-scenario "insert DT-Entity PathPlanning"

The evaluation of the Architecture-scenario **"insert DT-Entity-Scenario SchoolZone"**, presented in Figure 7.3, shows that in 65% of cases, no modifications are required. This demonstrates that this indirect Architecture-scenario is well supported.

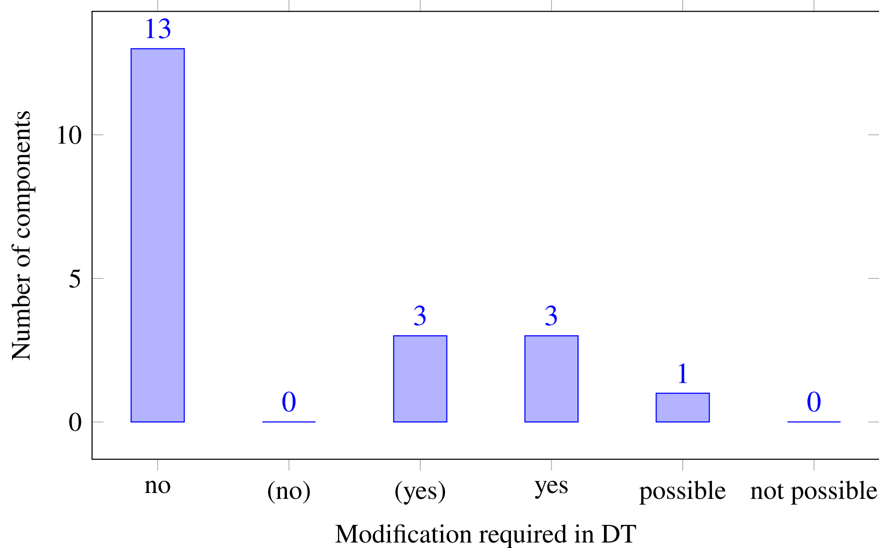

Figure 7.3: Bar chart for Architecture-scenario "insert DT-Entity-Scenario SchoolZone"

Reveal Architecture-scenario Interactions (examine how different scenarios interact with each other):

The possible interactions between the different scenarios have been addressed in the description part above.

Overall Evaluation (Summarize the findings, highlighting strengths and weaknesses of the architecture):

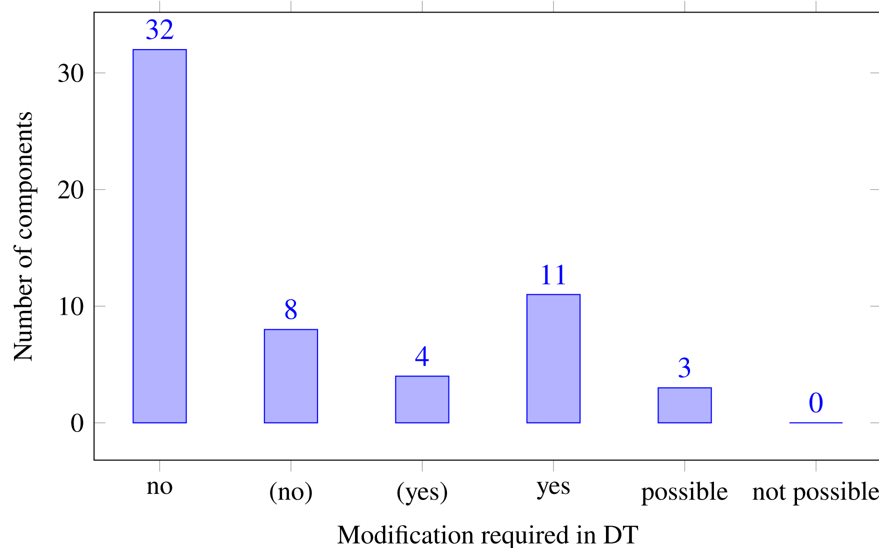

Figure 7.4: Bar chart for the 3 Architecture-scenarios of the evaluation

The evaluation of the 3 Architecture-scenarios, presented in Figure 7.4, shows that in 69% of cases, no or only minimal modifications are required.

All the evaluated (direct and indirect) scenarios are very well supported, i.e. implementable in the DT without restriction, considering the necessary adaptation.

The evaluation shows that the proof-of-concept implementation of the prototype DT for Autonomous Driving is flexible, composable and fully extendable.

7.3 Discussion

The experts indicated that the evaluation construct is appropriate and identified areas for improvement (future works). They also gave a reflective view of the architectural decisions made.

The findings of the expert's interview confirmed that if the architecture becomes more heavyweight, i.e. in adding more Entities into the DT, it will be necessary to address structural conflict handling, e.g. by externalizing this into dedicated Entity(ies).
Furthermore, they emphasised that in order to ensure optimal usability and facilitate further advancements of the DT, beyond the prototype phase, comprehensive qualitative documentation is of paramount importance. For instance, it is crucial to have detailed documentation on how changes could be performed, what the insertion steps would be, and to assess the workload such a change would involve.

A pattern-based approach as employed in this thesis, (e.g. usage of a message-based platform) contributes positively to extendability.
A usage and extension of the DT beyond the prototype stage would require a more extensive and adequate dashboard (user interface). It would also be advantageous to implement a patterned generic approach for the Dashboard / UI in order to facilitate the integration of extensions to the DT, when and how the necessity arises.
The experts concurred that in the event of a dependency between entities, a dedicated entity for conflict resolution should be established. They pointed out that in order to enhance extendability, it is essential that scenarios are designed to be managed by the new entity for conflict resolution. The author suggested that this new entity could centralize information and that the necessary categorisation and prioritisation could occur in the entity for conflict resolution.

The experts concurred with the thesis author's overall analysis of the scope and constraints and necessary modifications with regard to the evaluated Architecture-scenarios as well as with the results of the evaluation.

7.4 Threats to Validity

This section presents identified (possible) threats to the validity of the presented results.

- the concept and Use Case are inspired by previous works and therefore this thesis is inevitably influenced by them
- regarding the application domain, it is worth mentioning that the author of this thesis does not have the expertise in the automotive field that specialists and engineers might have
- moreover, the prototype and the scenarios have been designed to align with the specific characteristics and constraints of the Robot Cars of the university, in a laboratory setting. Consequently, it does not reflect the complexities and nuances of a real-world use case
- given that the DT in this thesis is a prototype and the technical unreliability of the PT, it has not been feasible to identify a suitable case study or appropriate performance indicators
- the number of experts consulted was relatively limited, given that the physical entity employed in the implementation is from the university. One of the experts consulted is the supervisor of this thesis
- it is possible that the results may not be applicable in different contexts

8 Conclusion and Outlook

This chapter concludes the thesis about the Conceptualization and Implementation of a Digital Twin for Autonomous Driving. It summarizes the key aspects in Section 8.1. The benefits are outlined in Section 8.2 and the limitations in Section 8.3. It also reflects some valuable lessons learned in Section 8.4. Section 8.5 provides an overview of potential avenues for future work and research.

8.1 Summary

This thesis introduces a concept for a Digital Twin (DT), with a particular emphasis on flexibility, composability and extendability. It is modelled in such a way that minor changes, major modifications and also adding new components / elements (e.g. new sensors, etc.) and/or functionalities / entities (e.g. other scenarios, simulations, testing, etc.) can be readily added and managed. This entails functional decompositions, the separation of concerns, and the establishment and maintenance of association relations between the twins (DT and PT). It employs self-adaptation, control, and model-based engineering techniques to specify the structural and behavioural aspects of DTs.

The focus of this thesis is Autonomous Driving, but the conceptual approach is sufficiently general that it could be applied to other areas of application.

In order to test and demonstrate the concept and architecture within a use case, a prototype DT was devised and implemented from scratch without relying on existing DT software platforms. It incorporates all the relevant components or elements of the four enabling technology domains for I/IoT, namely: object, networking, middleware and application. This entailed the elaboration of the design of the DT, encompassing aspects such as architecture, structure, data/information flow, etc. Two Arduino Robot Cars and a single lane track, in a laboratory environment, were used as Physical Twin. A scenario-based evaluation demonstrated especially its extendability. A series of showcase scenarios pertaining to Autonomous Driving were tested and presented in order to illustrate the scope, capabilities and potential of the DT.

The implementation shows that the prototype DT replicates the behaviour of the PT in the DT virtually, i.e. not only does shadowing but really operates in parallel to the PT and interacts with it. The aforementioned processes and states can be visualised and operated via the convenient and comprehensive dashboard. The DT is designed to facilitate alterations, provided they remain within the defined constraints. Furthermore, the DT offers the potential and capability for experimentation, including the ability to run simulations either partially or fully de-/coupled with the PT. This permits the examination of issues, modifications, or evolutions of DT and/or PT elements or functionalities, as well as new scenarios.

8.2 Benefits

The DT concept and architecture put forward in this thesis enable the efficient engineering of Digital Twins in both theoretical and practical contexts. The concept is structured in such a way that it enables evolutions in terms of adaptations, extensions and more, and extends beyond the use case of Autonomous Driving. In comparison to other works, this thesis presents a detailed architectural specification for the DT, encompassing not only its internal components and interconnections but also the interactions between the DT and the PT. In light of the aforementioned, this proposed concept and architecture provide a foundation for researchers to further elaborate on Digital Twins in general as well as for specific applications, and for practitioners to implement composable and extendable Digital Twins effectively.

8.3 Limitations

It should be noted that this thesis does not elaborate on the conceptual approach or implementation of Autonomous Vehicles themselves, but rather focuses solely on the Digital Twin. Consequently, this thesis does not examine aspects such as protocols, elements, algorithms, standards or interfaces of Autonomous Driving. The concept presented in this thesis is for a Digital Twin of a more general nature and at a higher definition level.

It is to be noted that the Arduino Robot Car model used for the prototype implementation is markedly distinct from a conventional vehicle in several key aspects. It is devoid of intricate car architecture, numerous components and, in particular, the array of sensing elements such as LIDAR, RADAR, cameras, and navigational elements that are characteristic of driving assistance systems. Furthermore, the absence of a genuine driver and a realistic environment, including road(s) with multiple lanes, traffic and weather-related factors, limits the extent to which the simulation can accurately reflect real-world driving conditions. The same is true of the interconnection and interaction between the PT and the DT, which are undoubtedly different in real vehicles.

Similarly, the selection of software tools and their respective versions may prove inadequate for deployment in real-world use cases.

The constraints and assumptions defined in this thesis are entirely relevant.

8.4 Lessons Learned

The following aspects, experienced or noted during the course of the research, the conceptualization and the implementation, are worth mentioning:

It is essential to assess and define the goals, scope and risks of a DT project at the outset, in collaboration with all relevant stakeholders and, where appropriate, users. For example, it is essential to reach a consensus on which modifications are to be made or planned in the physical entity(ies), either before or during the physical entity's realization, and which are not to be considered. This pertains to the addition, retrieval, replacement, or alteration of elements, including but not limited to actuators, sensors, ECUs, and other components. Additionally, it is crucial to determine which functionalities are available, are planned and will be included or excluded. Furthermore, it is advisable to reach a consensus on the precise scope of the DT. Any alterations to the project's external or internal scope, specifications, milestones, objectives and targets, during the realization of the DT, have the potential to jeopardize the project's success.

It is evident that the PT design, functionalities, and associated limitations and restrictions are of paramount importance in the development of a DT. It is therefore imperative that the PT remains accessible throughout the duration of the project and that an expert is available to provide comprehensive support with regard to the physical object or asset. It is crucial to consider the impact of external factors on a project and to avoid underestimating their influence. The same can be said of any uncoordinated modification of the PT. It is essential that all stakeholders make a clear and unambiguous commitment at the outset of a project and that their contributions are made in a timely manner if the project is to be feasible.

It is imperative that the prototype DT incorporates control and adaptation mechanisms to facilitate the DT's capacity to continuously shadow and react to alterations in the physical counterparts. Moreover, the prototype DT should provide the opportunity for experimentation, for example, by running simulations of the PT, either partially or fully decoupled, in order to examine issues, modifications or evolutions of PT elements or functionalities, or to explore new scenarios.

The absence of comprehensive documentation and information about the physical entity renders the process somewhat arduous.

A comprehensive review and clarification of the physical entity (PT) is essential and should include precise information about the functionalities, design and limitations (e.g. memory capacity, processor speed, hardware, software code, etc.).

It is essential to determine the most appropriate components for communication and interconnection with the Physical Twin(s), taking into account factors such as the proximity of both entities or their potential separation, whether at the same location or at disparate sites, including remote locations. It is essential to select the communication protocol and data format (e.g. MQTT and JSON in this prototype DT) in a manner that ensures a seamless data flow between the PT, DT, Database, Dashboard and other relevant components. Furthermore, the interconnection between the DT and the Physical Twin must be subjected to rigorous engineering analysis.

It would be advisable to consider the use of a pattern-based approach when and where appropriate, given the relevance of extendability (Req24, Req37).

During the implementation of the prototype DT, a number of issues were identified with the PT. In order to gain a full understanding of these issues, a comprehensive and meticulous examination, testing and experimentation was required. This process involved, on occasion, trial and error in order to identify the root cause. The erratic behaviour observed was due to limitations of the PT, or due to wear and tear (e.g. battery capacity decreasing), or other factors. A comprehensive dashboard is an invaluable tool for visualising the state of the system, issuing manual commands, conducting simulations, testing, and detecting and resolving issues. To illustrate, at a certain point, the Robot Car exhibited erroneous and non-reproducible behaviour. An examination of the distance values displayed on the Dashboard helped to elucidate the underlying cause. It was established that the distance sensor was unable to accurately gauge the distance of the vehicle in front due to the fact that its rear surface was not uniform.

It is advantageous, if not essential, to have a user interface (dashboard, cockpit, etc.) that is tailored to the specific objective and requirements. It is recommended that a data management layer and tool(s) be implemented as a gateway, in conjunction with the protocol broker, to facilitate the proper distribution of data and to assist with tasks such as data de-/serialization (Req15, Req16, Req28,Req31,Req38, Req39, Req40). The decision to utilize a JavaScript object-oriented programming tool, Node-RED, for the Data Manager and the Dashboard (Req04, Req05, Req12, Req18, Req21, Req39) has been demonstrated to be a prudent one. It offers excellent visibility of the (sub-)flows and provides additional tooling for de-/serialization and for use during the coding and debugging phases, as well as for testing purposes. Moreover, it integrated seamlessly with the other components. It is important to acknowledge that this is pertinent to this prototype DT, which had to be implemented within a limited timeframe. The complexity and objective of the respective DT project will determine the most appropriate selection of tools to be employed, given the specific requirements of the task at hand.

The deployment of the prototype and the necessity to surmount the challenges, especially through the utilization of test and simulation phases employing the DT and its Dashboard (at times in synchronized or simulation mode), provided evidence that the Digital Twin is a useful and valuable instrument.

8.5 Future Work

The future of Digital Twins in Autonomous Driving is promising, with ongoing research and development aimed at enhancing their capabilities and applications. This section outlines potential avenues for future research and work as follows:

in relation to the concept and architecture presented in this thesis:

- further, elaborate the concept and/or the architecture for example with a model-driven code generation approach
- validate and/or adapt it to other application areas/domains

in relation to the use case and the implementation DT presented in this thesis:

a) **with regard to the general aspects:**

- assess which programming tool(s) and language(s) are best suited for the targeted Digital Twin, e.g. with regard to runtime, etc.
- perform more comprehensive and qualitative documentation in order to ensure optimal usability and facilitate further advancements of the DT, beyond the prototype phase
- implement a patterned generic approach, when and where applicable
- usage and extension of the DT beyond the prototype stage will require to design and implement a more extensive and sophisticated user interface (dashboard, cockpit, etc.) in selecting the proper tooling, and making use for example of multiscreen, cascading windows, drop-down menus, dedicated layers, etc., to best structure and arrange it. It might be advantageous to implement a patterned generic approach for it, in order to facilitate the integration of future extensions (Req32)
- design and implement dedicated applications/entities, to manage conflict resolution between the respective Entities-Scenarios. Centralize such information using a hash map which would map the respective scenario entities to a list containing attributes. These attributes describe the respective Entities' areas of responsibility. The necessary categorization and prioritization could be centralized in the new Entity for conflict resolution
- add more simulation cases and for that also implement more dashboard items as well as more selectivity at the data management level dashboard
- develop the Database to the next level and, if need be, integrate more databases and/or sources, possibly remote ones, e.g. with map information for geo-localization
- evolution of the DT, if need be, with regard to the use of web, API, REST-API, etc.
- evolution of the DT, with regard to security measures (Req20) for classes and communication
- optimize the DT Performance, e.g. in the area of monitoring, etc.
- optimize the DT services (Req13) docking for services since all code is now via "PredefinedSystem", i.e. short internal code
- adapt, in the Java code, the weighting for ACC (currently all weights are 1.0)

- improve the PT configuration information transfer to the DT
- improve the DT in order to facilitate a more dynamic approach for extendability (Req24, Req37),
  e.g. with a better solution as timeout, and automation of a verified transfer of the PT configuration in the DT
- it may be beneficial to consider the potential advantages of utilizing an explicit rate limiter, provided that it does not impede the transmission of messages. In light of the time constraints and the absence of a clear necessity for this prototype, it was not subjected to further consideration. This implementation does not include an explicit rate limiter. The rate limiter is implicit, based on the utilization of fixed time intervals for querying sensor values. The interval length is determined based on empirical evidence
- investigate runtime performance (Req33)
- if necessary, implement more functionality into the application Environment.
  Thereby consider an evolution so that in future extension, it fulfils if need be, also a predictive role (Req14), e.g. by assessing the potential impacts of environmental changes, such as weather or road conditions, on vehicle performance.
  Thereby also consider an evolution so that in future extension, it fulfils if need be, also a functionality as Evaluator, e.g. in order to evaluate current environmental conditions based on the processed data in order to inform other Entities
- if necessary, implement more functionality into the application "DT Monitor".
  Thereby consider an evolution so that in future extension, it fulfils if need be, also a predictive role, e.g. by detecting anomalies and anticipating issues that may arise, prompting preventive measures, etc. in order to maintain a more sophisticated level of system integrity and performance
- if necessary (e.g. if too many messages from various vehicles), create more / dedicated (vehicle) control applications or extend the existing VehicleControl application with more functionality (Req26)
- explore and implement adequately, in conjunction with the Robot Car, a better behaviour when the vehicle follows the line (better sensor, PID, etc.)
- explore how to best solve the situation when the Robot Car gets off track so that it finds the way back
- the optimal message structure should be determined. In the prototype, a single message is sent, and the string is not too long for either the string data type or the message payload. Accordingly, the structure of the DT is currently designed in such a way that only the minimum required JSON elements are integrated into a message. This results in improved runtime performance, which is also attributable to the relatively high speed of JSON libraries when reading and writing.
- research on the possibility of evolution by integrating a more advanced approach in automating the adaptive process of verification and validation for the integration of modification and/or new application in the DT (Req33)
- research on areas of security and safety for the DT

- evaluate and possibly line up papers and requirements such as "Digital Twin Capabilities Periodic Table " from the Digital Twin Consortium [Dig24]

b) with regard to the domain Autonomous Driving:

- adapt the DT to a more complex PT (vehicle and road), e.g. Robot Cars with fewer limitations than the ones exposed in Subsection 6.2.1 or even more sophisticated (robot) cars.
  The same applies to the track/road which could accommodate for example multi lanes, crossroads, road signs, traffic lights, pedestrian crossings, etc.
- evolution of the DT, with regard to new components and/or functionalities or alterations in the PT, e.g. add more vehicle elements (e.g. brakes, LIDAR, RADAR, camera, etc.) (Req41)
- evolution of the DT, with regard to new scenarios cases, e.g. braking (ABS, etc.), hand brake, rear-sensor, etc. (Req22)
- evolution of the DT, with regard to more complex scenarios cases, e.g. add data fusion and aggregation of various sensors data relevant for a specific more sophisticated functionality, like localization and navigation, complex traffic situation, automated parking, etc. (Req42, Req45)
- explore how to best implement a full fledged vehicle spacial orientation
- evolution of the DT, with regard to more simulations cases
- evolution of the DT, with regard to the quantities of for a PT with higher volumes of vehicle (fleet, etc.)
- evolution of the DT, with regard to the type of vehicle (cars, bikes, motorbikes, trucks, busses, etc.) and possibly also address issues to be handled with a mix of them in the PT
- apply and adapt the DT to real PTs, i.e. connect to a real vehicle (instead of robot car)

c) with regard to other application areas/domains:

- apply and adapt the DT to other physical objects, assets, application areas/domains, etc.
- apply and adapt the DT to real PTs, i.e. connect and replicate real physical objects, assets, etc.

d) with regard to evolutions of the DT (AI, etc.):

- research on evolutionary step(s) to combine and /or integrate in/with the DT technologies like Big Data analytics, Machine Learning, Deep Learning, and AI technologies
- evolution in mapping and deploying the DT on platforms with higher automation potential and innovative technologies

In conclusion, the presented conceptual approach, the architecture and the use case, as well as the prototype Digital Twin, represent contributions that may be pursued along multiple avenues and in various domains.

Bibliography

[AKP+20] D. Adamenko, S. Kunnen, R. Pluhnau, A. Loibl, A. Nagarajah. “Review and comparison of the methods of designing the Digital Twin”. In: *Procedia CIRP* 91 (2020). Enhancing design through the 4th Industrial Revolution Thinking, pp. 27–32. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2020.02.146. URL: https://www.sciencedirect.com/science/article/pii/S2212827120307800 (cit. on p. 19).

[APT24] Z. Ali, M. Poursoltan, M. K. Traore. “DMFDT: Data Management Framework for Digital Twin”. In: *Navigating Unpredictability: Collaborative Networks in Non-linear Worlds (PRO-VE 2024)*. Springer, 2024, pp. 130–144. DOI: 10.1007/978-3-031-71743-7_9. URL: https://link.springer.com/chapter/10.1007/978-3-031-71743-7_9 (cit. on p. 84).

[Ard] Arduino SA. *Sketch build process*. URL: https://arduino.github.io/arduino-cli/0.20/sketch-build-process/ (cit. on p. 172).

[Ard23] Arduino Forum. *Why do people send and save numeric data as ASCII characters?* Accessed: 2024-09-26. 2023. URL: https://forum.arduino.cc/t/why-do-people-send-and-save-numeric-data-as-ascii-characters/308962 (cit. on p. 178).

[AVVT20] J. Autiosalo, J. Vepsalainen, R. Viitala, K. Tammi. “A Feature-Based Framework for Structuring Industrial Digital Twins”. In: *IEEE Access* 8 (2020), pp. 1193–1208. DOI: 10.1109/ACCESS.2019.2950507. URL: https://doi.org/10.1109/ACCESS.2019.2950507 (cit. on p. 49).

[BAR20] A. Bhat, S. Aoki, R. Rajkumar. “Tools and Methodologies for Autonomous Driving Systems”. In: *Proceedings of the IEEE*. Vol. 106. 9. 2020, pp. 1700–1716. URL: https://ieeexplore.ieee.org/document/8418447 (cit. on p. 46).

[BEFH20] L. Barth, M. Ehrat, R. Fuchs, J. Haarmann. “Systematization of Digital Twins: Ontology and Conceptual Framework”. In: *Proceedings of the 3rd International Conference on Information Science and Systems*. ICISS ’20. Cambridge, United Kingdom: Association for Computing Machinery, 2020, pp. 13–23. ISBN: 9781450377256. DOI: 10.1145/3388176.3388209 (cit. on p. 29).

[BS21] M. R. Bachute, J. M. Subhedar. “Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms”. In: *Machine Learning with Applications* 6 (2021), p. 100164. ISSN: 2666-8270. DOI: https://doi.org/10.1016/j.mlwa.2021.100164. URL: https://www.sciencedirect.com/science/article/pii/S2666827021000827 (cit. on p. 46).

[BT15] S. Behere, M. Torngren. “A functional architecture for autonomous driving”. In: *2015 First International Workshop on Automotive Software Architecture (WASA)*. 2015, pp. 3–10. DOI: 10.1145/2752489.2752491. URL: https://ieeexplore.ieee.org/abstract/document/7447216 (cit. on pp. 36, 37).

[CKK+02] P. Clements, R. Kazman, M. Klein, L. Bass, P. Clements, R. Kazman. "SCENARIO-BASED SOFTWARE ARCHITECTURE EVALUATION METHODS An overview". In: (2002). URL: https://www.sasg.nl/sasg16Dieter_Hammer.pdf (cit. on p. 137).

[CKM+21] B. Combemale, J. A. Kienzle, G. Mussbacher, H. Ali, D. Amyot, M. Bagherzadeh, E. Batot, N. Bencomo, B. Benni, J. M. Bruel, J. Cabot, B. H. Cheng, P. Collet, G. Engels, R. Heinrich, J. M. Jezequel, A. Koziolek, S. Mosser, R. Reussner, H. Sahraoui, R. Saini, J. Sallou, S. Stinckwich, E. Syriani, M. Wimmer. "A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems". In: *IEEE Software* 38.4 (2021), pp. 71–84. DOI: 10.1109/MS.2020.2995125 (cit. on p. 29).

[Com24] N.-R. Community. *node-red-node-sqlite*. https://flows.nodered.org/node/node-red-node-sqlite. 2024 (cit. on p. 82).

[Cro06] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. 2006. URL: https://www.rfc-editor.org/rfc/rfc4627 (cit. on p. 70).

[Cur23] M. Currey. *Arduino Serial: ASCII Data and Using Markers to Separate Data*. Accessed: 2024-09-26. 2023. URL: https://www.martyncurrey.com/arduino-serial-ascii-data-and-using-markers-to-separate-data/ (cit. on p. 178).

[Dig22] Digital Twin Consortium. *Digital Twin Capabilities Periodic Table*. 2022. URL: https://www.digitaltwinconsortium.org/initiatives/capabilities-periodic-table/ (cit. on p. 49).

[Dig24] Digital Twin Consortium. *Glossary of Digital Twins*. 2024. URL: https://www.digitaltwinconsortium.org/glossary/glossary/#digital-twin (cit. on pp. 26, 161).

[Ecl24a] Eclipse Foundation. *Eclipse Ditto™ - Overview*. 2024. URL: https://eclipse.dev/ditto/intro-overview.html (cit. on p. 77).

[Ecl24b] Eclipse Foundation. *Eclipse Mosquitto*. 2024. URL: https://mosquitto.org/ (cit. on p. 80).

[EI90] I. of Electrical, E. E. (IEEE). *IEEE Standard Glossary of Software Engineering Terminology*. 1990. URL: https://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology%20.pdf (cit. on pp. 21, 25, 49).

[FFDB20] A. Fuller, Z. Fan, C. Day, C. Barlow. "Digital Twin: Enabling Technologies, Challenges and Open Research". In: *IEEE Access* PP (May 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2998358 (cit. on p. 52).

[GFSA17] T. M. Gasser, A. T. Frey, A. Seeck, R. Auerswald. *Comprehensive definitions for automated driving and ADAS*. 2017 (cit. on p. 27).

[GGLM24] S. Gil Arboleda, C. Gomes, P. G. Larsen, P. H. Mikkelsen. "Survey on open-source digital twin frameworks–A case study approach". In: *Software: Practice and Experience* 54.1 (2024), pp. 1–32. DOI: 10.1002/spe.3305. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.3305 (cit. on p. 77).

[GLP17] J. C. Guzmán, G. López, A. Pacheco. “Defining ‘Architecture’ for Software Engineering – A Review of Terminology”. In: *Advances in Human Factors, Software, and Systems Engineering*. Springer, 2017, pp. 143–152. URL: https://link.springer.com/chapter/10.1007/978-3-319-60011-6_14 (cit. on p. 30).

[Gri05] M. W. Grieves. “Product lifecycle management: the new paradigm for enterprises”. In: *International Journal of Product Development* 2.1-2 (2005). URL: https://www.researchgate.net/publication/247833967_Product_lifecycle_management_the_new_paradigm_for_enterprises (cit. on p. 32).

[Int11] International Organization for Standardization. *ISO 26262:2011 Road vehicles - Functional safety*. Geneva, Switzerland, 2011. URL: https://www.iso.org/standard/43464.html (cit. on p. 36).

[Int15] International Organization for Standardization. *ISO/IEC 2382:2015: Information technology - Vocabulary*. Accessed: 2024-11-23. 2015. URL: https://www.iso.org/standard/63598.html (cit. on p. 49).

[Ion05] M. T. Ionita. *Scenario-based system architecting : a systematic approach to developing future-proof system architectures*. 2005. URL: https://pure.tue.nl/ws/portalfiles/portal/1764177/200512319.pdf (cit. on p. 137).

[IoT23] IoT For All. *Why MQTT Is Essential for Building Connected Cars*. 2023. URL: https://www.iotforall.com/why-mqtt-essential-connected-cars (cit. on p. 80).

[Jac15] D. Jackson. “Towards a Theory of Conceptual Design for Software”. In: *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology* (2015). URL: https://groups.csail.mit.edu/sdg/pubs/2015/concept-essay.pdf (cit. on p. 28).

[KABC96] R. Kazman, G. Abowd, L. Bass, P. Clements. *Scenario-Based Analysis of Software Architecture*. Tech. rep. CMU/SEI-96-TR-014. Software Engineering Institute, Carnegie Mellon University, 1996. URL: https://insights.sei.cmu.edu/documents/213/1996_019_001_29912.pdf (cit. on p. 137).

[KBAW94] R. Kazman, L. Bass, G. Abowd, M. Webb. “SAAM: a method for analyzing the properties of software architectures”. In: *Proceedings of 16th International Conference on Software Engineering*. 1994, pp. 81–90. DOI: 10.1109/ICSE.1994.296768 (cit. on p. 137).

[KJD+15a] J. Kichun, K. Junsoo, K. Dongchul, J. Chulhoon, S. Myoungho. “Development of Autonomous Car—Part I: Distributed System Architecture and Development Process”. In: *IEEE Transactions on Industrial Electronics* 62.8 (2015), pp. 5107–5118. DOI: 10.1109/TIE.2014.2321342. URL: https://ieeexplore.ieee.org/document/6809196 (cit. on pp. 27, 50).

[KJD+15b] J. Kichun, K. Junsoo, K. Dongchul, J. Chulhoon, S. Myoungho. “Development of Autonomous Car—Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture”. In: *IEEE Transactions on Industrial Electronics* 62.8 (2015), pp. 5119–5132. DOI: 10.1109/TIE.2015.2410258. URL: https://ieeexplore.ieee.org/document/7056521 (cit. on pp. 33, 34, 58, 59).

[KKG23] O. Kızılırmak, E. Kaplan, Ç. Güzay. "Digital Twin Architecture for Autonomous Driving Validation and Verification". In: *2023 14th International Conference on Electrical and Electronics Engineering (ELECO)*. 2023, pp. 1–6. DOI: 10.1109/ELECO60389.2023.10416035 (cit. on p. 45).

[KKT+18] W. Kritzinger, M. Karner, G. Traar, J. Henjes, W. Sihn. "Digital Twin in manufacturing: A categorical literature review and classification". In: *IFAC-PapersOnLine* 51.11 (2018). 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018, pp. 1016–1022. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.08.474. URL: https://www.sciencedirect.com/science/article/pii/S2405896318316021 (cit. on pp. 26, 27, 31, 32, 39, 47, 54, 194).

[KMG+20] S. Kriebel, M. Markthaler, C. Granrath, J. Richenhagen, B. Rumpe. "Modeling Hardware and Software Integration by an Advanced Digital Twin for Cyber-Physical Systems: Applied to the Automotive Domain". In: *Handbook of Model-Based Systems Engineering*. 2020, pp. 21–38. DOI: 10.1007/978-3-030-27486-3_21-1. URL: https://link.springer.com/referenceworkentry/10.1007/978-3-030-27486-3_21-1 (cit. on p. 44).

[KP00] R. Kazman, W. Pree. *Architecture analysis: The SAAM, ATAM*. Tech. rep. Carnegie Mellon University, 2000. URL: https://softwareresearch.net/fileadmin/user_upload/Teaching/SS02/SWA_slides3.pdf (cit. on p. 137).

[LK21] N. Lloyd, A. S. Khuman. "Adaptive Cruise Control Using Fuzzy Logic". In: Springer Nature Switzerland AG, 2021. Chap. 12, pp. 191–204. ISBN: 978-3-030-66474-9. DOI: 10.1007/978-3-030-66474-9_12 (cit. on pp. 61, 101, 106, 107).

[MC21] R. Minerva, N. Crespi. "Digital Twins: Properties, Software Frameworks, and Application Scenarios". In: *IT Professional* 23.1 (2021), pp. 51–55. DOI: 10.1109/MITP.2020.2982896. URL: https://ieeexplore.ieee.org/document/9340048 (cit. on pp. 47, 48).

[MPS20] P. Mundhenk, E. Parodi, R. Schabenberger. "Fusion: A Safe and Secure Software Platform for Autonomous Driving". In: *2nd International Workshop on Autonomous Systems Design (ASD 2020)*. Ed. by S. Steinhorst, J. V. Deshmukh. Vol. 79. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 2:1–2:6. ISBN: 978-3-95977-141-2. DOI: 10.4230/OASIcs.ASD.2020.2. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.ASD.2020.2 (cit. on p. 35).

[OC13] N. O'Leary, D. Conway-Jones. "Node-RED: A Visual Tool for Wiring the Internet of Things". In: *IBM Journal of Research and Development* (2013). Accessed: 2024-10-01. URL: https://knolleary.net/2013/09/26/node-red-a-visual-tool-for-wiring-the-internet-of-things/ (cit. on p. 81).

[OPV+21] B. J. Oakes, A. Parsai, S. Van Mierlo, S. Demeyer, J. Denil, P. De Meulenaere, H. Vangheluwe. "Improving digital twin experience reports". In: *9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*. 2021, pp. 1–10. URL: https://repository.uantwerpen.be/link/irua/177786 (cit. on pp. 48, 49).

[Ora24] Oracle. *Getting Started with Java Message Service (JMS)*. 2024. URL: https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html (cit. on p. 93).

[Pal+21] G. Paltoglou et al. "The Behavioral Diversity of Java JSON Libraries". In: *arXiv preprint arXiv:2104.14323* (2021). URL: https://arxiv.org/pdf/2104.14323 (cit. on p. 71).

[PMZ21] M. Picone, M. Mamei, F. Zambonelli. "WLDT: A general purpose library to build IoT digital twins". In: *SoftwareX* 13 (2021), pp. 100661–100668. DOI: 10.1016/j.softx.2021.100661. URL: https://www.softxjournal.com/article/S2352-7110%2821%2900006-6/pdf (cit. on p. 43).

[RGS+20] L. A. Rosero, I. P. Gomes, J. A. R. da Silva, T. C. dos Santos, A. T. M. Nakamura, J. Amaro, D. F. Wolf, F. S. Osório. "A Software Architecture for Autonomous Vehicles: Team LRM-B Entry in the First CARLA Autonomous Driving Challenge". In: *Proceedings of the 2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 123–130. URL: https://arxiv.org/abs/2010.12598 (cit. on p. 46).

[RMV+20] L. F. Rivera, H. A. Müller, N. M. Villegas, G. Tamura, M. Jiménez. "On the Engineering of IoT-Intensive Digital Twin Software Systems". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW)*. 2020, pp. 631–638. URL: https://dl.acm.org/doi/10.1145/3387940.3392195 (cit. on pp. 21, 43).

[Rob23] J. Robles. "OpenTwins: An open-source framework for the development of next-gen compositional digital twins". In: *Computers in Industry* 152.1 (2023), p. 104007. DOI: 10.1016/j.compind.2023.104007 (cit. on p. 77).

[SH] K. Söderby, J. Hylén. *How to upload a sketch with the Arduino IDE 2*. URL: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started/ide-v2-uploading-a-sketch (cit. on p. 172).

[SKZ+20] A. Sharma, E. Kosasih, J. Zhang, A. Brintrup, A. Calinescu. "Digital Twins: State of the Art Theory and Practice, Challenges, and Open Research Questions". In: *Journal of Industrial Information Integration* 30 (2020), p. 100383. URL: https://www.sciencedirect.com/science/article/pii/S2452414X22000516 (cit. on p. 52).

[SPV18] A. C. Serban, E. Poll, J. Visser. "A Standard Driven Software Architecture for Fully Autonomous Vehicles". In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2018, pp. 1–8. URL: https://ieeexplore.ieee.org/document/8432195 (cit. on p. 46).

[SSK+23] T. Samak, C. Samak, S. Kandhasamy, V. Krovi, M. Xie. "AutoDRIVE: A Comprehensive, Flexible and Integrated Digital Twin Ecosystem for Autonomous Driving Research amp; Education". In: *Robotics* 12.3 (May 2023), p. 77. ISSN: 2218-6581. DOI: 10.3390/robotics12030077. URL: http://dx.doi.org/10.3390/robotics12030077 (cit. on p. 44).

[Sta24] StackShare. *What is MQTT and what are its top alternatives?* 2024. URL: https://stackshare.io/mqtt/alternatives (cit. on p. 80).

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, New York, 1973. URL: https://archive.org/details/Stachowiak1973AllgemeineModelltheorie (cit. on p. 25).

[Stü23] D. Stürner. “Generating Code for Distributed Deployments of Cyber-Physical Systems Using the MechatronicUML”. Master’s Thesis. University of Stuttgart, 2023 (cit. on p. 62).

[Sur] A. Suri. *"What Is Autonomous Driving? What Are Self-Driving Cars?"* URL: https://www.ezoomed.com/blog/ev-knowledge/what-is-autonomous-driving/ (cit. on p. 27).

[TKZS16] Ö. Ş. Taş, F. Kuhnt, J. M. Zöllner, C. Stiller. “Functional system architectures towards fully automated driving”. In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2016, pp. 304–309. URL: https://ieeexplore.ieee.org/abstract/document/7535402 (cit. on pp. 37, 38).

[TLZ+19] F. Tao, W. Liu, M. Zhang, T.-l. Hu, Q. Qi, H. Zhang, F. Sui, T. Wang, H. Xu, Z. Huang, et al. “Five-dimension digital twin model and its ten applications”. In: *Computer Integrated Manufacturing Systems* 25.1 (2019), pp. 1–10. URL: https://www.researchgate.net/publication/331095580_Five-dimension_Digital_Twin_Model_and_its_Ten_Applications_shuziluanshengwuweimoxingjishidalingyuyingyong (cit. on pp. 32, 39, 54, 59–65, 146).

[VYBW20] G. Velasco-Hernandez, D. J. Yeong, J. Barry, J. Walsh. “Autonomous Driving Architectures, Perception and Data Fusion: A Review”. In: *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*. 2020, pp. 315–321. DOI: 10.1109/ICCP51029.2020.9266268 (cit. on p. 46).

[WBB+21] A. Wortmann, T. Bolender, G. Bürvenich, M. Dalibor, B. Rumpe. “Self-Adaptive Manufacturing with Digital Twins”. In: *arXiv preprint arXiv:2103.11941* (2021). URL: https://arxiv.org/abs/2103.11941 (cit. on pp. 54, 194).

[WBD+20] A. Wortmann, P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz. “Model-Driven Development of a Digital Twin for Injection Molding”. In: June 2020, pp. 85–100. ISBN: 978-3-030-49434-6. DOI: 10.1007/978-3-030-49435-3_6. URL: https://www.researchgate.net/publication/341844679_Model-Driven_Development_of_a_Digital_Twin_for_Injection_Molding (cit. on pp. 19, 30, 31, 39, 42, 53, 54, 59–62, 64, 65, 193).

[WEB+21] A. Wortmann, R. Eramo, F. Bordeleau, B. Combemale, M. Brand, M. Wimmer. “Conceptualizing Digital Twins”. In: *IEEE Software* PP (Nov. 2021). DOI: 10.1109/MS.2021.3130755 (cit. on pp. 26, 28, 29, 39, 42, 53, 59–63, 65, 193).

[WKM+20] A. Wortmann, J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga. *Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems*. Oct. 2020. URL: https://www.se-rwth.de/publications/Model-driven-Digital-Twin-Construction-Synthesizing-the-Integration-of-Cyber-Physical-Systems-with-Their-Information-Systems.pdf (cit. on p. 19).

[Wor] A. Wortmann. *Research*. https://awortmann.github.io/research/. See section “Digital Twins“ (cit. on pp. 26, 27).

[WYL+23] K. Wang, T. Yu, Z. Li, K. Sakaguchi, O. Hashash, W. Saad. “Digital Twins for Autonomous Driving: A Comprehensive Implementation and Demonstration”. In: *2024 International Conference on Information Networking (ICOIN)* (2023), pp. 452–457. URL: https://api.semanticscholar.org/CorpusID:267028122 (cit. on p. 44).

[YCT+22] B. Yu, C. Chen, J. Tang, S. Liu, J.-L. Gaudiot. *Autonomous Vehicles Digital Twin: A Practical Paradigm for Autonomous Driving System Development*. Aug. 2022. DOI: http://dx.doi.org/10.1109/MC.2022.3159500 (cit. on p. 27).

All links were last followed on November 25, 2024.

A Supplementary Material for the Robot Cars

This appendix provides supplementary information on the Robot Cars (see Figure 6.1 and Section A.4), utilized for the prototype implementation discussed in this thesis (refer to Chapter 6). Although these Robot Cars were not specifically designed for this thesis, certain software features were integrated to support the implementation objectives. Detailed information can be found in Section 6.2.

A.1 Robot Car Hardware Specification

The Robot Cars were constructed using the hardware available in the Software Quality and Architecture Group's RoboLab at the University of Stuttgart[1]. Both Robot Cars utilize identical hardware components and code, albeit with different hardcoded values for motor, servo and sensor configurations. In particular, each Robot Car utilizes the following hardware components, each with its specific functions:

- **Chassis**: A SunFounder PiCar-X AI Car Kit, model CN0351D, version 2023.03.21, enhanced with a customized 3D-printed vehicle frame.
- One **Arduino Mega 2560 Rev3 Microcontroller[2]**: Serves as the ECU for managing the Robot Car's driving behaviour by processing sensor data and directing the motor controller. This microcontroller board is based on the ATmega2560 and features 54 digital input/output pins (15 of which can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It includes all necessary components to support the microcontroller, requiring only a USB cable connection to a computer, or an AC-to-DC adapter or battery for power. The Mega 2560 is an updated version of the Arduino Mega.
- One **Arduino Nano Microcontroller[3]**: Initially intended to function as the ECU for coordinating behaviour with other cars via the WiFi module in a different project, but is bypassed for this project and thus remains unused. The Arduino Nano is a compact, complete, and breadboard-friendly board based on the ATmega328 (Arduino Nano 3.x), equipped with a Mini-B USB connector.
- One **WiFi Shield ESP8266-01S Module**: Includes a breakout board for connecting to an existing WiFi network. Refer to Subsection A.3.2 for network configuration details, including SSID, password, MQTT server, and debug messages.

[1] https://github.com/SQA-Robo-Lab

[2] https://docs.arduino.cc/hardware/mega-2560

[3] https://docs.arduino.cc/hardware/nano

- Two **DC Motors**: Basic DC motors with attached wheels to propel the Robot Car. They operate based on the supplied voltage (forward and reverse depending on polarity), with higher current resulting in increased speed.
- One **L298N Motor Controller Module**: Connects the DC motors to an external power source and controls their speed and direction. It also provides a 5V output.
- One **Clutch Gear Digital Servo (SF006C)**: Operates at 5V power, with dimensions of 32.5mm x 16.3mm x 27.3mm, and is mounted at the front-middle of the Robot Car for steering.
- One **Ultrasonic Sonar Distance Sensor (HC-SR04)**: Operates at 5V power, with a detection range of 2cm to 400cm, optimal performance between 10cm and 250cm, and dimensions of 45mm x 20mm x 15mm. It is mounted at the front to detect and measure the distance to objects in the car's path.
- One **3-Channel Grayscale Sensor (KY-033)**: Operates at 5V power, with a detection range of 2cm to 40cm and dimensions of 42mm x 11mm x 12mm. It is mounted at the front of the Robot Car to detect lines on the ground via reflection, determining the car's position relative to the line (left, right, or on the line).
- One **Rechargeable 12V Battery**: Provides power to all components of the Robot Car and includes a low voltage checker for convenience.
- **Additional Components**: Includes cables, small breadboards for assembly, and glue.

The data sheets for the specific resources are accessible online.

The Arduino microcontrollers are programmed utilizing a subset of C++ and C programming language features [Ard] [SH]. Although they lack a series number, each is equipped with a unique serial number associated with the FTDI chip (USB/serial interface) on the board. In light of the absence of a series number, the Robot Car expert plans to assign an ID to each Robot Car and record it in the EEPROM as a substitute for a series number.
In addition to the previously mentioned plans, there is also an intention to integrate a speed sensor in the future. No camera sensor was implemented in the Robot Car (the one available in the Robot Car Kit is only compatible with Raspberry Pi).

The Robot Cars operate using essentially the same code, but require distinct hardcoded values for motors, servos, and other components.

The Robot Car expert noted that while there are minimal differences in the motors, which are generally imperceptible, there is a significant variation in the servos. Therefore, when loading the code onto the respective vehicle, it is necessary to comment out the sections of code pertinent to the other Robot Car.

For more information on the capabilities and limitations on the Robot Car kindly refer to Section 6.2.

Consequently, due to the time constraints faced by the Robot Car expert, the Physical Twin was directly mapped into the Digital Twin without any hardware modifications.

A.2 Robot Car Configuration Data

Listing A.1 presents the used initial configuration data in JSON format, specifically for Robot Car 1, presented for reasons of space and clarity, as the data for Robot Car 2 is identical except for the ID, MQTTTopics, and color. The second Robot Car has the ID `2`, the MQTTTopics `Car-2-In`, `Car-2-Out`, and `Car-2-Debug`, and the color `red`.


```
{
    "vehicles": [
        {
            "ID": 1,
            "revision": "1",
            "isCalibrated": true,
            "seriesnumber": "no information about this",
            "functionalities": [
                { "ID": 1, "name": "PauseOnObstacle", "isActive": false, "status": "OK" },
                { "ID": 2, "name": "StayOnTrack",     "isActive": false, "status": "OK" }
            ],
            "software": {
                "coding": {
                    "language": "Simplified and user-friendly version of C++",
                    "name": "Arduino Programming Language",
                    "version": "no information about this",
                    "softwareversion": "1.8.19"
                },
                "application": {
                    "name": "no information about this",
                    "version": "no information about this"
                },
                "transmission": {
                    "name": "no information about this",
                    "version": "no information about this"
                },
                "wifi": {
                    "network": "eduroam", "MQTTServer": "devonport.informatik.uni-stuttgart.de",
                    "SSID": "Robocars", "password": "Sofdcar-HAL"
                },
                "MQTTTopics": [
                    { "topic": "Car-1-In"     },
                    { "topic": "Car-1-Out"    },
                    { "topic": "Car-1-Debug"  }
                ]
            },
            "hardware": {
                "KIT": {
                    "name": "PiCar-X AI Car Kit",
                    "manufacturer": "SunFounder",
                    "model": "CN0351D",
                    "version": "2023.03.21"
                },
                "color": "blue",
                "carWheelCircumference": 204,
                "carWidthInMm": 170,
                "carLengthInMm": 170,
                "carWidthInMmWheelMidToWheelMid": 120,
                "carLengthInMmWheelMidToWheelMid": 100,
```

```
        "battery": { "status": "OK", "level": 100 },
        "ECUs": [
            { "name": "Arduino Mega 2560", "revision": "Rev3" },
            { "name": "Arduino Nano", "revision": "no information about this" },
            { "name": "WIFI MODULE ESP8266-01S", "revision": "EX" }
        ],
        "actuators": {
            "motors": [
                {
                    "ID": 1,
                    "name": "MotorDriverModule",
                    "usagesInVehicle": [ "locomotion" ],
                    "attachmentOnVehicle": "back-right_1",
                    "operation": "motorDriver",
                    "type": "DC",
                    "partNumber": "L298N",
                    "power": "5V",
                    "dimensionsInMm": [ 43, 43, 26 ],
                    "status": "OK",
                    "speedCmPerSecond": 0,
                    "minSpeedCmPerSecond": -150,
                    "maxSpeedCmPerSecond": 150,
                    "brakeQuality": 1.0,
                    "accelerationQuality": 1.0,
                    "driveControllerState": "STOPPED"
                },
                {
                    "ID": 2,
                    "name": "MotorDriverModule",
                    "usagesInVehicle": [ "locomotion" ],
                    "attachmentOnVehicle": "back-left_1",
                    "operation": "motorDriver",
                    "type": "DC",
                    "partNumber": "L298N",
                    "power": "5V",
                    "dimensionsInMm": [ 43, 43, 26 ],
                    "status": "OK",
                    "speedCmPerSecond": 0,
                    "minSpeedCmPerSecond": -150,
                    "maxSpeedCmPerSecond": 150,
                    "brakeQuality": 1.0,
                    "accelerationQuality": 1.0,
                    "driveControllerState": "STOPPED"
                }
            ],
            "servos": [
                {
                    "ID": 1,
                    "name": "ClutchGearDigitalServo",
                    "usagesInVehicle": [ "steering" ],
                    "attachmentOnVehicle": "front-middle_1",
                    "operation": "DigitalServo",
                    "type": "ClutchGear",
                    "partNumber": "SF006C",
                    "power": "5V",
                    "dimensionsInMm": [ 32.5, 16.3, 27.3 ],
                    "status": "OK",
```

```
                        "angle": 0,
                        "maxLeftAngle": -35,
                        "maxRightAngle": 35
                    }
                ]
            },
            "sensors": [
                {
                    "ID": 1,
                    "name": "UltrasonicSonarDistanceSensor",
                    "usagesInVehicle": [ "distance_front" ],
                    "attachmentOnVehicle": "front_1",
                    "operation": "sonar",
                    "type": "ultrasonic",
                    "partNumber": "HC-SR04",
                    "detectionRangeInCm": [ 2, 400 ],
                    "bestResultsInRangeInCm": [ 10, 250 ],
                    "power": "5V",
                    "dimensionsInMm": [ 45, 20, 15 ],
                    "status": "OK",
                    "distanceToClosestMm": 0,
                    "angleToClosest": 0,
                    "rawSensorData": 0
                },
                {
                    "ID": 2,
                    "name": "3chGrayScaleSensor",
                    "usagesInVehicle": [ "lineTracking" ],
                    "attachmentOnVehicle": "front_1",
                    "operation": "grayscale",
                    "type": "3 channel",
                    "partNumber": "KY-033",
                    "detectionRangeInCm": [ 2, 40 ],
                    "power": "5V",
                    "dimensionsInMm": [ 42, 11, 12 ],
                    "status": "OK",
                    "blackThreshold": 0,
                    "whiteThreshold": 1,
                    "linePositionMm": 0,
                    "legacyPosition": "LINE_POSITION_UNKNOWN",
                    "lineAngle": 0,
                    "numberOfSensors": 3,
                    "rawSensorData": [ 0, 0, 0 ],
                    "sensorDistanceInMm": 16,
                    "lineToFollow": 1
                }
            ]
        }
    },
    {
        "ID": 2,
   ...
    }
  ]
}
```


Listing A.1: Initial Configuration Data for Robot Car 1 in JSON Format

The initial configuration data comprises only those elements for which information is available. All other elements are excluded and not implemented. Since the most relevant elements have already been explained in Subsection 6.4.4 and most of the other elements are assumed to be self-explanatory, only the remaining most significant elements will be elaborated upon in the following.

To identify the corresponding hardware components, the JSON fields `usagesInVehicle` and `attachmentOnVehicle` are utilized for all hardware elements.

Although the specification for the Robot Cars (see Section A.1) lists the two `DC motors` separately from the `L298N Motor Controller Module`, for simplicity in the Digital Twin, each of the two `DC motors` has been integrated with the `L298N Motor Controller Module` in the configuration data, as well as in the internal structure of the Digital Twin. Setting a speed value not equal to zero changes the `driveControllerState` to "DRIVING", while a value of zero changes the `driveControllerState` to "STOPPED", unless the PT's functionality "Pause on Obstacle" is active, which might set the `driveControllerState` to "PAUSED" depending on the occurence of a possible obstacles ahead of the respective vehicle. The parameters `minSpeedCmPerSecond` and `maxSpeedCmPerSecond` represent speed boundaries that can be set. However, these values do not ensure that the car will move at every specified speed. According to the Robot Car expert, inclines, minor surface irregularities, and battery levels affect the power with which the `DC motors` operate, and thus they may not always be strong enough to ensure actual movement. Values for `brakeQuality` and `accelerationQuality` are relevant for the `AdaptiveCruiseControl` scenario, and they are provisionally set to 1.0.

`maxLeftAngle` and `maxRightAngle` are angle boundaries, as provided by the Robot Car expert.

Although the current implementation of this thesis employs only the steering servo, servos have a multitude of applications beyond steering, serving numerous other critical functions in modern vehicles. For example, servos are capable of regulating throttle control in internal combustion engines and adjusting power delivery in electric vehicles. Furthermore, servos are integral to transmission shifting, enabling smooth gear changes by controlling hydraulic pressure. In climate control systems, servos adjust air vents, blend doors, and temperature settings, ensuring optimal passenger comfort. Additionally, servos are utilized in seat adjustment mechanisms, enabling the forward, backward, or tilting of seats. Servos also adjust side mirrors for enhanced visibility and manage headlight levelling in vehicles with adaptive headlights, adjusting the angle based on speed and road conditions.
In conclusion, should future work on the prototype include the integration of these additional servo functions, the well-designed structure of the configuration data would facilitate their straightforward incorporation. This principle applies not only to servos but extends to all other elements within this configuration data JSON format.

Regarding the Ultrasonic Sonar Distance Sensor, `distanceToClosestTMm` describes the measured distance to objects in the Robot Car's path in millimeters. The value 65.535 denotes infinity, as this is the maximum value of an unsigned 16-bit integer.

Regarding the 3-Channel Grayscale Sensor, `blackThreshold` numerically defines what the sensor should identify as black and thus as a road track, while `whiteThreshold` numerically defines what the sensor should identify as white and thus as the environment, i.e. not the road track. `linePositionMm` numerically describes the sensor's position relative to the road track in millimeters. The value 16 for `linePositionMm` means that the road track is outside the array range, indicating that the line's position is unknown. Additionally, `legacyPosition` describes, in strings, the sensor's position relative to the road track.

The possible values are:
"LEFT_OF_LINE", "ON_LINE", "RIGHT_OF_LINE", "LINE_POSITION_UNKNOWN".
lineAngle describes the angle of the sensor relative to the road track. sensorDistanceInMm describes the distance between the three sensors (corresponding to the element numberOfSensors) in millimeters. linetoFollow specifies which edge of the line the sensor should align with. A value of 0 indicates the left edge, while a value of 1 indicates the right edge. The element is named linetoFollow because, at a junction, two road track diverge and, accordingly, the Robot Car would ultimately follow the edge of the road track it is aligned with.

angleToClosest and lineAngle are always set to zero because the sensors do not measure angles. These values serve as placeholders in the Hardware Abstraction Library for potential future hardware upgrades where sensors will be capable of measuring angles.

rawSensorData describes the respective raw sensor data of the sensors. However, these data are not provided by the PT due to time constraints faced by the Robot Car expert, which hindered their implementation.

In the absence of data regarding the application and transmission processes, these elements have been designated as placeholders, as they remain relevant to the configuration data.

To enhance memory efficiency, some elements of the Robot Cars are transmitted as numeric values, whereas a string assignment would be more comprehensible in the DT. Consequently, certain elements transmitted to the Digital Twin have been converted from numeric values to strings for clarity.

For the driveControllerState, the following mappings apply:

0 corresponds to DRIVE_STATE_STOPPED

1 corresponds to DRIVE_STATE_PAUSED

2 corresponds to DRIVE_STATE_DRIVING

In the DT, for LocomotionMotor objects, which correspond to the DC motors in this context, the driveControllerState is stored as the booleans DRIVE_STATE_STOPPED, DRIVE_STATE_PAUSED, and DRIVE_STATE_DRIVING to maintain consistency with the content structure of the Robot Car coding. However, in the configuration data JSON-string message, the driveControllerState is derived for simplicity as a string, corresponding to either "STOPPED", "PAUSED", or "DRIVING".

The legacyPosition determines the relative position to the road track. These values have been converted to strings for better user readability in the Dashboard.

0 corresponds to LEFT_OF_LINE

1 corresponds to ON_LINE

2 corresponds to RIGHT_OF_LINE

3 corresponds to LINE_POSITION_UNKNOWN

A.3 Communication with Robot Car

A.3.1 Robot Car Command Structure

This section provides a detailed explanation of the types of commands that can be sent to the Robot Cars, the required formats for these commands, and the corresponding responses or messages that will be sent from the Robot Cars. The detailed specifications of the functionalities and methods provided by the Robot Cars, as utilized in this context, are elaborated upon in Section A.2.

Each command sent to a Robot Car adheres to a structured pattern to ensure clarity and consistency, allowing for precise control and monitoring of the Robot Car's functionalities, as well as facilitating the addition of new commands with ease.

Every message must be sent to the topics `Car-1-In` / `Car-2-In`, respectively.
The responses and subscription-based request messages will be sent by the Robot Car to `Car-1-Out` / `Car-2-Out`, respectively. The content of the message must be sent in the MQTT payload. To optimize storage space, the HAL's RPC library formats the MQTT message payload as a buffer, requiring numerical values to be transmitted as ASCII. This method reduces the memory footprint and simplifies parsing, ensuring efficient data handling and storage for the Arduino microcontroller in the respective Robot Car [Ard23] [Cur23].

The HAL contains a DriveController, which has a driveControllerState implicitly set at any time, which is `DRIVE_CONTROLLER_STATE_DRIVING`, `DRIVE_CONTROLLER_STATE_PAUSED`, or `DRIVE_CONTROLLER_STATE_STOPPED`. For this reason, there are the following 3 fixed commands. Here, "dc" stands for "driveController", indicating its affiliation to the functionality, the string after the dot specifies the specific action, and the parameters, if present, provide the necessary values for the action.

- `dc.drive(angleSpeed)`
  This command initiates the drive mode of the Robot Car. The `angleSpeed` parameter is a concatenation of two values: an 8-bit ASCII value representing the angle, followed directly by a 16-bit ASCII value representing the speed. This allows for precise control over both the direction and velocity of the Robot Car.

- `dc.pause()`
  This command pauses the Robot Car's movement implicitly by setting its speed to 0. Importantly, it retains the previous speed value, enabling the Robot Car to resume its movement with the same speed when the `resume()` command is issued.

- `dc.resume()`
  This command resumes the Robot Car's movement, restoring the speed to the value it had before the pause. This ensures a seamless continuation of the Robot Car's operation without the need to reconfigure the speed.

All other commands can be categorized as *set* and *subscription* commands.
Set commands set specific values or functionalities in the Robot Cars and therefore play a crucial role in defining the behaviour and actions of the Robot Cars.
Subscription commands allow one to subscribe to a specific element such that the Robot Car sends

updates on that element either periodically or when triggered by a change.
The commands follow a structured pattern such that each segment of the command conveys specific instructions regarding the command category, i.e. set command or subscription, frequency, and the properties involved. Therefore, the patterns for *set* and *subscription* messages are presented below as regular expressions with respective breakdowns of the corresponding regular expression:

Listing A.1 Regular expression for *set*

```
^[A-Za-z]+(\.[A-Za-z0-9]+)?\.set[A-Z][a-zA-Z]*\((?:[0-9A-Fa-f]{4}|[0-9A-Fa-f]{2})\)$
```

- `^` asserts the position at the start of a line.
- `[A-Za-z]+` matches one or more letters (uppercase or lowercase). This denotes the affiliation to the corresponding functionality, for example, the "dc" part of the fixed commands presented above.
- `(\.[A-Za-z0-9]+)?` optionally matches a period, followed by one or more alphanumeric characters. The alphanumeric characters denote the high-level property related to the functionality of the *set* command. It also indicates where the subsequent method can be found.
- `\.set` matches the fixed ".set" part, which indicates a *set* command.
- `[A-Z][a-zA-Z]*` matches a string of letters and of arbitrary length where the first letter must be uppercase, while subsequent letters can be either uppercase or lowercase.
  This specifies the particular functionality or action to be set.
- `\(` matches the fixed "(" part necessary for potential parameters.
- `(?: ... )` is a non-capturing group.
- `[0-9A-Fa-f]{4}|[0-9A-Fa-f]{2}` matches either exactly four hexadecimal digits (representing a 16-bit ASCII number) or exactly two hexadecimal digits (representing an 8-bit ASCII number). This describes potentially existing parameters.
- `\)` matches the fixed ")" part necessary for potential parameters.
- `$` ensures that the entire string is matched from start to end.

Using this pattern, the available *set* commands per Robot Car, provided by the Robot Car expert and used in this thesis are the following:

- `dc.setAngle(x)`
- `dc.setSpeed(xx)`
- `shc.getLineFollower.setMode(x)`
- `shc.getLineFollower.setLineToFollow(x)`

Here, `x` is a placeholder for an 8-bit ASCII value, and `xx` is a placeholder for a 16-bit ASCII value.

Listing A.2 Regular expression for a *subscription*

```
^:sub([0-9A-Fa-f]{4}):([A-Za-z]+)(\.([A-Za-z0-9]+))?\.get[A-Z][a-zA-Z0-9]*\(\)$
```

- `^` asserts the position at the start of the string.
- `:sub` matches the fixed ":sub" part, which indicates a *subscription* command.
- `([0-9A-Fa-f]{4})` matches exactly four hexadecimal digits and captures them in a group, representing a 16-bit ASCII number. This specifies the frequency at which the message should be sent by the Robot Car in milliseconds. A value of 0_{10} indicates that the message should be sent as soon as a change in the value is detected.
- `:` matches a colon to indicate where the string for the affiliation to the corresponding functionality starts.
- `([A-Za-z]+)` matches one or more letters (uppercase or lowercase) and captures them in a group. This denotes the affiliation to the corresponding functionality, for example, the "dc" part of the fixed commands presented above.
- `(\.([A-Za-z0-9]+))?` optionally matches a period, followed by one or more alphanumeric characters. The alphanumeric characters are captured in a group and denote the high-level property related to the functionality of the subscription. It also indicates where the subsequent method can be found.
- `\.get[A-Z][a-zA-Z0-9]*\(\)` matches a literal period, followed by the string "get", followed by any uppercase letter, followed by any combination of uppercase letters, lowercase letters, and digits of arbitrary length (including zero length), followed by the literal parentheses "()". This specifies the method which describes the specific action or data that the subscription should return.
- `$` ensures that the entire string is matched from start to end.

Using this pattern, the available *subscription* commands per Robot Car, provided by the Robot Car expert and used in this thesis are the following:

- `:subxx:dc.getAngle()`
- `:subxx:dc.getSpeed()`
- `:subxx:dc.getState()`
- `:subxx:ds.getDistanceToClosestMm()`
- `:subxx:shc.getLineDetector.getLinePositionMm()`
- `:subxx:shc.getLineDetector.getLegacyPosition()`
- `:subxx:shc.getLineFollower.getMode()`

Each command message sent to the Robot Car is acknowledged by receiving a specific message exactly once as confirmation of successful processing:

Listing A.3 Format to acknowledge a *set* message

```
^return:set[A-Z][a-zA-Z0-9]*:$
```

In this case, `set[A-Z][a-zA-Z0-9]*` specifies the particular functionality or action to be set.

Listing A.4 Format to acknowledge a *subscription* message

```
^:sub:return:get[A-Z][a-zA-Z0-9]*$
```

Here, `get[A-Z][a-zA-Z0-9]*` describes the specific data that the subscription should return.

According to the Robot Car expert, if the received acknowledgement message does not conform to the specific format, this indicates a potential problem or bug in the Robot Car's Arduino Mega microcontroller, requiring a manual reset of the microcontroller.

Messages received as a result of subscribing to a functionality will be sent by a Robot Car in the form of a *get* message:

Listing A.5 Format of a *get* message

```
^return:get[A-Z][a-zA-Z0-9]*:(?:([0-9A-Fa-f]{4})|([0-9A-Fa-f]{2}))$
```

- `^` asserts the position at the start of the string.
- `return:` matches the fixed string "return:", which indicates the beginning of a return message.
- `get[A-Z][a-zA-Z0-9]*` matches the string "get", followed by an uppercase letter, and then any combination of uppercase letters, lowercase letters, and digits of arbitrary length (including zero length). This specifies the method which describes the specific data that the return message should convey.
- `:` matches a colon to indicate where the specific data starts.
- `(?:([0-9A-Fa-f]{4})|([0-9A-Fa-f]{2}))` is a non-capturing group that matches either exactly four hexadecimal digits, representing a 16-bit ASCII number, or exactly two hexadecimal digits, representing an 8-bit ASCII number. The digits are captured in separate groups. This specifies the specific data of the return message.
- `$` ensures that the entire string is matched from start to end.

Based on the established pattern, the messages that can be received as a result of subscribing to a functionality for each Robot Car (as provided by the Robot Car expert) and used in this thesis include the following:

- `return:getAngle:x`
- `return:getSpeed:xx`
- `return:getState:x`
- `return:getDistanceToClosestMm:xx`
- `return:getLinePositionMm:x`
- `return:getLegacyPosition:x`
- `return:getMode:x`

As previously mentioned, `x` is a placeholder for an 8-bit ASCII value, while `xx` is a placeholder for a 16-bit ASCII value.

The Arduino Mega is currently configured with a timeout of 500 milliseconds, allowing for the transmission of approximately 30 commands within this interval, following optimization. Consequently, the prototype DT of this thesis is capable of processing all subscriptions concurrently, given the present volume of commands.

Despite the technical feasibility of providing additional data (e.g. battery status, raw sensor data, etc.) via RPC, time constraints encountered by the Robot Car expert hindered its implementation.

Although the sensors' angles can be subscribed to, the sensors themselves do not measure angles. Therefore, the value is hardcoded to 0 in the Robot Cars. This serves as a placeholder in the Hardware Abstraction Library for potential future hardware upgrades where the sensors will be capable of measuring angles.

A.3.2 Robot Car Network Configuration

The network configuration for the Robot Cars is as follows:

- SSID: `Robocars`
- Password: `Sofdcar-HAL`
- MQTT Server: `devonport.informatik.uni-stuttgart.de`

The topics `Car-1-Debug` and `Car-2-Debug` are designated for the reception of debug data from Robot Cars 1 and 2, respectively. Upon transmission of the "Hello World" message by a debug topic, it can be inferred that the respective Robot Car has successfully established a connection with the network. It is imperative that the network be accessed via eduroam, as this will ensure that it is within the same network. The Robot Cars use MQTT 3.1.1 on the Arduino microcontrollers, and the Robot Car expert successfully utilized Mosquitto 2.0.18 with MQTT 5.0/3.1.1/3.1 on a laptop.

A.4 Supplementary Images for the Robot Cars

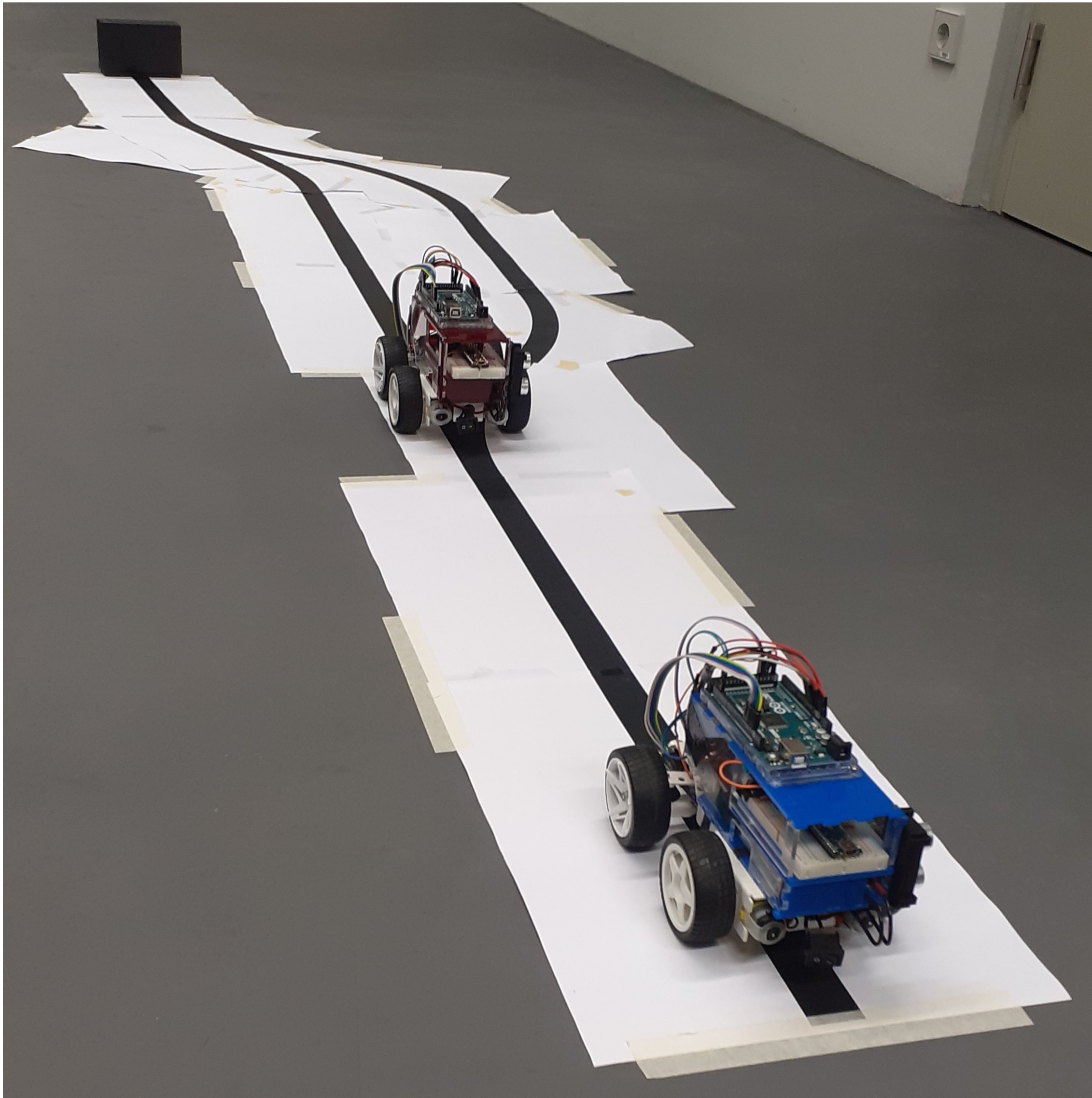

Figure A.1: Road Track with both Robot Cars

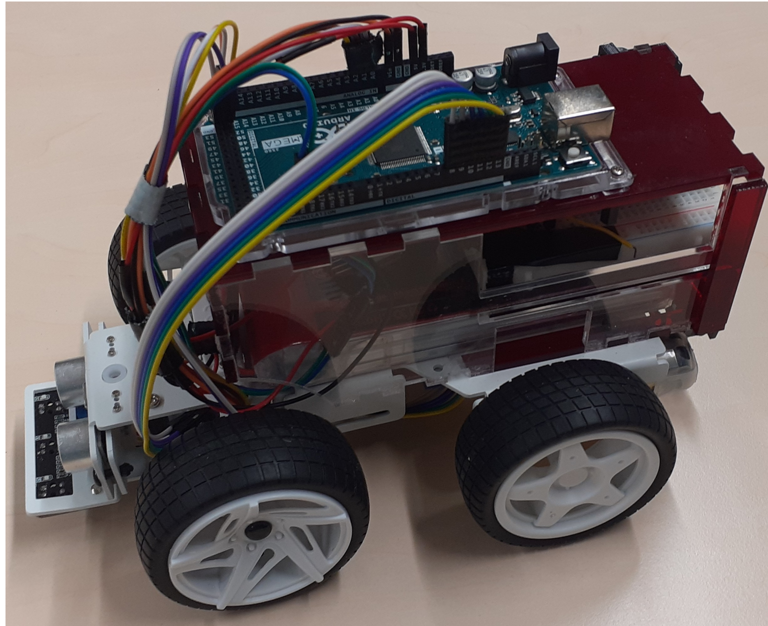

Figure A.2: Sideview of Red Robot Car

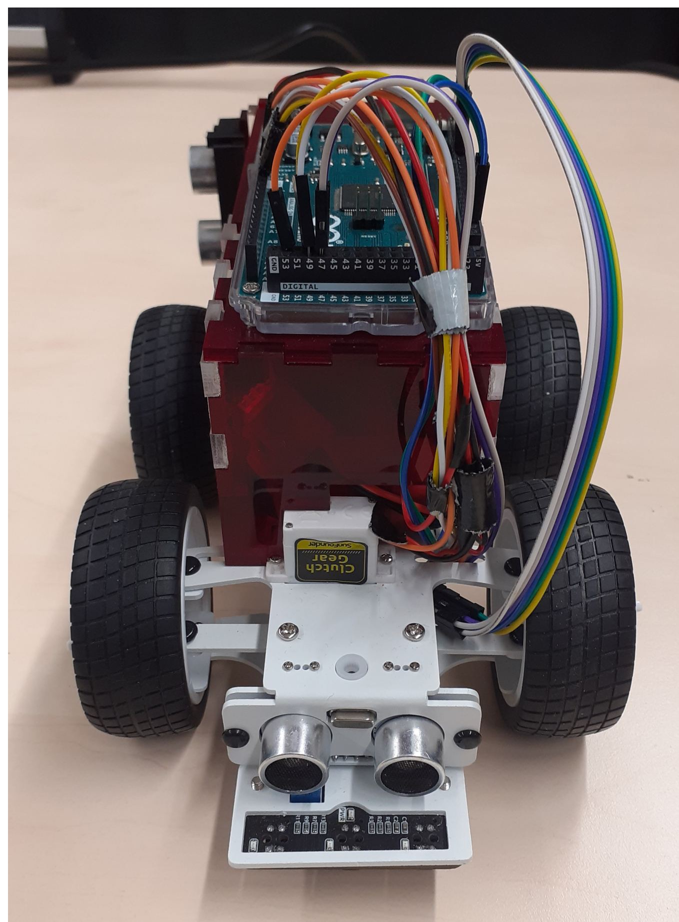

Figure A.3: Frontview of Red Robot Car

B Supplementary Material for the Software Tools

MQTT Broker:

- Mosquitto version 2.0.18 with MQTT version 5.0/3.1.1

Software for Data Manager, Dashboard and Database:

- `node-red` version 3.1.7
- `node-red-dashboard` version 3.6.5
- `node-red-node-sqlite` version 1.1.0
- `node-red-node-ui-table` version 0.4.3
- `node-red-contrib-ui-led` version 0.4.11

Software for the Digital Twin Entities:

- Java Development Kit (JDK) version 11.0.2
- Maven version 3.8.0 release 11

C Structure of the Prototype DT-Entities Implementation coded in Java

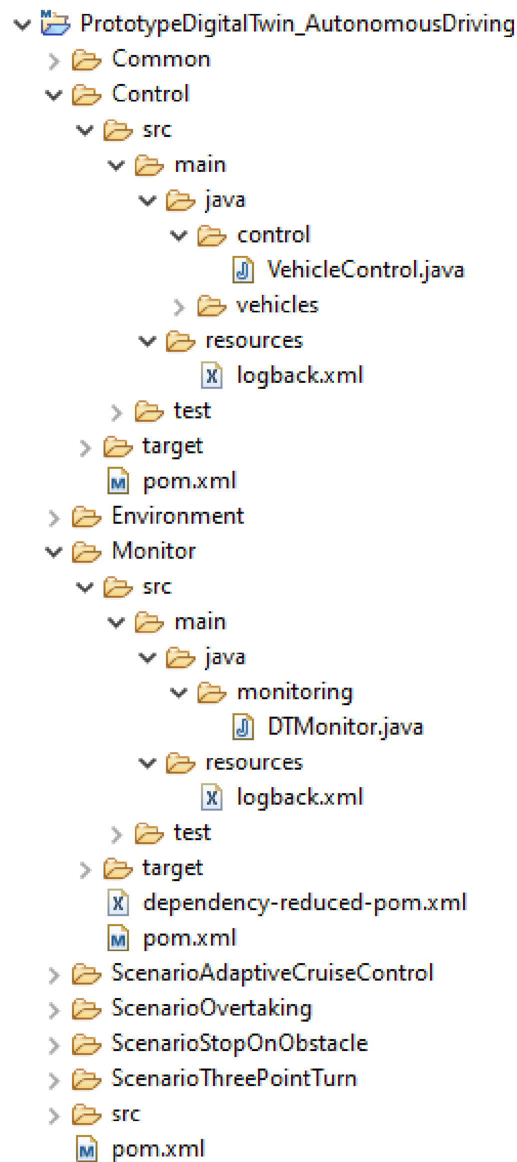

Figure C.1: Screenshot of the Package Explorer in Eclipse of the Prototype DT

View of the Eclipse package explorer illustrating the structure of the prototype DT-Entities implementation coded in Java shown in Figure C.1.

In a Maven-based Java project where multiple applications communicate via messaging, it is essential to manage these components efficiently.

In this thesis, the project has been structured into submodules within the parent Maven project, resulting in separate `JAR` files for each DT-Entity or DT-Entity-Scenario Java application. Thereby, the parent `pom.xml` includes the common configurations and dependencies, and each submodule has its own individual `pom.xml` files for specifying their unique dependencies and configurations. By packaging the project, distinct outputs (i.e. separate `JAR` files) for each submodule can be generated by Maven. This modular approach enhances maintainability, facilitates independent development, and allows for easier deployment of each application. Furthermore, having separate modules ensures that each part of the project can be managed, updated, and tested independently, leading to more efficient debugging and streamlined project management. Finally, multiple `JAR` files from different submodules can be efficiently managed and executed using a script.

The following structure adheres to best practices to provide a clear and logical organisation, facilitating the management and understanding of relationships between different components.

Common Package The Common Package contains shared classes utilized across multiple packages. This approach eases the management and access of shared resources without creating dependencies between unrelated packages. Given that the `TaskObject` class is used across multiple classes in different packages, it is placed within this Common Package.

Control Package The Control Package includes applications used as controlling entities. In this prototype DT, the `VehicleControl` class, responsible for managing vehicle objects, is the sole controlling entity and is therefore included within this Control Package. As `VehicleControl` is the only application to access the object classes (`Vehicle`, `Functionality`, `Hardware`, `Software`, etc.), these are also placed within this Control Package, but in separate folders/packages to maintain a correct hierarchical structure analogous to the one presented in A.2 and to keep the control logic distinct from the vehicle definitions. This arrangement keeps all vehicle-related classes together, making it clear that the classes `Hardware`, `Software`, and `Functionality` are part of an instance of the `Vehicle` class, and further divides the vehicles package into more specific packages like software and hardware, as well as more granular sub-packages for actuators, sensors, etc.

Environment Package The Environment Package contains the `Environment` class, which describes the environment in which the vehicles operate. By establishing this package, future work can include additional environmental classes, allowing each to have a dedicated environment description or related functionality. The separation into several environmental classes ensures the overall DT remains loosely coupled and thus more flexible

Monitor Package The Monitor Package includes the `DTMonitor` class, which monitors the operational status of various applications. This package structure allows for the addition of further monitoring classes in future work, with each class possessing dedicated functionality. This separation into multiple monitoring classes ensures the overall DT remains loosely coupled and therefore more adaptable

ScenarioAdaptiveCruiseControl Package This package contains the `AdaptiveCruiseControl` class, which executes adaptive cruise control, and the associated object class, stored in a dedicated folder, which it depends on to memorise the correct states for the respective vehicles

ScenarioOvertaking Package This package includes the `Overtaking` class, which executes the overtaking manoeuvre, and the associated object class, stored in a dedicated folder, which it depends on to memorise the correct states for the respective vehicles

ScenarioStopOnObstacle Package This package contains the `StopOnObstacle` class, which executes the stop on an obstacle at a specified distance, and the associated object class, stored in a dedicated folder, which it depends on to memorise the correct states and thresholds for the respective vehicles

ScenarioThreePointTurn Package This package comprises the `ThreePointTurn` class, which executes a basic three-point turn

Logging entails the documentation of runtime data, which serves as a valuable resource for debugging, monitoring, and maintaining applications.

In this thesis, `Logback` has been used for logging, with each submodule having its own `Logback` configuration file. Configuring `Logback` separately for each submodule enables the independent definition of logging levels, providing granular control over logging behaviour and tailoring the output to specific needs, thereby enhancing system transparency and debugging efficiency.

D Supplementary Information on Design Decisions for Dashboard

The choice of Node-RED for the dashboard in this thesis is justified by its user-friendly interface, low learning curve, and modular architecture, which facilitate the efficient creation of interactive and visually appealing dashboards. A Node-RED dashboard is structured hierarchically, consisting of tabs, groups, and widgets. Tabs function as the primary sections of the dashboard, akin to distinct pages in a web application, and each tab may encompass multiple groups. Within each tab, groups are employed to organize widgets into coherent segments, thereby enhancing the layout and user experience. Each group contains widgets, which are the individual elements such as charts, buttons, sliders, and gauges that display data or facilitate user interaction.

To ensure a clear and synchronous representation of both the PT and DT, the gauges and sliders for the basic driving capabilities in both groups, namely the PT Dashboard group and the DT Dashboard group, are kept identical. This arrangement also aids in distinguishing between PT control and DT control.

In consultation with the Robot Car expert, it was determined that the hardware quality could lead to unsuccessful subscription attempts, necessitating multiple message transmissions or manual resetting of the microcontroller. To address this issue, a manual subscription feature using the "SUBS" button has been implemented in place of a one-time automatic subscription.

The parameters `angleToClosest` (concerning the Ultrasonic Sonar Distance Sensor) and `lineAngle`, `linePositionMm` (concerning the 3-Channel Grayscale Sensor) are implemented in the DT but are not displayed on the Node-RED dashboard to conserve space, as their values are zero and remain unchanged since the sensor cannot recognise angles.

The parameters `batteryLevel`, `rawSensorData` (concerning the Ultrasonic Sonar Distance Sensor), and `rawSensorData` (concerning the 3-Channel Grayscale Sensor) are also implemented in the DT but are not shown on the Node-RED Dashboard to save space, as their values are not provided by the PT.

Due to space limitations in the PT Dashboard group, the activity status of the PT's functionalities "Pause on Obstacle" and "Stay on Track" is represented solely by an LED, where green indicates active and red indicates inactive. Consequently, the number corresponding to the "MODE" is displayed as a control view in the PT Dashboard group. However, displaying the `legacyPosition` (concerning the 3-Channel Grayscale Sensor) as LEDs instead of strings in the Dashboard (e.g. three horizontal LEDs indicating the legacy position of the respective Robot Car) would have been possible but was not implemented, as it would compromise the Dashboard's clarity and compactness.

The "CONFIG" button is located on the DT side because configuration is not required on the PT side.

E Summarized Considerations to RQ1.3

RQ1.3 : What concept and meta-model could be elaborated for a Digital Twin in the context of Autonomous Driving, with specific consideration given to the flexibility, composability and extendability of the proposed approach?

A Digital Twin for Autonomous Driving should be elaborated in ways to:

- have a complete separation of concerns as it simplifies composability and flexibility
- have a unified communication structure within the DT to help ensure an extendability with limited efforts
- be adaptive to a certain degree, meaning that the respective components of the DT should be able to respond to an input with a specific reaction, driven by a certain logic
- have short communication pathways as the duration of data exchange is crucial

It seems that a concept / meta-model with the following characteristics addresses the concerns of this thesis:

- a DT consisting of separate entities, each with its own concern, characterized by multiple attributes and associations
- Entities can communicate and interact with each other by sending data over directed connections
- these directed connections are characterized by knowing their recipient and possessing various attributes (e.g a message can be timed or triggered by a condition)
- the directed connection can be synchronous, implying tight coupling, or asynchronous, implying loose coupling, which is appropriate for maintaining the composability of the DT
- The data sent over the directed connection can serve various purposes, such as commands or requests, but can also include simple information like status messages, confirmations, or responses to requests
- derived from [WEB+21], each entity can have one to three roles: descriptive, predictive and/or prescriptive
- additionally, an entity can possess, derived from [WBD+20], one to four functionalities: DataProcessor, Evaluator, Reasoner and/or Executor
- the DataProcessor functionality refers to the entity's ability to process the data in a way that makes it usable within the entity
- messages pertinent to a specific entity typically contain information, data, updates, and often require corresponding actions. Evaluator refers to the entity's ability to evaluate received data

- the subsequent action is determined by the intelligence of an entity, referred to as the Reasoner
- there are multiple potential approaches to reasoning
- in a manual system, a user receives the necessary evaluated information and subsequently specifies the appropriate consequence or action
- for greater autonomy, a predefined system where reasoning is conditionally based on pre-specified possibilities for evaluation results, would/could be involved
- for full autonomy, the reasoning would be based on a self-adapting system that can operate using, for example, CBR (see [WBB+21]) or AI (deep learning, etc.)
- for the reasoning process, it may also be necessary to use additional knowledge beyond what is provided by a message. Accordingly, each entity incorporates a KnowledgeBase, which may take the form of access to a proper or shared database, or the local runtime memory of the entity in question
- the Executor delineates the functionality of an entity responsible for executing the actions determined by the Reasoner
- since each entity has its own scope and necessity by being separate from the other entities, each entity can be seen as a Service that might need or use another entity as a Service as well
- thus, an entity can provide multiple services
- to ensure that the entire DT operates correctly, each entity could further be defined by its nonfunctional requirements
- the status of an active DT is contingent upon the active status of its constituent entities. Therefore, each entity has a running state, which can be either Running or Not Running, indicating whether an entity is active or available within the DT

For the concept in this thesis, it is assumed that:

- the four functionalities (DataProcessor, Evaluator, Reasoner and/or Executor) may, but are not required to, reside within a single entity. This is contingent upon the specific use case of the Digital Twin.
- for establishing a feedback loop, the three functionalities, namely Evaluator, Reasoner, and Executor, are present for each entity, either all in one entity or one functionality used as a service or separate entity by the entity, to ensure feedback. This establishes an automatic two-way data flow, leading to the conclusion that this concept, according to Kritzinger et al. [KKT+18], indeed characterizes a Digital Twin, in comparison for example with shadowing only

The class diagram presented in Figure 5.1 outlines the structure and relationships between various entities/components within the composable, flexible and extendable Digital Twin. This comprehensive structure ensures a flexible system capable of adapting to various operational requirements.

F Summarized Considerations to the Use Case

Derived from the meta-model concept, the following components have been devised relevant for the architectural framework of the Use Case, which serves as an instance of the meta-model, i.e. a model:

- VehicleControl Entity,
  with roles: `Descriptive`, `Prescriptive`
  with functionalities: `DataProcessor`, `Executor`
  can be a `Service`
- DTMonitor Entity,
  with roles: `Descriptive`, `Prescriptive` and possibly `Predictive`
  with functionalities: `Evaluator`, `Reasoner`, `Executor`
  can be a `Service`
- Environment Entity,
  with roles: `Descriptive` and possibly `Predictive`
  with functionality: `DataProcessor` and possibly `Evaluator`
  can be a `Service`
- ThreePointTurn Entity-Scenario,
  with roles: `Prescriptive`
  with functionality: `Executor`
  can not be a `Service`
- StopOnObstacle Entity-Scenario,
  with roles: `Prescriptive`
  with functionalities: `Evaluator`, `Executor`
  can not be a `Service`
- AdaptiveCruiseControl Entity-Scenario,
  with roles: `Predictive`, `Prescriptive`
  with functionalities: `Evaluator`, `Reasoner`, `Executor`
  can be a `Service`
- Overtaking Entity-Scenario,
  with roles: `Predictive`, `Prescriptive`
  with functionalities: `Evaluator`, `Reasoner`, `Executor`
  can be a `Service`

These components are depicted in the architecture diagram presented in Figure 5.2 and Tables 5.1, 5.2 and 5.3.

Inter-Entities communication aspects (as described in Subsection 5.3.3):

- the Entities must be capable of inter-Entity communication, through directed asynchronous connections, which may be timed, triggered, conditional, or unconditional
- MQTT is utilized as the protocol for these interactions
- each scenario stores relevant information during runtime, correlated to the respective vehicle it concerns
- each scenario can both request information from the VehicleControl Entity regarding specific vehicles and receive responses, as well as issue commands to alter the values of individual vehicles, thereby changing the status of the DT
- as the AdaptiveCruiseControl Entity-Scenario relies heavily on environmental factors, it sends additionally a request to the Environment Entity, which supplies the relevant information in return
- all Entities send a heartbeat signal as a message to the DTMonitor Entity
- to ensure the effective and synchronized functioning of all Entities, the DTMonitor Entity regularly checks which Entities are active and send those heartbeats accordingly
- this comprehensive setup enables communication, data exchange and system monitoring, laying a foundation for the Digital Twin's performance and reliability

The necessary additional elements (as described in Subsection 5.3.4) are:

- a Data Manager to facilitate communication and interaction between the DT and the PT as well as with the User Interface and Database. It acts as a gateway, managing data flows, etc. It could also be created as an Entity as provided by the presented concept
- a Physical Twin, able to interact interactively with the DT
- a Message Broker to enable communication and coordination between DT and PT
- a User Interface (Dashboard) to manually control and visualise the states of both the DT and PT, as well as to switch operational modes via selector (shadowing, emulation, simulation, testing, etc.)
- a database is beneficial for ensuring and maintaining persistent storage of the states of the DT and PT, as well as to log data

G Supplementary Material for the Database

Table G.1 describes the SQLite tables of the DB_DT Database.

Name	Type	Schema
Actuators		CREATE TABLE Actuators (ID INTEGER PRIMARY KEY AUTOINCREMENT, vehicleID INTEGER, type TEXT, FOREIGN KEY (vehicleID) REFERENCES Hardware(vehicleID))
ID	INTEGER	"ID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
type	TEXT	"type" TEXT
Battery		CREATE TABLE Battery (vehicleID INTEGER PRIMARY KEY, status TEXT, level INTEGER, FOREIGN KEY (vehicleID) REFERENCES Hardware(vehicleID))
vehicleID	INTEGER	"vehicleID" INTEGER
status	TEXT	"status" TEXT
level	INTEGER	"level" INTEGER
Coding		CREATE TABLE Coding (vehicleID INTEGER PRIMARY KEY, language TEXT, name TEXT, version TEXT, softwareversion TEXT, FOREIGN KEY (vehicleID) REFERENCES Software(vehicleID))
vehicleID	INTEGER	"vehicleID" INTEGER
language	TEXT	"language" TEXT
name	TEXT	"name" TEXT
version	TEXT	"version" TEXT
softwareversion	TEXT	"softwareversion" TEXT

Name	Type	Schema
DistanceSensorChanges		CREATE TABLE DistanceSensorChanges (changeID INTEGER PRIMARY KEY AUTOINCREMENT, distanceSensorID INTEGER, vehicleID INTEGER, timestamp TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now')), name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, detectionRangeInCm TEXT, bestResultsInRangeInCm TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, distanceToClosestMm INTEGER, angleToClosest INTEGER, rawSensorData INTEGER, changedBy TEXT, FOREIGN KEY (distanceSensorID) REFERENCES DistanceSensors(ID), FOREIGN KEY (vehicleID) REFERENCES Sensors(vehicleID))
changeID	INTEGER	"changeID" INTEGER
distanceSensorID	INTEGER	"distanceSensorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
timestamp	TEXT	"timestamp" TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now'))
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
detectionRangeInCm	TEXT	"detectionRangeInCm" TEXT
bestResultsInRangeInCm	TEXT	"bestResultsInRangeInCm" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
distanceToClosestMm	INTEGER	"distanceToClosestMm" INTEGER
angleToClosest	INTEGER	"angleToClosest" INTEGER
rawSensorData	INTEGER	"rawSensorData" INTEGER
changedBy	TEXT	"changedBy" TEXT

Name	Type	Schema
DistanceSensors		CREATE TABLE DistanceSensors (ID INTEGER, sensorID INTEGER, vehicleID INTEGER, name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, detectionRangeInCm TEXT, bestResultsInRangeInCm TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, distanceToClosestMm INTEGER, angleToClosest INTEGER, rawSensorData INTEGER, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (sensorID) REFERENCES Sensors(sensorID), FOREIGN KEY (vehicleID) REFERENCES Sensors(vehicleID))
ID	INTEGER	"ID" INTEGER
sensorID	INTEGER	"sensorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
detectionRangeInCm	TEXT	"detectionRangeInCm" TEXT
bestResultsInRangeInCm	TEXT	"bestResultsInRangeInCm" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
distanceToClosestMm	INTEGER	"distanceToClosestMm" INTEGER
angleToClosest	INTEGER	"angleToClosest" INTEGER
rawSensorData	INTEGER	"rawSensorData" INTEGER
ECUs		CREATE TABLE ECUs (ID INTEGER PRIMARY KEY AUTOINCREMENT, vehicleID INTEGER, name TEXT, revision TEXT, FOREIGN KEY (vehicleID) REFERENCES Hardware(vehicleID))
ID	INTEGER	"ID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
revision	TEXT	"revision" TEXT
Functionalities		CREATE TABLE Functionalities (ID INTEGER, vehicleID INTEGER, name TEXT, isActive BOOLEAN, status TEXT, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (vehicleID) REFERENCES Vehicles(ID))
ID	INTEGER	"ID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER

Name	Type	Schema
name	TEXT	"name" TEXT
isActive	BOOLEAN	"isActive" BOOLEAN
status	TEXT	"status" TEXT
FunctionalityChanges		CREATE TABLE FunctionalityChanges (changeID INTEGER PRIMARY KEY AUTOINCREMENT, functionalityID INTEGER, timestamp TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f','now')), name TEXT, isActive BOOLEAN, status TEXT, changedBy TEXT, FOREIGN KEY (functionalityID) REFERENCES Functionalities(ID))
changeID	INTEGER	"changeID" INTEGER
functionalityID	INTEGER	"functionalityID" INTEGER
timestamp	TEXT	"timestamp" TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now'))
name	TEXT	"name" TEXT
isActive	BOOLEAN	"isActive" BOOLEAN
status	TEXT	"status" TEXT
changedBy	TEXT	"changedBy" TEXT
GrayscaleSensorChanges		CREATE TABLE GrayscaleSensorChanges (changeID INTEGER PRIMARY KEY AUTOINCREMENT, lineTrackingSensorID INTEGER, vehicleID INTEGER, timestamp TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now')), name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, detectionRangeInCm TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, blackThreshold INTEGER, whiteThreshold INTEGER, linePositionMm INTEGER, legacyPosition TEXT, lineAngle INTEGER, numberOfSensors INTEGER, rawSensorData TEXT, sensorDistanceInMm INTEGER, lineToFollow INTEGER, changedBy TEXT, FOREIGN KEY (lineTrackingSensorID) REFERENCES GrayscaleSensors(ID), FOREIGN KEY (vehicleID) REFERENCES Sensors(vehicleID))
changeID	INTEGER	"changeID" INTEGER
lineTrackingSensorID	INTEGER	"lineTrackingSensorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
timestamp	TEXT	"timestamp" TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now'))
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT

Name	Type	Schema
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
detectionRangeInCm	TEXT	"detectionRangeInCm" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
blackThreshold	INTEGER	"blackThreshold" INTEGER
whiteThreshold	INTEGER	"whiteThreshold" INTEGER
linePositionMm	INTEGER	"linePositionMm" INTEGER
legacyPosition	TEXT	"legacyPosition" TEXT
lineAngle	INTEGER	"lineAngle" INTEGER
numberOfSensors	INTEGER	"numberOfSensors" INTEGER
rawSensorData	TEXT	"rawSensorData" TEXT
sensorDistanceInMm	INTEGER	"sensorDistanceInMm" INTEGER
lineToFollow	INTEGER	"lineToFollow" INTEGER
changedBy	TEXT	"changedBy" TEXT
GrayscaleSensors		CREATE TABLE GrayscaleSensors (ID INTEGER, sensorID INTEGER, vehicleID INTEGER, name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, detectionRangeInCm TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, blackThreshold INTEGER, whiteThreshold INTEGER, linePositionMm INTEGER, legacyPosition TEXT, lineAngle INTEGER, numberOfSensors INTEGER, rawSensorData TEXT, sensorDistanceInMm INTEGER, lineToFollow INTEGER, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (sensorID) REFERENCES Sensors(sensorID), FOREIGN KEY (vehicleID) REFERENCES Sensors(vehicleID))
ID	INTEGER	"ID" INTEGER
sensorID	INTEGER	"sensorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
detectionRangeInCm	TEXT	"detectionRangeInCm" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT

Name	Type	Schema
blackThreshold	INTEGER	"blackThreshold" INTEGER
whiteThreshold	INTEGER	"whiteThreshold" INTEGER
linePositionMm	INTEGER	"linePositionMm" INTEGER
legacyPosition	TEXT	"legacyPosition" TEXT
lineAngle	INTEGER	"lineAngle" INTEGER
numberOfSensors	INTEGER	"numberOfSensors" INTEGER
rawSensorData	TEXT	"rawSensorData" TEXT
sensorDistanceInMm	INTEGER	"sensorDistanceInMm" INTEGER
lineToFollow	INTEGER	"lineToFollow" INTEGER
Hardware		CREATE TABLE Hardware (vehicleID INTEGER PRIMARY KEY, color TEXT, vehicleWheelCircumference INTEGER, vehicleWidthInMm INTEGER, vehicleLengthInMm INTEGER, vehicleWidthInMmWheelMidToWheelMid INTEGER, vehicleLengthInMmWheelMidToWheelMid INTEGER, FOREIGN KEY (vehicleID) REFERENCES Vehicles(ID))
vehicleID	INTEGER	"vehicleID" INTEGER
color	TEXT	"color" TEXT
vehicleWheelCircum.	INTEGER	"vehicleWheelCircumference" INTEGER
vehicleWidthInMm	INTEGER	"vehicleWidthInMm" INTEGER
vehicleLengthInMm	INTEGER	"vehicleLengthInMm" INTEGER
vehicleWidthInMmWheel.	INTEGER	"vehicleWidthInMmWheelMidToWheelMid" INTEGER
vehicleLengthInMmWheel.	INTEGER	"vehicleLengthInMmWheelMidToWheelMid" INTEGER
KIT		CREATE TABLE KIT (vehicleID INTEGER PRIMARY KEY, name TEXT, manufacturer TEXT, model TEXT, version TEXT, FOREIGN KEY (vehicleID) REFERENCES Hardware(vehicleID))
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
manufacturer	TEXT	"manufacturer" TEXT
model	TEXT	"model" TEXT
version	TEXT	"version" TEXT
MQTTTopics		CREATE TABLE MQTTTopics (ID INTEGER PRIMARY KEY AUTOINCREMENT, vehicleID INTEGER, topic TEXT, FOREIGN KEY (vehicleID) REFERENCES Software(vehicleID))
ID	INTEGER	"ID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
topic	TEXT	"topic" TEXT

Name	Type	Schema
MotorChanges		CREATE TABLE MotorChanges (changeID INTEGER PRIMARY KEY AUTOINCREMENT, motorID INTEGER, vehicleID INTEGER, timestamp TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f','now')), name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, speedCmPerSecond INTEGER, minSpeedCmPerSecond REAL, maxSpeedCmPerSecond REAL, brakeQuality REAL, accelerationQuality REAL, driveControllerState TEXT, changedBy TEXT, FOREIGN KEY (motorID) REFERENCES Motors(ID), FOREIGN KEY (vehicleID) REFERENCES Actuators(vehicleID))
changeID	INTEGER	"changeID" INTEGER
motorID	INTEGER	"motorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
timestamp	TEXT	"timestamp" TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now'))
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
speedCmPerSecond	INTEGER	"speedCmPerSecond" INTEGER
minSpeedCmPerSecond	REAL	"minSpeedCmPerSecond" REAL
maxSpeedCmPerSecond	REAL	"maxSpeedCmPerSecond" REAL
brakeQuality	REAL	"brakeQuality" REAL
accelerationQuality	REAL	"accelerationQuality" REAL
driveControllerState	TEXT	"driveControllerState" TEXT
changedBy	TEXT	"changedBy" TEXT

Name	Type	Schema
Motors		CREATE TABLE Motors (ID INTEGER, actuatorID INTEGER, vehicleID INTEGER, name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, speedCmPerSecond INTEGER, minSpeedCmPerSecond REAL, maxSpeedCmPerSecond REAL, brakeQuality REAL, accelerationQuality REAL, driveControllerState TEXT, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (actuatorID) REFERENCES Actuators(ID), FOREIGN KEY (vehicleID) REFERENCES Actuators(vehicleID))
ID	INTEGER	"ID" INTEGER
actuatorID	INTEGER	"actuatorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
speedCmPerSecond	INTEGER	"speedCmPerSecond" INTEGER
minSpeedCmPerSecond	REAL	"minSpeedCmPerSecond" REAL
maxSpeedCmPerSecond	REAL	"maxSpeedCmPerSecond" REAL
brakeQuality	REAL	"brakeQuality" REAL
accelerationQuality	REAL	"accelerationQuality" REAL
driveControllerState	TEXT	"driveControllerState" TEXT
Sensors		CREATE TABLE Sensors (ID INTEGER, vehicleID INTEGER, type TEXT, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (vehicleID) REFERENCES Hardware(vehicleID))
ID	INTEGER	"ID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
type	TEXT	"type" TEXT

Name	Type	Schema
ServoChanges		CREATE TABLE ServoChanges (changeID INTEGER PRIMARY KEY AUTOINCREMENT, servoID INTEGER, vehicleID INTEGER, timestamp TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f','now')), name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, angle INTEGER, maxLeftAngle REAL, maxRightAngle REAL, changedBy TEXT, FOREIGN KEY (servoID) REFERENCES Servos(ID), FOREIGN KEY (vehicleID) REFERENCES Actuators(vehicleID))
changeID	INTEGER	"changeID" INTEGER
servoID	INTEGER	"servoID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
timestamp	TEXT	"timestamp" TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now'))
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
angle	INTEGER	"angle" INTEGER
maxLeftAngle	REAL	"maxLeftAngle" REAL
maxRightAngle	REAL	"maxRightAngle" REAL
changedBy	TEXT	"changedBy" TEXT
Servos		CREATE TABLE Servos (ID INTEGER, actuatorID INTEGER, vehicleID INTEGER, name TEXT, usagesInVehicle TEXT, attachmentOnVehicle TEXT, operation TEXT, type TEXT, partNumber TEXT, power TEXT, dimensionsInMm TEXT, status TEXT, angle INTEGER, maxLeftAngle REAL, maxRightAngle REAL, PRIMARY KEY (ID, vehicleID), FOREIGN KEY (actuatorID) REFERENCES Actuators(ID), FOREIGN KEY (vehicleID) REFERENCES Actuators(vehicleID))
ID	INTEGER	"ID" INTEGER
actuatorID	INTEGER	"actuatorID" INTEGER
vehicleID	INTEGER	"vehicleID" INTEGER
name	TEXT	"name" TEXT
usagesInVehicle	TEXT	"usagesInVehicle" TEXT

Name	Type	Schema
attachmentOnVehicle	TEXT	"attachmentOnVehicle" TEXT
operation	TEXT	"operation" TEXT
type	TEXT	"type" TEXT
partNumber	TEXT	"partNumber" TEXT
power	TEXT	"power" TEXT
dimensionsInMm	TEXT	"dimensionsInMm" TEXT
status	TEXT	"status" TEXT
angle	INTEGER	"angle" INTEGER
maxLeftAngle	REAL	"maxLeftAngle" REAL
maxRightAngle	REAL	"maxRightAngle" REAL
Software		CREATE TABLE Software (vehicleID INTEGER PRIMARY KEY, FOREIGN KEY (vehicleID) REFERENCES Vehicles(ID))
vehicleID	INTEGER	"vehicleID" INTEGER
Vehicles		CREATE TABLE Vehicles (ID INTEGER PRIMARY KEY, revision TEXT, isCalibrated BOOLEAN, seriesnumber TEXT)
ID	INTEGER	"ID" INTEGER
revision	TEXT	"revision" TEXT
isCalibrated	BOOLEAN	"isCalibrated" BOOLEAN
seriesnumber	TEXT	"seriesnumber" TEXT
WIFI		CREATE TABLE WIFI (vehicleID INTEGER PRIMARY KEY, network TEXT, MQTTServer TEXT, SSID TEXT, password TEXT, FOREIGN KEY (vehicleID) REFERENCES Software(vehicleID))
vehicleID	INTEGER	"vehicleID" INTEGER
network	TEXT	"network" TEXT
MQTTServer	TEXT	"MQTTServer" TEXT
SSID	TEXT	"SSID" TEXT
password	TEXT	"password" TEXT
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
name		"name"
seq		"seq"

Table G.1: SQLite Tables of `DB_DT` Database

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature