

MASTER

MQTT based Communication Framework for AGVs in a Digital Twin

Thijssen, E.A.

Award date:
2021

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



MQTT based Communication Framework for AGVs in a Digital Twin

*Master Thesis - Manufacturing Systems
Engineering*

Student:

Ennio Anthonius Thijssen

TU/e student number:

0810786

Supervisors TU/e:

dr. ir. M.J.G. van de Molengraft

prof. dr. ir. I. Adan

dr. Q.V. Dang

ir. N. Singh

Version 1.0

Eindhoven, The Netherlands, December 2020

Abstract

The Brainport Industry Campus (BIC) in Eindhoven is an operation bringing together high-tech manufacturers, knowledge institutes and the government to share knowledge in order to compete in the manufacturing business. Its vision sets an innovative interpretation of common goods. With the principles of the fourth industrial revolution (smart manufacturing 4.0) in mind, data generation and resource sharing are the keypoints.

In this thesis a proposal is made for a communication framework (CF) capable of serving multiple tenants sharing a fleet of heterogeneous automated guided vehicles (AGVs) in a Digital Twin.

For further alignment with smart manufacturing the Digital Twin uses bi-directional communication to enable real-time data generation used to feed the algorithm. The CF also includes a user interface (UI) as order input system, an order database and a centralized Scheduler Module that houses the algorithm. The Scheduler Module encompasses the main intelligence by communicating with all aforementioned components.

This CF which serves multiple tenants sharing an AGV fleet is the first of its kind (to the best of my knowledge). A digital test bed is delivered for cost-reduction in the development of a CF in the real life manufacturing environment of the BIC. The CF serves as a template for analyzing and optimizing scheduling algorithms developed for the BIC case.

Acknowledgements

Writing this master thesis has been a huge challenge as the start aligned with the first COVID-19 lockdown. I would like to thank a few people who have helped and supported me throughout this period. First, I would like to thank my day-to-day supervisors Vinh Dang and Nitish Singh that trusted me with the opportunity to be a part of the innovative project together with the BIC. They both supported and critically reviewed my weekly updates but most of all motivated me if it were face to face or in online meetings. I couldn't have done it without them. I also want to thank Prof. Dr. Ir. I.J.B.F. Adan who always wanted be kept in the loop to check whether my thesis was moving in the right direction.

I would also like to thank Jack Put who guided me through the first couple of months by introducing the BIC.

Next, a special thanks to Pieter Wetering and the rest of the team of Prespective. Although the software is still under development, Pieter and his team always fit in a meeting in their busy schedule to provide me with the highly needed information.

Last but not least, I would like to thank my friends and family who unconditionally supported and motivated me. The weekly meals provided by my mother kept me focused on this thesis.

Nomenclature

<i>CF</i>	Communication Framework
<i>FMS</i>	Fleet Management System
<i>QoS</i>	Quality of Service level for MQTT
<i>AGV</i>	Automated Guided Vehicle
<i>CPPS</i>	Cyber Physical Production Systems
<i>DES</i>	Discrete Event Simulator
<i>UI</i>	User Interface

Contents

Contents	iii
1 Introduction	2
2 Literature Review	5
3 Problem Description	7
3.1 Scope of Project	7
3.2 Assumptions	9
3.3 Research questions	10
4 Functional Design	11
4.1 Purpose and Scope	11
4.2 Functional Requirements	11
4.3 Non-Functional Requirements	12
4.4 Performance Requirements	14
4.5 System Requirements	14
4.5.1 System Documentation	14
4.5.2 System integration	15
5 Software Architecture	16
5.1 Structure of Communication Framework	16
5.1.1 Scheduler Module: Flask Application	17
5.1.2 User Interface: HTML Page Rendering	18
5.1.3 Request Database: PostgreSQL	18
5.1.4 Simulation: Prespective Digital Twin Software	19
5.1.5 MQTT Broker	22
5.2 Behaviour of Communication Framework	23

5.2.1	UML Use Case Diagrams	24
5.2.2	Use Case 1: Submit, Edit and Delete Request	24
5.2.3	Use Case 1 in Detail	24
5.2.4	Use Case 2: Complete Request	30
5.2.5	Use Case 2 in Detail	30
5.2.6	Use Case 3: Find a Request	32
5.2.7	Use Case 3 in Detail	32
6	Validation and Discussion	35
6.1	MQTT Communication Validation	35
6.1.1	Test set up	35
6.1.2	Results	38
6.2	Functional Design Validation including Software Assessment	40
6.3	Discussion	43
7	Limitations and Conclusion	45
7.1	Limitations	45
7.2	Conclusion	46
Appendices		48
A	Additional UML diagrams to Use Case 1 (Submit, edit and delete request) . . .	48
A.1	Use Case 1 in detail	48
B	ISO 25010	54
C	Input/Output Data Model	55
C.1	Scheduler Algorithm	55
C.2	SQL Database	58
C.3	User Interface: HTML Pages	61
C.4	MQTT Broker	62
C.5	Digital Twin (Prespective)	62
C.6	Scheduler Module (Flask application)	64
D	Prespective Scripts	67
D.1	Standard Prespective Scripts	67
D.1.1	DES Controller and Pre Logic Simulator	67
D.1.2	DESCue and DSpline	69

CONTENTS

D.2	Adapted Prespective Scripts	69
D.2.1	AGVManagerInstructor.cs	70
D.2.2	AGVActor.cs	70
D.2.3	mqttStruct.cs	71
D.2.4	getDistanceMatrix.cs	72
E	Scheduler Module’s Scripts	73
E.1	SchedulerModule.py	73
E.2	SQLEditor.py	73
E.3	Algorithm.py	73
E.4	Models.py	74
E.5	Initialization Files	74
F	Installation Guide	76
F.1	Introduction	76
F.2	Purpose of Document	76
F.3	Prespective Digital Twin Software	76
F.4	Visual Studio Code	77
F.5	PostgreSQL	78
F.6	Mosquitto MQTT Broker	79
G	User Manual	80
G.1	Monitoring Processes	80
G.1.1	MQTT.fx	80
G.1.2	Database insight	80
G.2	Getting Started	81
G.3	Initialize CF with own request dataset	85
G.4	Implement custom algorithm	85
G.5	Generate custom factory Layout	86
G.6	AGV Adjustments	87
H	Scientific Code of Conduct	89
	Bibliography	91

1 Introduction

In late 2019, the Brainport Industries Campus (BIC) opened its doors to a variety of companies and institutions. This new 'Factory of the Future' connects High-Tech suppliers with knowledge institutions. Sharing and combining data produced by every individual company can help them improve their performances, enhance innovation and compete with rival companies.

In the eyes of tenants in the BIC, the 'Factory of the Future' should share data and its resources for further optimization of throughput and cost reduction. Automated Guided Vehicles (AGV), storage locations and other shareable rooms/machines are representing these shared resources. This vision could be the next step for manufacturing innovation. Besides the clear efficiency and cost motives, reducing the ecological footprint of factory environments is an attractive factor for governments to get engaged.

The vision of the 'Factory of the Future' aligns with the fourth industrial revolution. Data exchange in physical production systems for optimizing processes are the key points nowadays in smart manufacturing. These systems are called Cyber Physical Production Systems (CPPS) by equipping machines with data generating electronic parts and communication technologies [1]. Besides machines, AGVs can also be equipped to share data.

The data exchange of shop floor machines and resources is the latest trend in cutting-edge smart factories in the high-tech industry. Machinery and resources (AGVs for instance) are linked and read out, producing data in order to better comprehend factory processes. Doing so, decreases throughput times and reduces costs which are the main objectives for almost every factory. The use of AGVs for transport purposes support these goals as they can be controlled by scheduling algorithms. These algorithms become more accurate as they are feeded with real-time data. Feeding more data can support in the better comprehension of factory processes.

In this research the aim is to design a modular communication framework (CF) that can process incoming order requests from multiple tenants. This CF will control the AGVs in real-

time by assigning transport requests to AGVs. Through the concept of virtual commissioning [2], a Digital Twin will serve as substitute for the real-time AGVs. Besides the Digital Twin the CF includes a user interface as order input system, an order database and a centralized Scheduler Module that houses the algorithm. The communication between the Digital Twin and the rest of the CF is governed by a MQTT broker. A detailed look into these components and the structure of the CF can be found in Chapter 5. For more information about its behaviour see Chapter 5.2.

The biggest challenge in developing such a CF is to bring together several pieces of software and establish an inter-modular communication. A scheduler algorithm, database, user-interface and simulation software have to be combined where each entity could have a different programming language and protocol for communication. Each module needs an interface for communication towards other modules.

The main contribution of the proposed CF is its modularity. By keeping it modular, the CF serves as a template for similar cases. Modules can be replaced or adapted to fit the user's needs. Every module is locally hosted, which means the complete virtual testing environment can run on a single computer.

The scope of the project is defined by a test case at the BIC. Multiple decision support tools and models are developed which are awaiting to be tested. The current developed tools are:

- Single-load AGVs Scheduling
- Multi-load AGVs Scheduling
- AGV Dispatching Path Planning

With the help of these tools the aim is to investigate the maximum automation allowance in the BIC. The development of decision support tools and models regarding the multi-tenant AGV fleet management system (FMS) aims to maximize the usage of AGVs.

The logistics at the BIC can be split up into warehouse logistics and manufacturing logistics. The current research however will concentrate on the warehouse logistics. This includes all logistics before reaching the tenant's warehouses: all transport coming from the central

warehouse (BIC) to the tenant's warehouses and transport from incoming external orders to the central warehouse. The proposed CF in this thesis brings together all tools. It forms a test environment for these tools before employment in the real world.

The CF will result into three deliverables:

1. A report defining the software's structure and behaviour architecture
2. Software package for setting up the CF.
3. Software documentation including Input/Output(I/O) data model, installation guide and user manual.

In this report, the current literature on this topic is set out in Chapter 2. Chapter 3 defines the problem and scope in depth of this project. In Chapter 4 a document is set up that houses all requirements (functional, non-functional, technical and performance) concerning the software which is to be developed. It serves as a mutual understanding between the client and developer. In Chapter 5 the architecture blueprints of the software are laid. The first part describes the structure whereas the second part describes the behaviour of the system. In Chapter 6 a validation is performed to assess the proposed CF's accuracy. Chapter 7 states the discussion and conclusion including limitations.

2 Literature Review

Searching for literature about communication frameworks for a heterogeneous AGV fleet serving multiple tenants, is bounded by a precise definition of the system to be designed. Erol et al. [3] propose a multi agent-based system (MAS) for AGVs and machines within a manufacturing environment. The system is built for a single tenant's AGV fleet but its structure is interesting. They define MAS as a decentralized control systems where each subsystem is controlled by its own controller based on local information and actions. In their system every agent contains some form of intelligence and acts autonomously [3], [4]. The term multi-agent system occurs more often in the search for communication frameworks. This paper proposes a modular structure for a CF. Mayer et al. [2] present a framework consisting of a simulation model, an MAS including centralized or decentralized control logic and a data-exchange interface. They define MAS to be a 'loosely coupled network of problem-solving entities that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity' (Mayer et al., 2019, as cited in Stone et al., 2000). This definition sounds paradoxal with the term centralized in front of it. One should be careful when using or reading the term MAS. They excluded all forms of intelligence from the simulation model. All decision making occurs in the MAS whereas the simulation is used for sending triggers when specific events happen (trigger events).

Lohse et al. (2020) discuss the increase of data generation in production environments nowadays [1]. They propose a Real Time Reaction (RTR) concept that is constructed for a CPPS. In RTR, decision-making makes use of live production states. Additionally they build an RTR-relevant software that includes a plant visualization to visualize a manufacturing environment (e.g., machines and AGVs), a fleet management system (FMS) for controlling the AGVs, an Internet of Thing platform used as middleware for data generated by IoT devices and a manufacturing execution system (MES) to provide the production order, routing and process parameters. This paper introduces components needed for a modular CF which can be adapted to fit different production environments. This template like structure is interesting for the BIC case as multiple algorithms need to be tested.

Like mentioned in the Introduction, a Digital Twin will be used as visual representation for the real life AGVs. A Digital Twin is similar to a simulation model but has one significant difference: a Digital Twin integrates real time data. The communication framework should be able to handle this real time data from the Digital Twin. A suitable communication protocol for IoT devices is to be chosen. The most important factors to be considered are scalability, power consumption and reliability. Naik [5] provides an assessment on IoT messaging protocols MQTT, CoAP, AMQP and HTTP. Among them MQTT and CoAP are most suitable as both are designed to work on low bandwidth and resource requirement. CoAP is slightly favoured when it comes to power consumption [5], [6]. Khaled et al. [7] mentioned that MQTT is scalable with the help of a program called EMQTT, a fully open source and massively scalable MQTT technology. In contrast Kovatsch et al. (2014) propose scalable IoT cloud services based on CoAP. Both are able to scale up to big numbers. However, in terms of reliability, MQTT is superior to CoAP. The reliability is expressed according to the Quality of Service level (QoS) of MQTT. It has three stages 0, 1 and 2 that means a message delivered at most once, at least once and exactly once, respectively. CoAP has no explicit QoS levels but has protocols on reliability which are similar to QoS of 0 and 1. Only the QoS level of 2 is suitable for this thesis because each message sent should have a guaranteed delivery. This means CoAP is not suitable and MQTT is suitable.

3 Problem Description

In the factory of the future, multiple tenants housed together share the same transport resources. An AGV-fleet to serve multiple suppliers can increase productivity and decrease costs heavily. Because the BIC in Eindhoven is not ready for a full-on working AGV system as the AGVs are not present yet, a digital twin is created to experiment. This project focuses on the implementation of the communication framework using a Digital Twin. In this chapter the scope of the project is set out and its limitations are defined.

3.1 Scope of Project

In this research a field lab in the BIC is used which is depicted in Figure 1. In total 21 pick up and delivery nodes (machines or warehouse), 2 charging stations and 5 AGVs are implemented in the digital twin. The paths consists of a main path going round the whole production floor and branches going to specific machinery or warehouse locations. The field lab is built based on a single tenant. For the sake of this research each individual node can be seen as a single tenant which defines the field lab as a multi-tenant environment.

The separate components of the CF consists of tenants (clients), a database, scheduler, scheduling algorithm and a Digital Twin which represents the fleet management system (FMS) and the AGVs. In order to establish a working system, a communication framework between the different components should be established. In this framework the information flows and possible events between or by modules have to be indicated. In Figure 2 the global communication lines between the different components of the system are shown. Note that the system is centralized as the scheduler module obtains all necessary information to schedule a request.

The Digital Twin has to send information about resources (AGV availability, battery capacity and location) to the scheduler when asked. In turn it receives a scheduled/processed request from the scheduler. Communication is bi-directional. The database is another partial supplier of data needed for the scheduling algorithm (housed in the scheduler module). It will send the to be scheduled orders when asked by the scheduler. The scheduler will send the processed

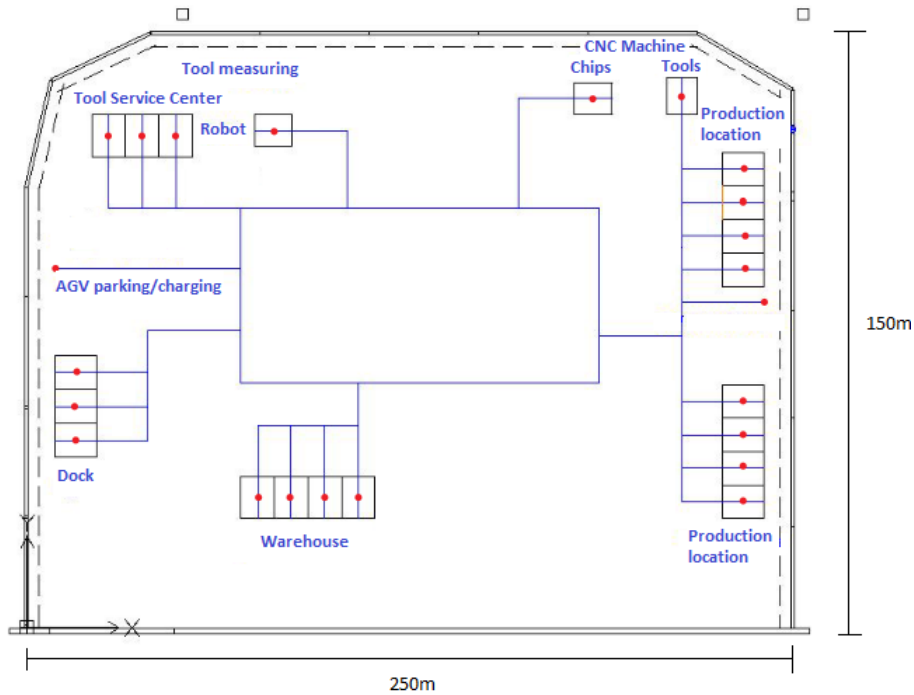


Figure 1: BIC's warehouse layout

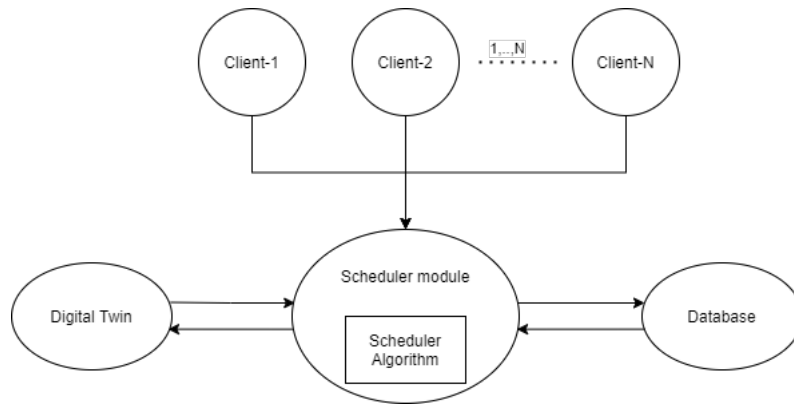


Figure 2: Global communication flows between components

request to the database for updating the request's status and to the Digital Twin for execution of the request. Users of the system can extract request information from the database if needed.

In order to ensure ease of further use, modularity is key. Meaning that if future users want to use the framework but with a custom made module(s), this should be implementable. A future user might want to implement the CF in a multi-tenant environment but with its own scheduler algorithm. The same counts for the layout of the production environment. As

Mayer et al. [2] mentioned, the simulation model (Digital Twin) has to be modular and scalable to support custom layouts or/and changes in resources (AGVs) and simulation nodes. The challenge lays in the communication between the modules because each module could be using different communication protocols and data languages. In order to make all modules communicate with each other converters should be implemented. As more modules are implemented, diverse syntax should be handled by these converters. Ensuring modularity can be made possible by generating a detailed input-output model (I/O model) for interchangeable modules that defines exactly how data is handled within a module.

In this research the aim is to design a modular communication framework to analyze and optimize algorithms applicable to multi-tenant environments. Note that the focus is on the communication between modules.

3.2 Assumptions

A few aspects will be left out of scope for now. These will generate assumptions stated in the text below.

1. Human intervention
2. Optimization of Scheduling Algorithm
3. Multi unit load at a time
4. Security mechanism

The first point basically implies no workers are present in the AGV's routes. The workers will only be present at starting/ending points. The second aspect means that the Scheduling Algorithm only has to be adapted to its dynamic environment and implemented. The optimization of this algorithm is left out of scope. The third point is for later work. A student at the Technical University of Eindhoven is currently doing research in optimal route finding and task planning for multi unit load at a time. Combining these projects is left for further research. The protection of data and authorization for the scheduler is not handled. The General Data Protection Regulation (GDPR) is followed to only handle fictional data in this project.

3.3 Research questions

Concluding the whole problem description, a main research question is set up. It is formulated as follow:

How to develop a communication framework to implement a scheduling algorithm for a heterogeneous AGV fleet in a multi-tenant environment?

To answer this main research question, sub questions are set up:

1. *What is needed for designing software?*
2. *How can all components communicate on the edges using different syntax?*
3. *How to assess the quality of this software?*

All of these sub questions are to be answered throughout this thesis.

4 Functional Design

The first step when developing software is setting up the functional design document. In this document the functional, non-functional, technical and performance requirements combined with its appearance are written down and agreed upon by both developers and clients. It serves as a contract between developer and the client on demarcating the scope and functionalities which should be implemented.

4.1 Purpose and Scope

This document serves as a mutual understanding between the client (BIC) and the developer (myself) that defines on a high-level the software's functional, technical and performance requirements. In this project, the functional design document also serves as a development guide (for the developer) to make sure all needed functionalities are incorporated.

The software which is to be developed will be delivered as a package consisting of multiple pieces of software. Its goal is to help future users to analyze and optimize similar real life cases. Its modularity should enable these future users to adapt and scale the CF to fit their needs. The stakeholders and owners of this project are the 21 BIC tenants and the future tenants that will house in the BIC.

4.2 Functional Requirements

In this subsection the functional requirements will define what the CF should be capable of doing. These functional requirements are based on the certified ISO 25010 norm for software quality assessment. This method will be explained in detail in Chapter 6.2. The important thing to know for this chapter is that the ISO 25010 norm assesses software by means of characteristics. In Figure 3 the traditional characteristics of this norm are depicted on the left. In order to fit assessing the CF it is adapted (see Figure 3 at the right).

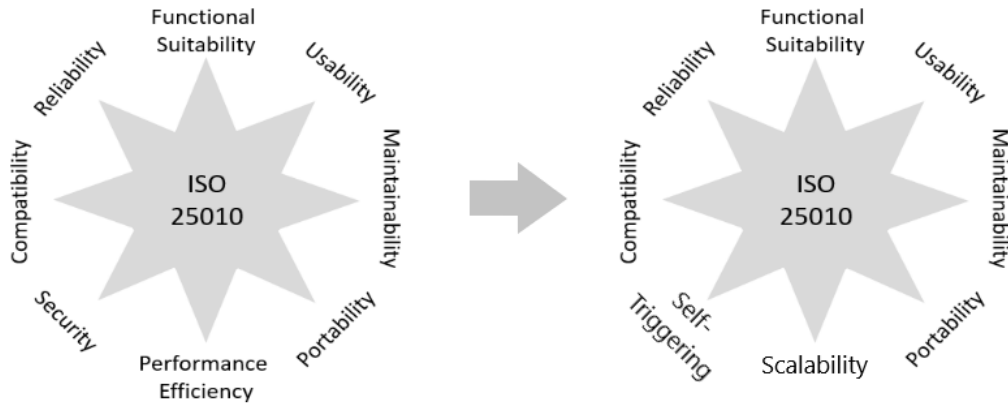


Figure 3: ISO 25010 main characteristics adjusted to fit this research [8].

It should contain the following primary functionalities which are split up into concrete functionalities:

- **Ability to initiate Use Cases (Functional Suitability)**
 - Users should be able to submit, edit and delete requests. Users should also be able to find a request in the database to check its status and other properties.
 - Editing and deleting of requests should only be possible for unscheduled requests.
- **Self-triggering**
 - All trigger events should force the algorithm to reschedule.
 - Trigger events should be definable inside or outside of the production environment (Digital Twin). Outside means not from the Digital Twin (for instance when a request is submitted).
- **User Error Protection (Usability)**
 - Only the input for the use cases is user dependent. Submitting data in the wrong format should invoke an error.
 - User input should be returned upon submitting data for one of the use cases

4.3 Non-Functional Requirements

In this section only requirements are mentioned that are non-functional. These are functionalities that define how the system should be built up. These requirements are more focused

on the software's structure. As the name suggests, non-functional requirements are the counterpart of functional requirements that define the software's behaviour. All non-functional requirements are set out beneath. All techniques to achieve these non-functional requirements are stated.

- **Interoperability (Compatibility):** the system should be able to handle input from other systems and produces output that can be used by other systems.
 - A detailed specification document (I/O data model) should define how certain input and output information should be formatted to be used by other systems/modules.
- **Recoverability (part of Reliability):** the system should be able to find already processed orders.
 - Setting up an external database which is independent of the rest of the system acknowledges that requests will not be lost in the occurrence of an error.
- **Maintainability:** the system should be adaptable by administrators.
 - Error debugging and traceability of failures should be possible. Providing the necessary print and try and except statements in the code helps in tracing errors. For errors regarding communication between the Digital Twin and the rest of the system, some sort of message tracking has to be recommended.
 - Design documentation should be available.
 - Enough comments to explain the scripts/functions. This is intuitive as there is no rule of thumb on the ratio lines of code/comment
- **Adaptability (Portability):** the measure of how much the software's modules can be adapted to other software/hardware and environments.
 - The software's documentation should be sufficient enough that new users can apply the software to their own system. The input and output interfaces should be well defined. The aforementioned I/O data model, a user manual and installation guide satisfies this requirement.

- **Quality assessment**

- One of the most used software quality assessment is the ISO 25010 norm. The aspects assessed can be found in Table B1. This norm is a metric for quality characteristics. In the table all green fields are applicable to this project. The aspects considered can be found with its approach of how to achieve it. More information can be found in Chapter 6.

4.4 Performance Requirements

In this section the required performance parameter will be stated. As the software is a CF which focuses on modularity, its performance will not be assessed on basis of the scheduler algorithm. A simple algorithm is implemented only to verify if the CF is working properly. Instead the parameters used to assess its performance are:

- **Message reliability (part of Reliability)** - By using a Digital Twin data is constantly exchanged with the CF. Loss of data could have a big impact on the performance of the CF. A suitable communication protocol has to be selected.
- **Scalability** - For future purposes the CF should be able to massively scale up. Users may want to extend data generation out of Digital Twin or use bigger layouts. In order to enforce scalability a lot of parameters could be of influence. This thought has to be kept in mind with all decisions that could affect scalability.

4.5 System Requirements

4.5.1 System Documentation

The software package which will be developed is in need of a strong system documentation. The package has to form a template to use for analyzation and optimization of similar like cases. As parts of the system may be replaced by custom made parts of the user, a detailed system documentation is key for further use. The main parts of the documentation are: an I/O data model, user manual and installation guide. Despite the CF's modularity, scaling up methods should be well defined. Scaling up could mean multiple modules need adjusting. Parts considered for scaling up should be the Digital Twin's layout and its data generation. Another

argument to stress the importance of a strong documentation is that it may be implemented on a real life production environment. The engineer in control of the implementation needs to adjust the software to his/her liking. The same applies for maintenance engineers.

4.5.2 System integration

As mentioned before, the final CF should be usable by in combination with custom modules. The accessible data and the system's functions should be clearly set out to facilitate ease of use. This effectively means, a data input/output document which specifies how to get information out of the every module and how the data input should be formatted. System integration is the coupling of all physical and/or virtual parts.

Additional documents complementing the system integration are an installation guide and a user manual. These documents are desirable for both administrators and future users of the CF.

5 Software Architecture

In the world of software development, UML has grown to be a de facto standard for design and development of object-oriented systems [9]. This modelling tool is used to form a bridge between IT and business departments. Both behaviour and the structure of systems can be designed using UML. Simply starting with programming a script can cause problems later on. If the UML diagrams are designed correctly, it could ease the development process. For programmers it forms a blueprint for the software. Because of its highly visual notation it is also understandable for non-software engineers (like business departments etc.).

In the design phase first the behavioural UML diagrams are designed, followed by the structure UML diagrams. If the behaviour is modelled accurately, it should encompass all the system's components. The other way around could deliver more problems. If the structure of the system (all applications and physical entities) are modelled first and its behaviour after, forgotten components could come to light that were not thought of before. This means the structure should be adapted, which can lead to major programming issues.

In Subchapter 5.1 the structure will be handled first. The structure of the framework is first set out using a structural diagram explaining all components. Additional diagrams are set up to go more into detail. After the structure is clear the behaviour is set out in Subchapter 5.2 using UML diagrams. Starting UML Use Case diagrams which are behavioural diagrams based on real life applications. These are worked out in detail using UML sequence, activity and timing diagrams. The subchapters will elaborate on a specific functionality using these diagrams.

5.1 Structure of Communication Framework

In Figure 4 the proposed communication framework can be seen with all modules defined. The Scheduler module is represented by a Flask web application. The database used is PostgreSQL. The client side (User Interface) is generated by a set of HTML pages. The Digital Twin is designed in Prespective, a Digital Twin software for Unity. To enable bi-directional communication between the Scheduler module and the Digital Twin, an MQTT broker is im-

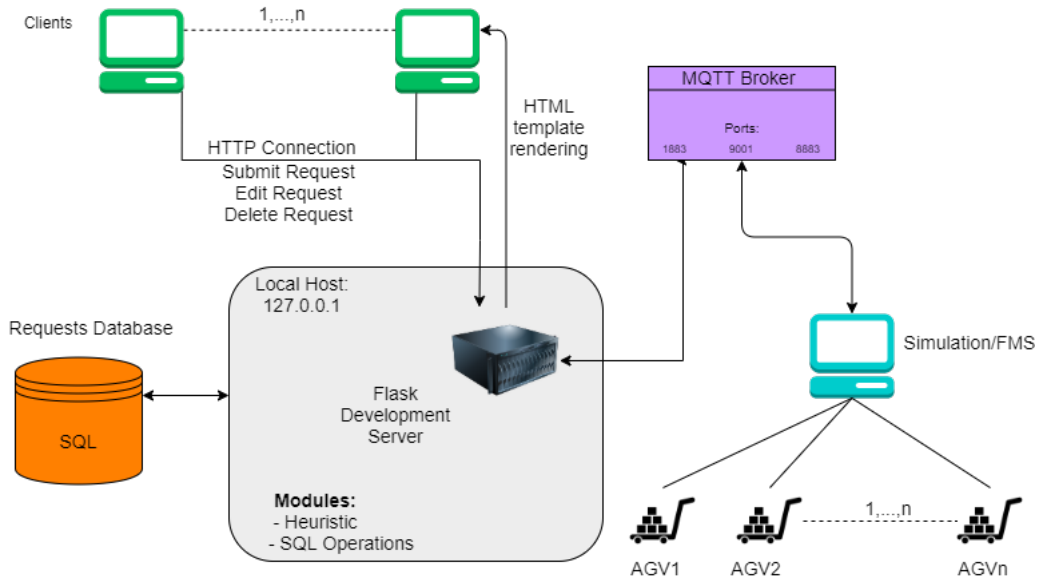


Figure 4: Global structure of communication framework

plemented as communication protocol. Each component is handled separately and explained how they are designed.

5.1.1 Scheduler Module: Flask Application

One of the most important components of the CF is the Scheduler Module. All communication towards the remaining modules is initiated or passed through this module. The whole framework can be seen as a centralized communication framework. Almost all intelligence is housed in this component in the form of an algorithm. In order to let all modules communicate with this centralized component, it is set up as a web-based application called Flask running on Python.

Flask is a microframework for web-based applications. It holds the term microframework because of the absence of required tools and libraries in order to run. Although Flask is a microframework, it is provided with a number of extensions in order to vastly extend its functionalities. Flask sets up a server where you can define specific routes (URL scheme) to which the user can navigate to using HTML pages. The HTML page rendering and data abstraction is done via an HTTP connection between clients and the Flask server. For the data abstraction of these HTML pages there exists an extension in Flask.

The user input abstracted from the HTML pages is communicated to the PostgreSQL database using a library called *sqlalchemy*. This library enables Object-Relational Mapping (ORM). ORM is used for data conversion between two incompatible applications. This effectively means that operations done on the PostgreSQL-database can be initiated from Flask using Python queries. This saves time converting SQL queries into Python. The same holds for custom classes defined in Python. These custom classes can be mapped onto an SQL table. If a query is made for an SQL object or a request is submitted/edited to PostgreSQL, it doesn't need converting between the two applications (Flask and PostgreSQL database).

From Figure 4 it becomes clear that all communication towards and from the Digital Twin is channeled through the MQTT broker. More information about the MQTT broker is provided at 5.1.5.

5.1.2 User Interface: HTML Page Rendering

The clients are the end users of the system. HTML pages are rendered to keep it user friendly and easy to use the framework. Forms are implemented for the user to submit their request details. Using HTTP methods *POST* and *GET*, data can be shared between the client and Flask. For the graphic design the library bootstrap is used. This library is also available as an extension in Flask: *flask_bootstrap*.

All use cases are initiated from the User Interface (UI): submitting, editing, deleting and finding a request. These use cases can all be performed during runtime. This module is the only part of the system where user input is required. A screenshot of the home page of the User Interface can be seen in Figure 5.

5.1.3 Request Database: PostgreSQL

For the storage of requests and their current status, a PostgreSQL database (server) is set up which runs on the language SQL. *PostgreSQL* is used to manage the locally hosted database. By using ORM, SQL queries and object modification/abstraction can be initialized from Flask.

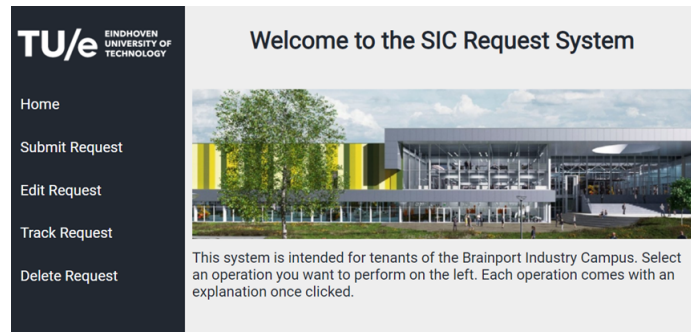


Figure 5: Visual representation of User Interface using rendered HTML-pages

In the PostgreSQL database it is possible to set an attribute of a row as primary key. This primary key is a unique identifier for each database entry (a request). The primary key is set to the request ID. Only requests are communicated between the PostgreSQL database and Flask. Each request consists of the following attributes: request ID, ordertype, pick up date, pick up time, end delivery date, end delivery time, source, destination, request cost, capacity, status, AGV assigned and end timestamp. For an explanation of each attribute see Appendix C.1.

5.1.4 Simulation: Prespective Digital Twin Software

Smart manufacturing is seen as the cornerstone of the fourth industrial revolution [10]. It defines the current trend in intelligent manufacturing by connecting machines and resources to the internet. By this connection, generated data from these machines and resources is used for optimization purposes. The AGV's status with respect to the battery charge and location are of importance to the scheduler algorithm. Therefore, bi-directional communication in the Digital Twin is essential. The main reason for the choice to use Prespective is its ability for bi-directional communication. Already mentioned in Chapter 1, smart factories generate a lot of data which are used for optimization techniques. Prespective can communicate with external modules during run-time by using Messaging Queuing Telemetry Transport (MQTT).

In Figure 6 the main structure of Prespective is visualized. All participants are mentioned: *Actors*, *Instructor* and *Cues*. All participants fall under a Discrete Event Simulator (DES) which contains the actual script for the behaviour of the simulation itself. The DES is also armed with a so-called Pre Logic Simulator. This simulator forms the bridge between Pre-

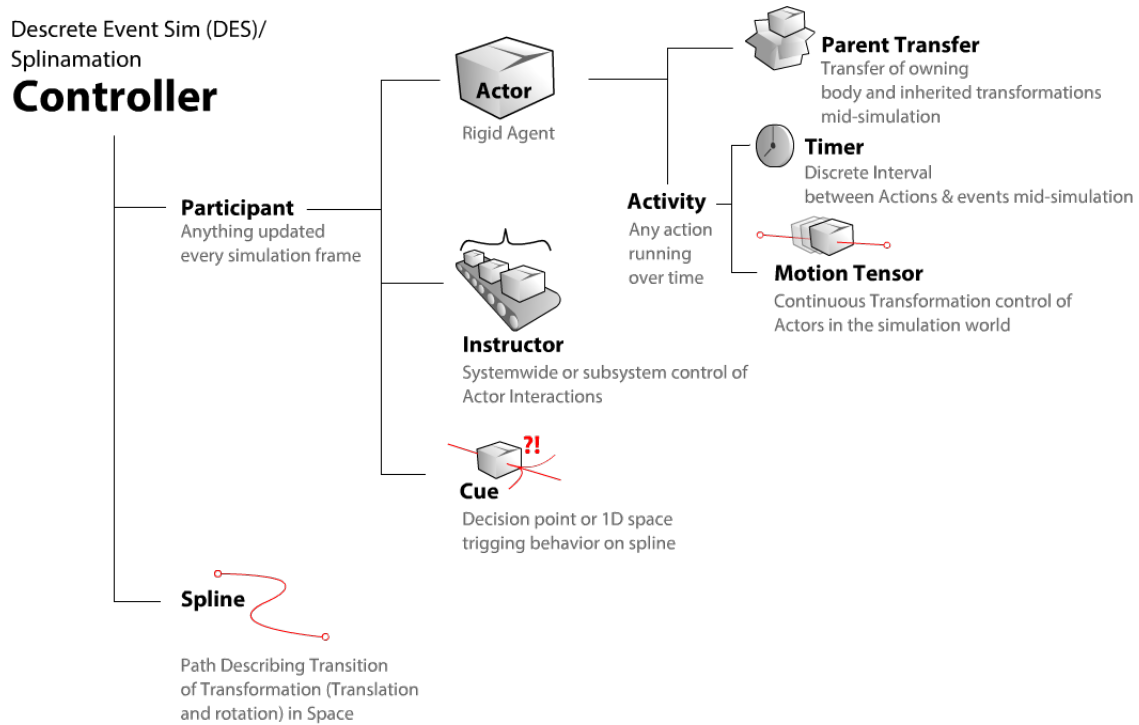


Figure 6: Perspective Digital Twin's structure

spective and the rest of the CF. The MQTT adapter is defined in this component.

This software is the replacement of a real AGV fleet which is controlled by a Fleet Management System (FMS). In Perspective a similar approach is implemented. They use *Actors* and an *Instructor*. *Actors* are the resources used, in this case AGVs. They must be provided with a script in which certain attributes can be defined which are updated with each simulation's frameupdate. Current updated attributes are *Charge* and *AGVPosition*. The *Instructor* is the control unit for *Actors* which means it controls the behaviour of each AGV on the system-level. The *Instructor* is similar to an FMS. In the *Instructor* script incoming requests are dissected and assigned to the concerning AGV. It sends the AGV on its transport route giving a sequence of steps to take in order to complete the request. The script is able to connect user input source and destination nodes to machine stations in the Digital Twin. The stations are declared in Perspective with the help of *Cues*. *Cues* are defined as junctions of paths which are decision points on a specific path. The remaining parts are the *Splines*. These are the user defined paths that describe the dynamics of *Actors* inside of the simulation. *Cues* are defined on this *Splines*. In order to reach a certain *Cue* the *Actor* knows how to reach its destination

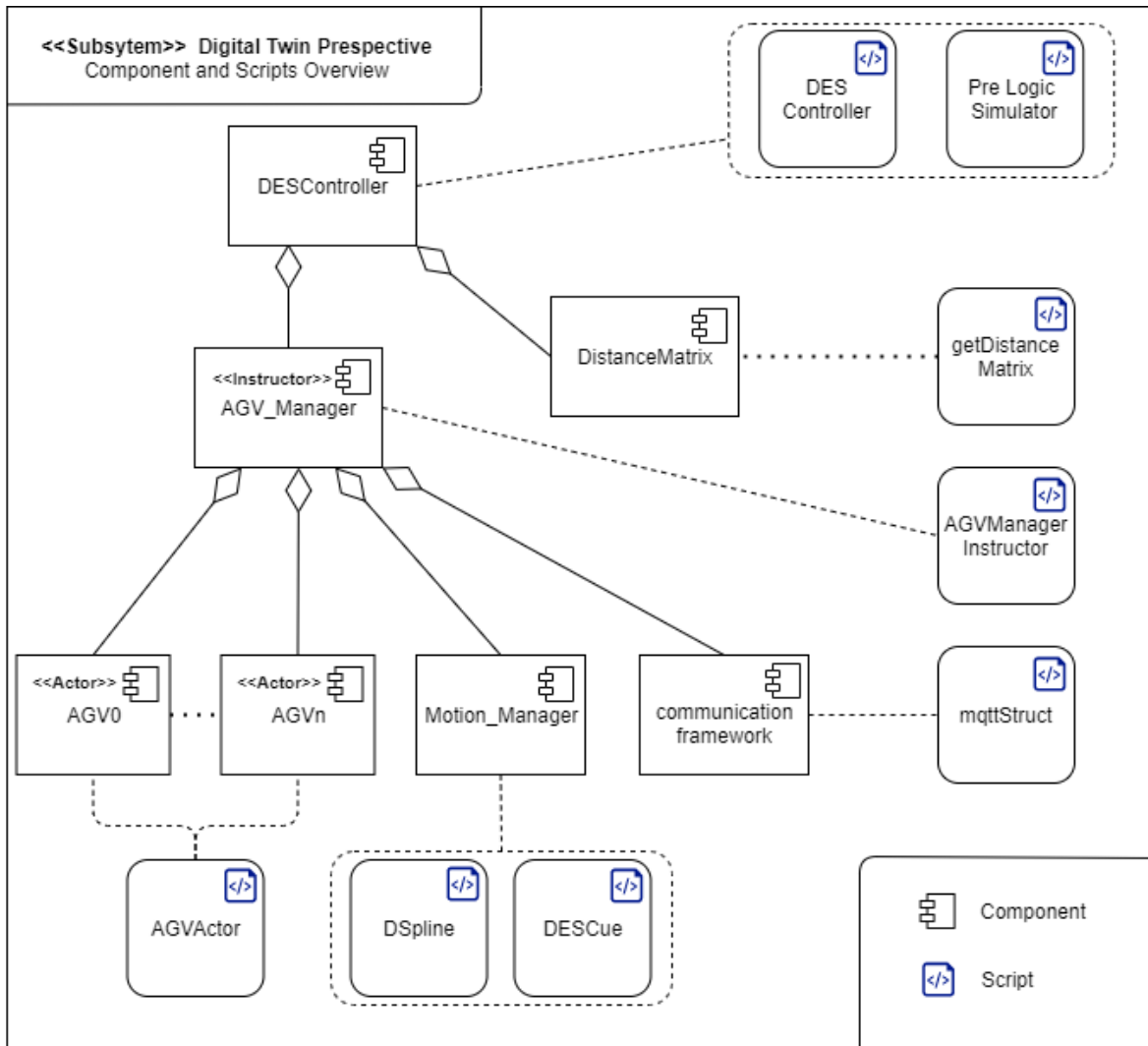


Figure 7: Prespective Component Tree with applied scripts

using these *Splines*.

A lot of new terms came to light in this chapter. It is clear that Prespective is mainly built on components and scripts. In Figure 7 the concerning scripts are coupled to the components. There are 2 types of lines depicted. The dashed line which shows which script is connected to which component. The lines with an empty diamond attached are defined as an aggregation relationship. It can be read as a 'part-of' relationship. For instance the component *AGV_Manager* is part of the component *DESController*. For more information about the used scripts in the Digital Twin software Prespective, see Appendix D.

5.1.5 MQTT Broker

An assessment was done on several IoT messaging protocols. MQTT is chosen for its reliability. It has the ability to choose a Quality of Service level (QoS) of two which guarantees a message is delivered exactly once which complies with the performance requirements. MQTT even has a function called "*last will and testament*" to ensure message delivery after disconnect. Naik (2017) did a comparison on CoAP, MQTT, AMQP and HTTP. MQTT wins the race on reliability. Together with its scaling abilities, it made the choice decisive to choose MQTT.

In this framework a locally hosted and open source broker called Mosquitto is used. A broker is a message mediator to which all participating modules (MQTT clients) can connect by using the broker URL (127.0.0.1 for locally hosted brokers) and a specific port. MQTT is a publish and subscribe messaging mechanism which utilizes a message mediator (broker). Upon connecting to the broker the client can subscribe to certain topics. If another client would publish (sending a message) on this topic through the broker, all subscribed clients will receive this message. Figure 8 gives more insight. E.g. the Digital Twin is subscribed to the topic `"/requests"`. If the Flask server publishes a message to this topic, the Digital Twin will receive the message as it is subscribed to that topic. Additionally, it is a lightweight communication protocol meaning its power consumption is low. Clients only make a connection to the broker once. If connected, a keep-alive signal is sent to check whether the client is still connected. If not, it is automatically reconnected. This feature supports the sustainability aspect of smart manufacturing [5].

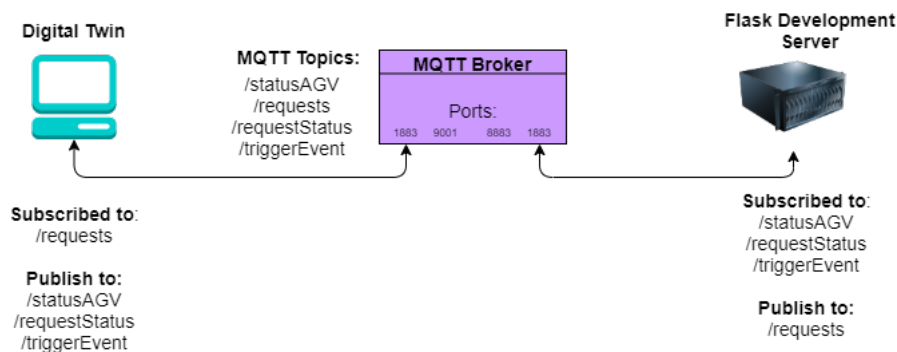


Figure 8: MQTT: a publish/subscribe messaging protocol

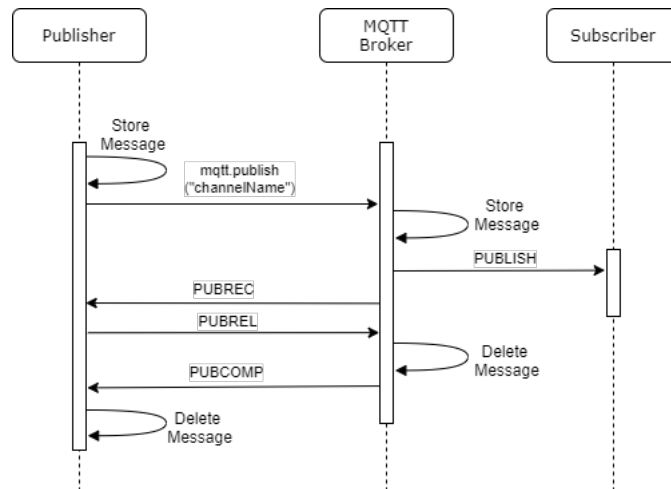


Figure 9: Four part handshake corresponding to QoS of 2

In order to better understand the behaviour of the CF, first the behaviour of an MQTT publish message is explained (see Figure 9). By selecting a QoS of 2 (every message is delivered exactly once) every publish message consists of a 'four part handshake'. This sequence occurs every time a message is published to the MQTT broker.

When the receiver obtains the initial *PUBLISH* message from the sender it acknowledges it with a *PUBREC* message. The original *PUBLISH* message is deleted by the sender and *PUBREC* is stored. The sender responds with a *PUBREL* message. If the receiver gets the *PUBREL* message, all stored messages are deleted and a *PUBCOMP* message is sent to acknowledge receiving *PUBREL*.

5.2 Behaviour of Communication Framework

In order to fully explain the CF's behaviour a distinction is made. Behaviour can be externally or internally triggered. With user initiated behaviour (externally triggered) an action is performed by a user which triggers specified behaviour. Internal triggers are trigger events that occur in the Digital Twin. In Chapters 5.2.1 until 5.2.5 both behaviours are depicted as UML Sequence diagrams. All behaviour is set out with the help of use cases.

5.2.1 UML Use Case Diagrams

Use case diagrams are used to define the system's functionality with respect to end users. These diagrams depict the overall functionalities. Alhumaidan (2012) stated that "The use case diagrams provide interactions between roles known as actors and system to achieve a certain goal" [9]. Per functionality a system boundary is denoted to define in which part of the system it takes place. Two types of relationships are used: the *include* and *extend* relation. *Include* implies a function that always follow after its origin function. *Extend* suggests a function which could be called but not necessarily.

From these use case diagrams test cases can be derived as they are based on the needs of the end user. Three use cases are depicted in diagrams to encompass its main functionalities: submitting, editing, deleting, completing and finding a request.

5.2.2 Use Case 1: Submit, Edit and Delete Request

The first three are combined into one use case diagram (Figure 10) since they use the same actors. These cases are straightforward. Submitting a request can result in an acceptance or a rejection which is dependent on the correctness of the user input. The user input undergoes several format checks before acceptance. E.g. a new request can't contain a end delivery date in the past. For *Edit Request* and *Delete Request* a *Verify Request ID* function is implemented. After this verification there are three subsequent function that could be executed: *Verification error*, *Accept Edit* or *Accept Deletion*.

5.2.3 Use Case 1 in Detail

The superficiality in the first use case (Figure 10) is obvious as more behaviour is hidden behind the functions mentioned. There exists a sequence of events for each of the three functions (Submit Request, Edit Request and Delete Request) which is hidden in the use case.

Submit Request

The first case which will be elaborated is Submit Request. A UML sequence diagram of

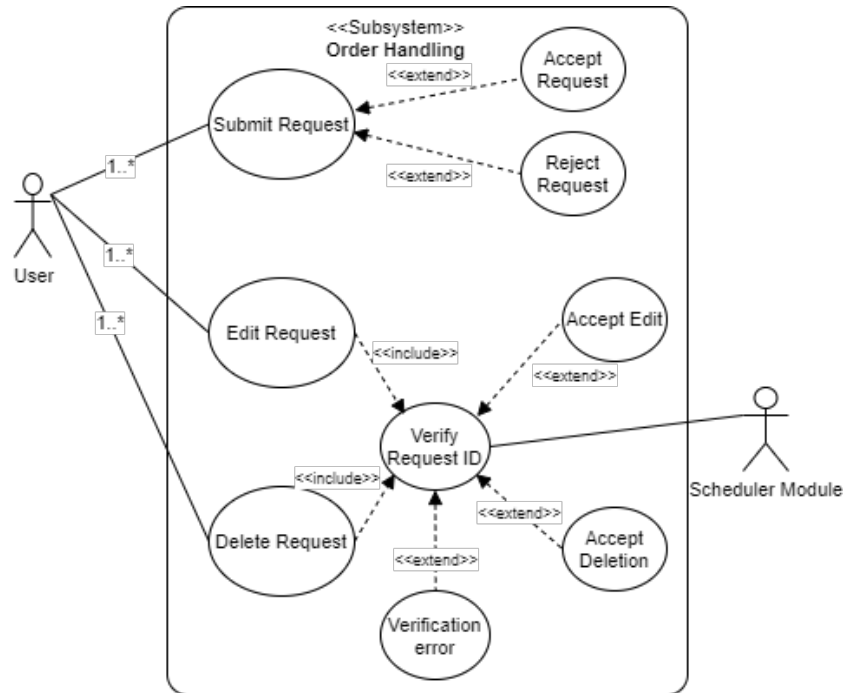


Figure 10: Submitting, editing and deletion of a request in a use case diagram.

its behaviour is depicted in Figure 11. After the initial function *requestSubmitted()*, which is triggered by the user via the UI, the Scheduler Module sends the new request to the PostgreSQL database with *db.session.add()* and subsequently *db.session.commit()*. At the same time the user is informed of the acceptance of his request using *render_template()* which contains the submitted request's details. The scheduler sends a message to MQTT-topic "communicationframework/externalTrigger" letting the Digital Twin know that an external trigger event occurred. This topic is solely used when a request is submitted, nonetheless it can be used for any external trigger event. The remainder of the sequence that follows is almost the same for every internal/external trigger event. From hereon the sequence can result in four ways. Note that only the first UML sequence diagram is located in this chapter because the main sequence of events of each result is comparable with minor changes.

Upon noticing that a trigger event occurred, the Digital Twin will send the AGV's statuses, which is always followed by a trigger message which triggers the Scheduler Algorithm. This use case can result in four ways:

1. Receiving this trigger message makes the Scheduler obtain the request with earliest End delivery date (*SQLEditor.database_sort()*) and send it to the Scheduler Algorithm (*Algorithm.main()*). The Algorithm assigns an AGV to the request and returns it to the Scheduler Module. The Scheduler Module publishes the scheduled request to the MQTT-broker which sends it to the Digital Twin. Do note that the scheduled request is also updated in the database using *SQLEditor.request_status_update()* (See Figure 11 for the concerning UML sequence diagram.)
2. For the second possible result the sequence stays the same as in 1 until *Algorithm.main()* is called. Now there is no idle AGV with sufficient charge to transport the selected request. The Algorithm asks for all unscheduled request with *SQLEditor.database_sort_all()*. It loops over all selected requests to check if one does not exceed the AGV's charge. If so, *Algorithm.main()* is triggered again with the selected request. After it assigned an AGV it gets published to the MQTT-broker which sends it to the Digital Twin. See Figure A2.
3. If there was no request (in the loop mentioned at point 2) to match the AGV's charge, the scheduler module will generate a charging request with *Algorithm.chargingRequest()*. This chargingrequest will be published to the MQTT-broker which will send it to the Digital Twin. See Figure A3.
4. The last case is a variation of point 3. When the *Algorithm.chargingRequest()* is initiated but the AGV's charge already has reached the upper threshold, this function will return *None* which ends the sequence. The sequence of this case can be found in Figure A4.

For better understanding the dynamics on submitting a request before the external trigger is sent over MQTT-topic "communicationframework/externalTrigger", an activity diagram is generated in Figure 12. In the lane User Interface all the possible routes regarding the concerning use case are shown. Beginning at the *index* page where all use cases are present. Its sequence can be explained as follow:

- At the *submitform()* route, the details of the request to be submitted can be entered. From this route it will go to *requestsubmitted()*.

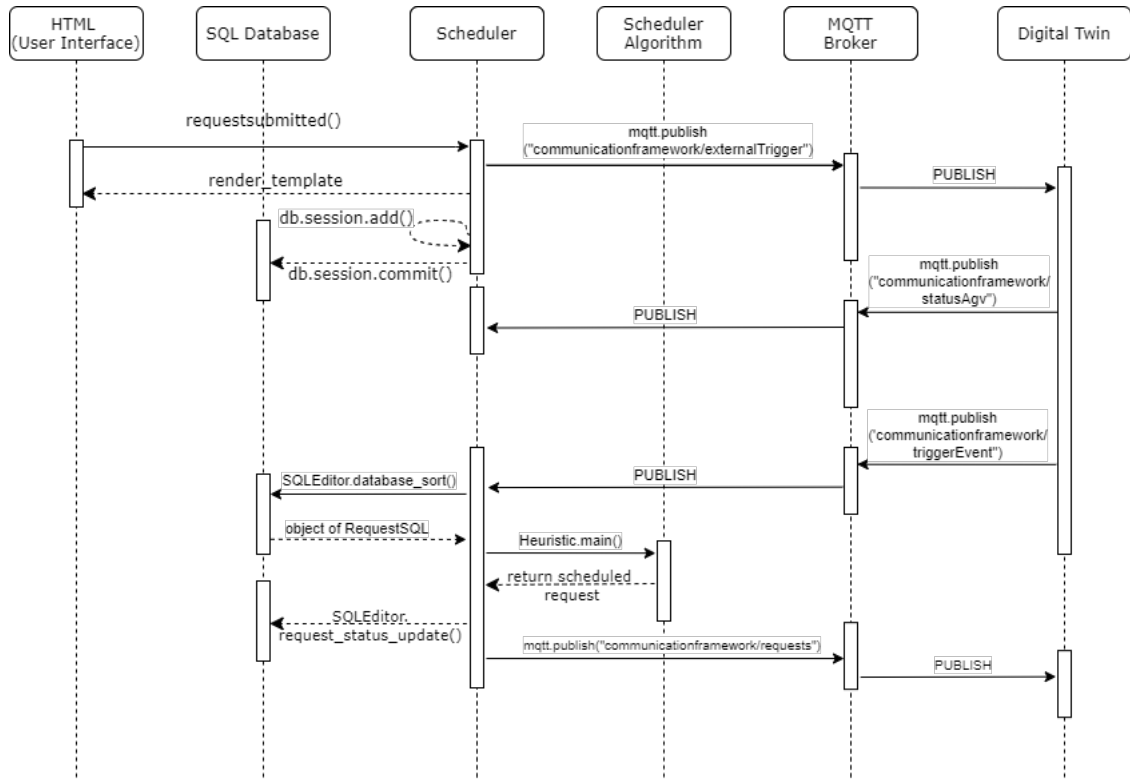


Figure 11: UML Sequence diagram in case of submitting a new request

- Before entering this route, input validations are executed. It is checked if all required fields are filled in and if all dates are eligible.
- If all input information is validated, an object of the *RequestSQL* class (see Table C2) is instantiated in order to submit the request to the PostgreSQL database using *SQLEditor.sql_submitRequest()*.
- This will be followed with *render_template('submitcompleted.html',newRequest* which returns the success HTML-page with the newly added request's details *newRequest*.

Note that the route *requestsubmitted()* is in a dashed rectangle together with all possible *render_template()* instances. In order reach the *requestsubmitted()* route, a *render_template()* must always be exposed. This notation will return in the coming UML Activity diagrams.

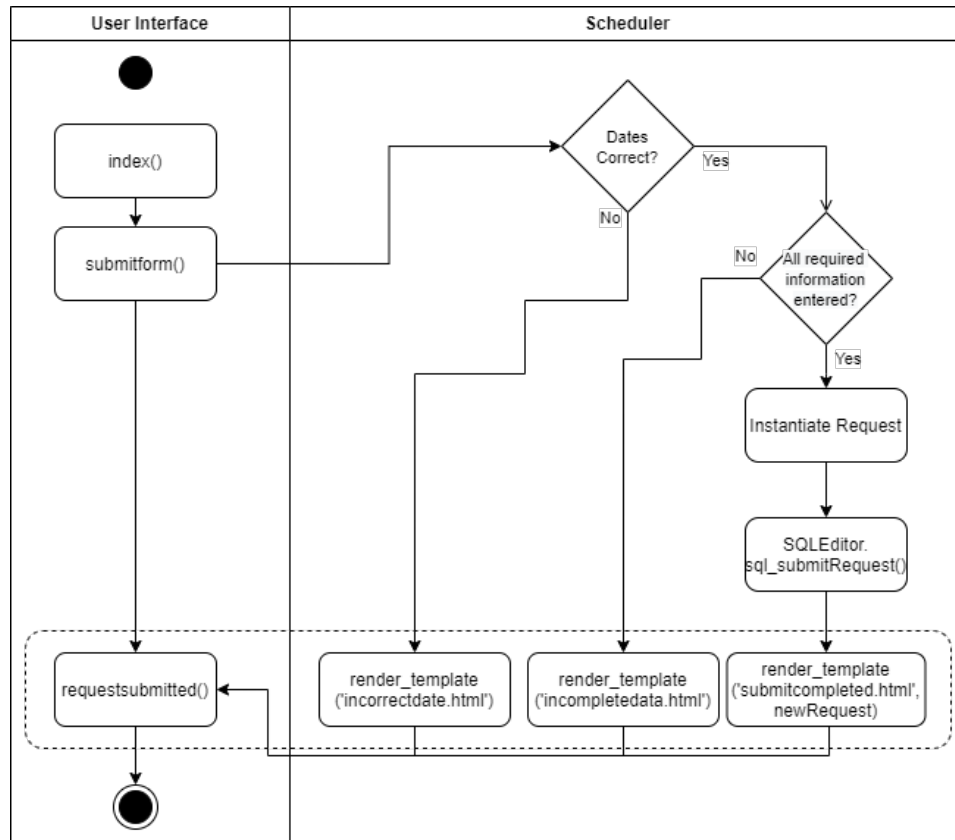


Figure 12: UML activity diagram for submitting a request

Edit Request

In the Edit Request function there exist a sequence of events as well. When a tenant wants to execute an edit, a request ID has to be submitted so the system knows which request needs to be edited. In Figure 13 a verification of the request ID is performed. It can either match a request or the ID is invalid which invokes resubmitting the correct request ID.

Before a template is rendered back to the user, it will perform the aforementioned validations. Validations are done on the request ID and if its status is still unscheduled. Dependent on this validations it renders the matching template. If all is correct the function *SQLEditor.sql_submitEdit()* is performed.

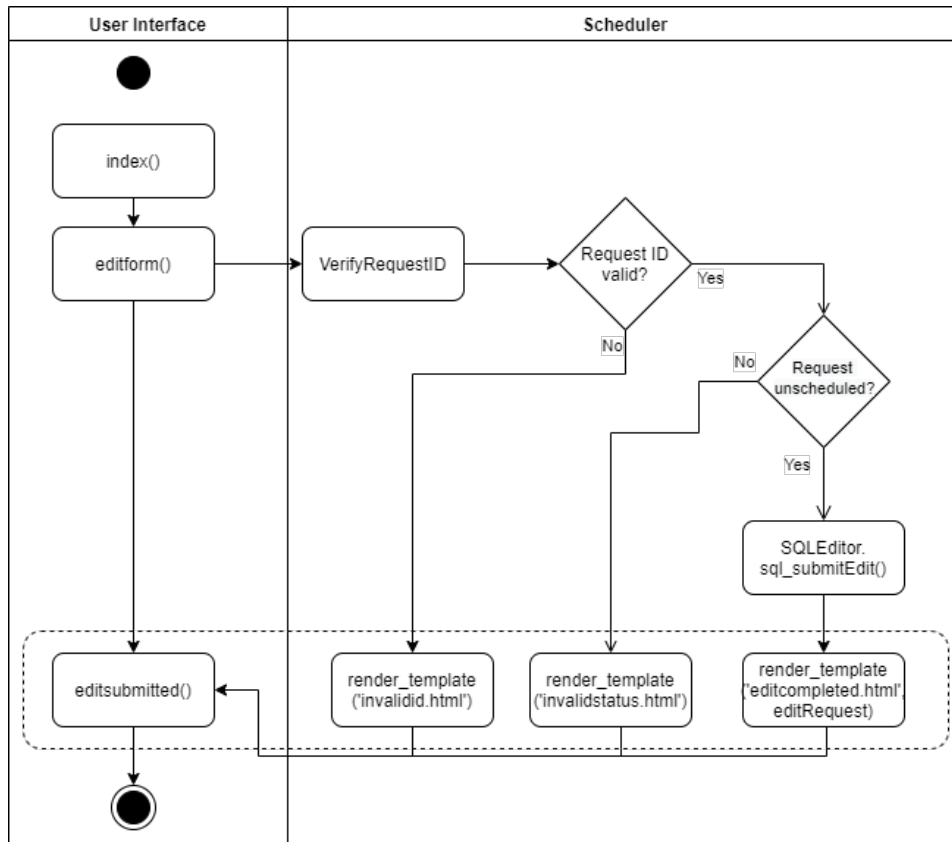


Figure 13: UML activity diagram for editing a request

Delete Request

The first part of the behaviour of a request deletion is practically the same as for the request edit. A request ID verification is needed for the system to know which request needs to be deleted. After the verification, the system tracks down if the request is still unscheduled. The UI routes are different as are the templates rendered.

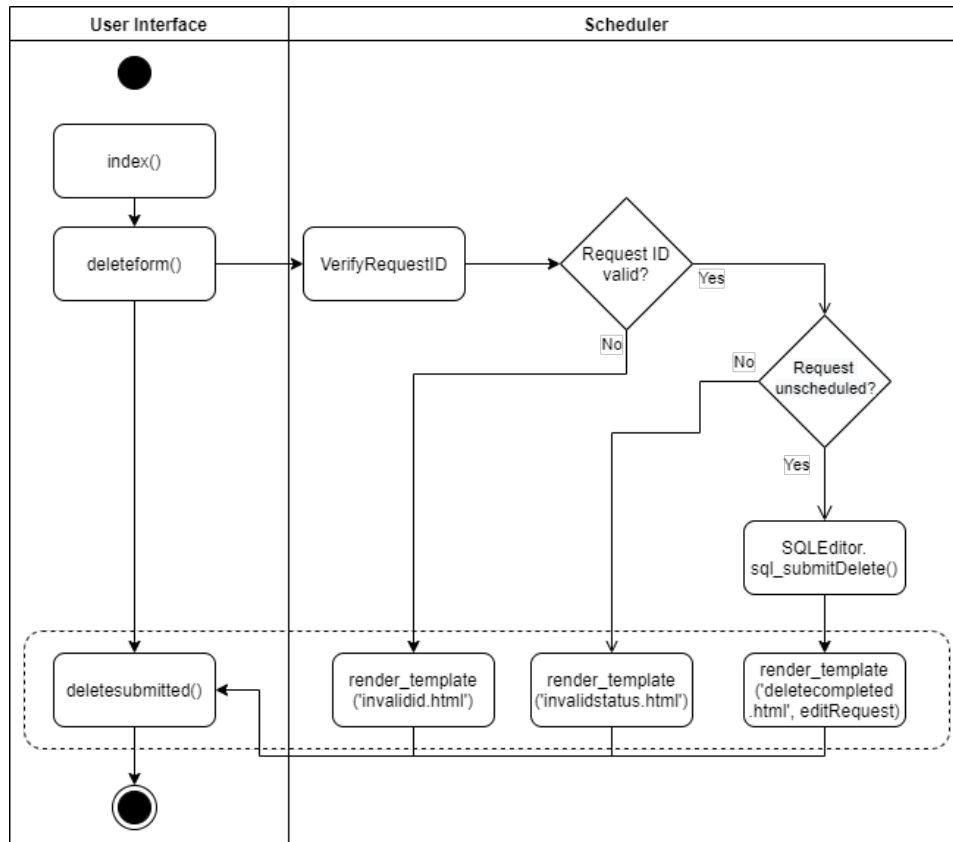


Figure 14: UML activity diagram for deleting a request

5.2.4 Use Case 2: Complete Request

In Figure 15 the use case of a completed request, is depicted. If this event occurs, instantly a timestamp is added to this particular request. Now the request can be verified if it is delivered in time. For now, if *Complete Request* occurs it will instantly trigger scheduling the next request followed up by assigning the next request in line or a charging request to the concerning AGV. The latter will happen if the AGV is unable to transport any of the unscheduled request from SQL database. This could occur in case of a battery charge shortage or an empty database.

5.2.5 Use Case 2 in Detail

The completion of a request is a trigger event. Meaning that it will trigger a sequence of events which is partly covered in Chapter 5.2.3 at the Submit Request use case. All four mentioned

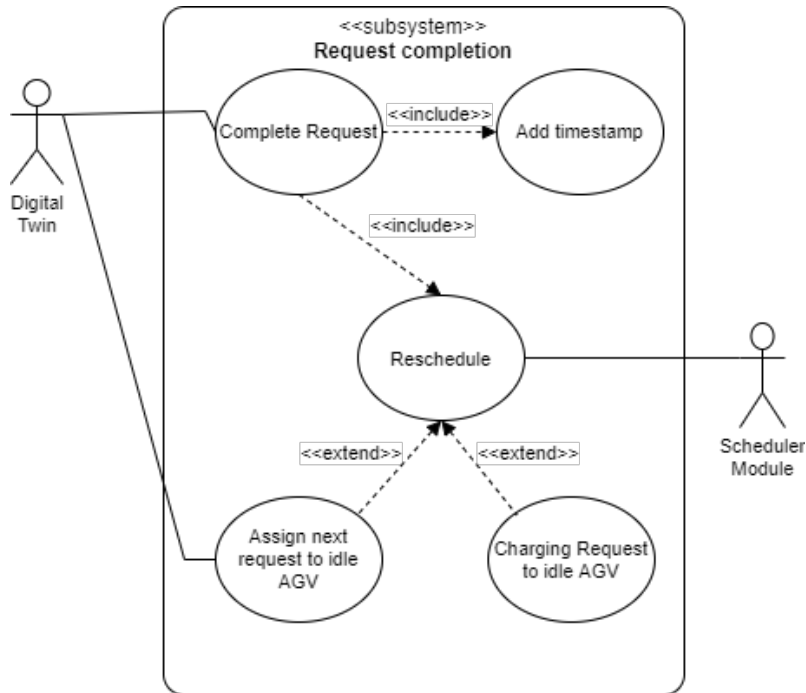


Figure 15: Request completion as a use case diagram

possible results can occur again. The only difference is that the completion of a request is an internal trigger as it occurs in the Digital Twin.

For the UML Sequence diagram when a request is completed, see Figure 16. The remaining possible results, after completion of a request (with function name: *OnAGVDoneWithThisAssignment*), are the same as in Figures A2, A3 and A4 with the internal trigger replacing (*OnAGVDoneWithThisAssignment*) the external trigger (submitting a request). As can be seen, the part of processing the submitted request is left out. Note that after a completed request, the Digital Twin sends a request update over MQTT topic *communicationframework/requestStatus* which updates the status of the concerning request to *processed*.

Currently, the completion of a request is the only internal trigger defined. Future users might want to implement additional internal triggers. A global UML Sequence diagram for internal triggers is generated in Figure A5. This sequence is initiated with a global internal trigger event (an arrow pointing at itself is added in Figure A5: *internal trigger*). This is actually the global case of when an internal trigger occurs in Prespective.

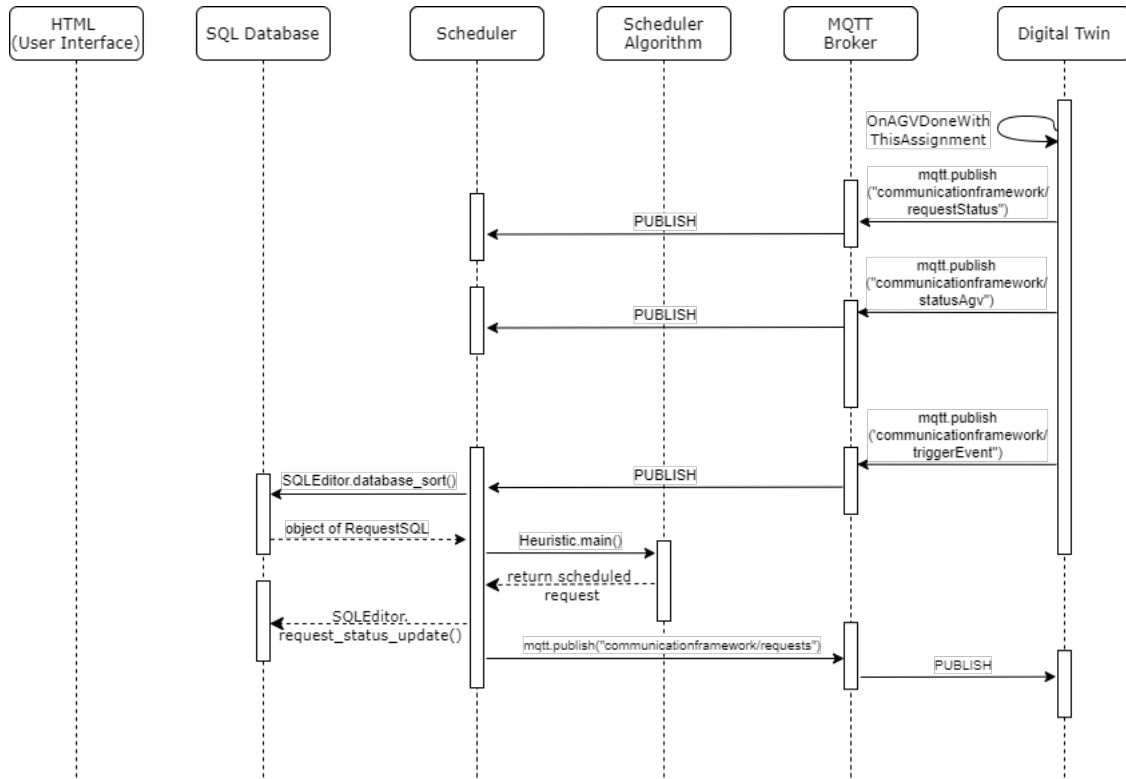


Figure 16: UML Sequence diagram in case of a completed request

5.2.6 Use Case 3: Find a Request

The last use case is when a user wants to obtain a request’s status. The user only has to enter a request ID to show its details. The details that will be displayed are defined in Table C2. The use case is depicted in Figure 17. After the initiation of this use case (*Finding Request* in Figure 17) the request ID is verified. If correct, the Scheduler Module will obtain the request’s attributes for the concerning request which will be followed by displaying them via the UI.

5.2.7 Use Case 3 in Detail

To clear up its exact behaviour, a UML Activity diagram suits best. It is quite similar to the previous activity diagrams defined in this chapter. In the diagram for use case 3 (see Figure 18) only one check has to be performed. The user entered request ID is valid-

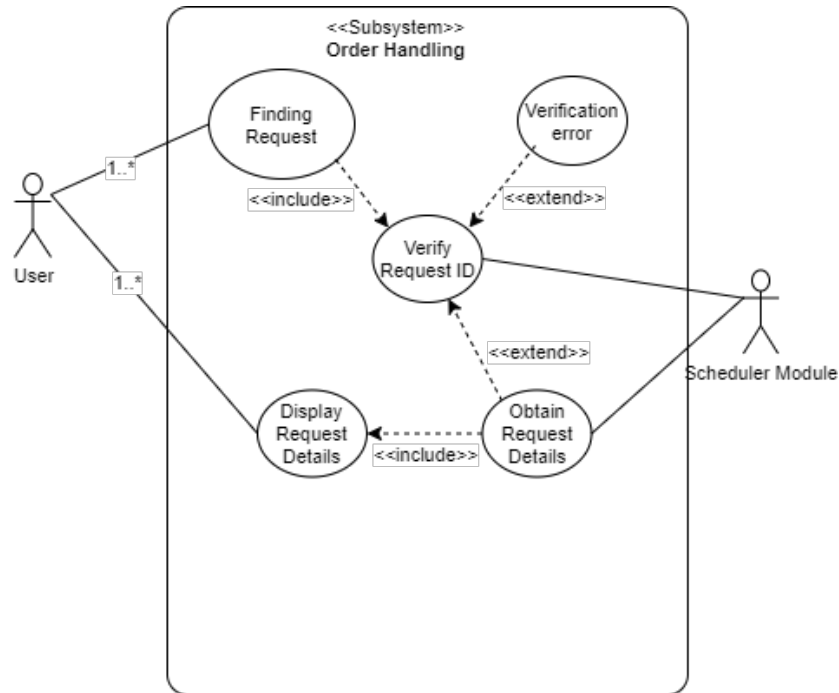


Figure 17: UML Use Case diagram for finding a request.

ated. If incorrect it shows the *"invalidid.html"* page. If validated, the Scheduler Module requests for the details using *SQLEditor.sql_find_request()*. The details will be displayed using *render_template('editcompleted.html', findRequest)*. In this function *findRequest* is the object of the request which was inquired. More information regarding this function can be found in Chapter C.2.

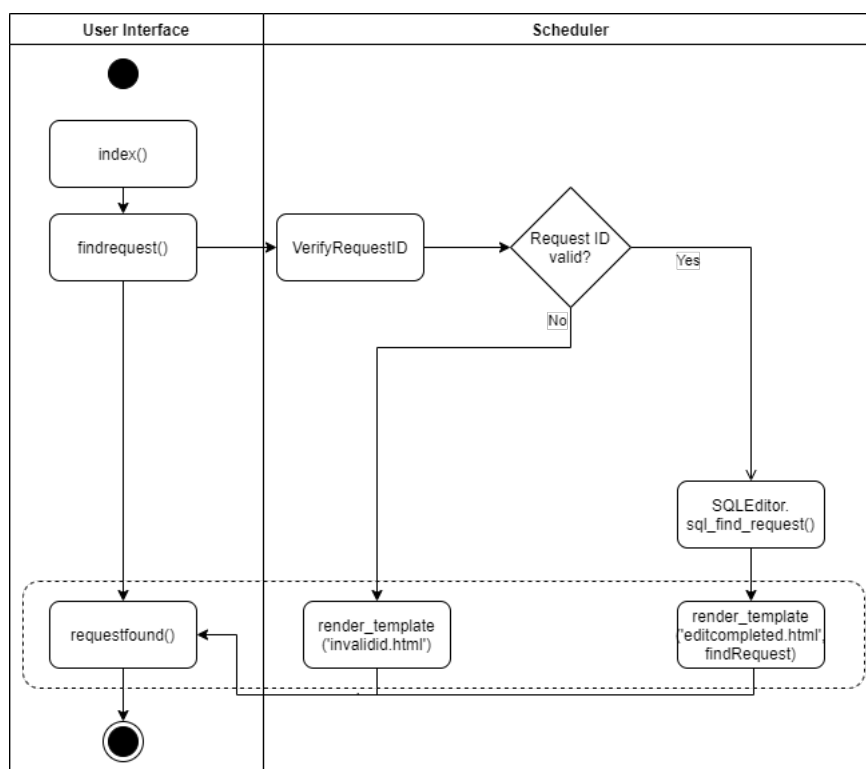


Figure 18: UML activity diagram for deleting a request

6 Validation and Discussion

In order to validate the CF on system level, the communication sequences are the main focus. The most important part for the communication validation is between the Digital Twin and the Scheduler Module. This communication makes use of the MQTT protocol. As can be seen in the UML Sequence diagrams in Appendix A, the MQTT messages play a big part in the inter-modular communication. In the first part of the validation, a few test runs are done to obtain the sequences in which MQTT communication is performed. For each run, all sequences are checked and validated using a stripped version of the UML Sequence diagrams depicted in Appendix A.

In the second part of the validation, the main functionalities of the CF are assessed and validated using the Functional Design document stated in Chapter 4. This chapter is set up to clearly state the functionalities the software should encompass. Most functionalities defined in this document are inherited from the ISO 25010 [11] standard for assessing software quality. This standard is set up to support developers specifying and evaluating quality requirements [11]. A short introduction to the ISO 25010 will be introduced in Chapter 6.2.

6.1 MQTT Communication Validation

In order to validate all sequences regarding MQTT messages, all possible sequences have to be stated. With the help of the aforementioned UML Sequence diagrams (see Appendix A) and the explanation about the behaviour of the CF (see Chapter 5.2) four MQTT flow diagrams are set up (see Figures 19 and 20).

6.1.1 Test set up

With the possible sequences set out, the system can be executed to see if it displays this behaviour. Analyzing MQTT behaviour can be performed using a MQTT client software. In this validation *MQTT.fx* is used (see Appendix G.1.1). With the help of this software an MQTT client can be initialized by connecting to the MQTT broker. Just like with any other client, topic subscribe and publish functionalities are included.

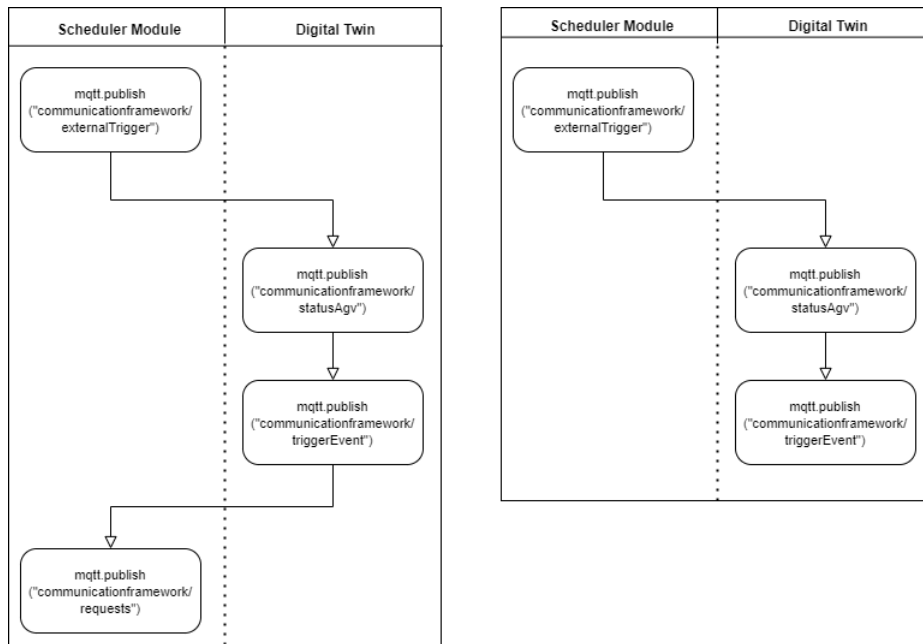


Figure 19: MQTT flow chart when triggered externally. **a)** A suitable request is found. **b)** No suitable request is found

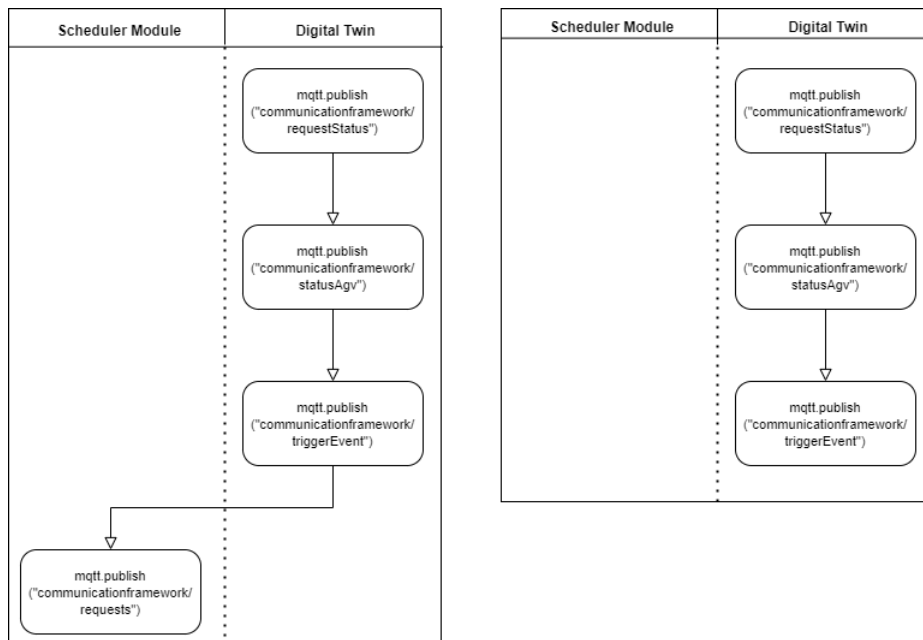


Figure 20: MQTT flow chart when triggered internally. **a)** A suitable request is found. **b)** No suitable request is found

For analyzing the MQTT flow, this client is subscribed to all possible MQTT topics over which data is being sent which are:

- Topic "communicationframework/requests": All scheduled requests from Scheduler Module to Prespective.
- Topic "communicationframework/statusAgv": AGV's status from Prespective to Scheduler Module.
- Topic "communicationframework/requestStatus": Used for updating a request's status.
- Topic "communicationframework/externalTrigger": Used when an external trigger occurs.
- Topic "communicationframework/triggerEvent": Used to trigger the Scheduler Module's algorithm to schedule a request.

More information regarding the specific usage of these channels can be found in Appendix C.5.

Three cases will be tested on their specific MQTT message sequences. Each case is executed ten times. Using Figure 1 as layout with 5 AGVs performing the transport of orders, an analysis is made. The following cases are generated:

- **Case 1:** The database is initialized with 10 requests. The AGVs are set to have a high charge. This situation is based on use case 2: completion of a request (see Chapter 5.2.4). It simulates the occurrence of an internal trigger (completion of request). This case is set up to validate the accuracy of scheduling requests as a result of internal triggers without interference of charging requests.
- **Case 2:** The database is initialized with 10 requests. The charge of all AGVs is set extremely low meaning not all requests can be transported as the maximum charge of an AGV is lower than some requests. This case is set up to force the Scheduler Module to generate charging requests. The ability to schedule charging requests is therefore validated.
- **Case 3:** Start with an empty database. 5 request are submitted using the UI. This case is likely to generate more messages as it can't be self-triggering although it tries to

after completing a request (can't be triggered due to empty database). The AGVs are set to have a high charge. This situation is set up to generate use case 1: submitting a request (see Chapter 5.2.2). This case validates the scheduling sequence in occurrence of an external trigger.

All three cases are executed and the sequence occurrence is validated. All sequences should be explainable by using the specified behaviour of the CF.

6.1.2 Results

For each case, the sequences are captured using *MQTT.fx*. In Table 1 the number of a specific sequence occurring is set out. The columns represent a specific sequence matching the sequences depicted in Figures 19 and 20. For each case there exists an initial sequence that differs a bit from the sequences stated in Figures 19 and 20. These initial sequences will be explained per case. There were no deviations in sequence occurrences meaning each execution resulted in the same sequences occurring for each separate case.

Case	18a)	18b)	19a)	19b)	Initial	Other
1	0	0	5	5	1	0
2	0	0	8	5	1	0
3	4	0	0	5	1	1

Table 1: number of sequence occurrences per case. Each column represents a specific sequence.

Firstly, case 1 is analyzed. As can be seen in Table 1 there are no occurrences that start with an external trigger. This is logical, because no requests are submitted through the UI. The initial sequence is depicted in Figure 21. The CF always starts with sending out all AGV's statuses over topic "communicationframework/statusAgv" followed by a trigger over topic "communicationframework/triggerEvent". With the database filled and all 5 AGVs idle, this first sequence results in 5 scheduled requests being sent over topic "communicationframework/requests".

6. VALIDATION AND DISCUSSION

The remaining sequences are 5 times the sequence in Figure 20a) and 5 times the sequence in Figure 20b). This can be explained, because the AGVs have enough charge (as stated in the case description) and 5 requests left to schedule, the sequence in 20a) occurs 5 times. After each of these requests is completed (which is a internal trigger), the Digital Twin goes on to sending out the updated AGV's status followed by a trigger. The Scheduler Module is unable to schedule requests as it now lacks them in the database.

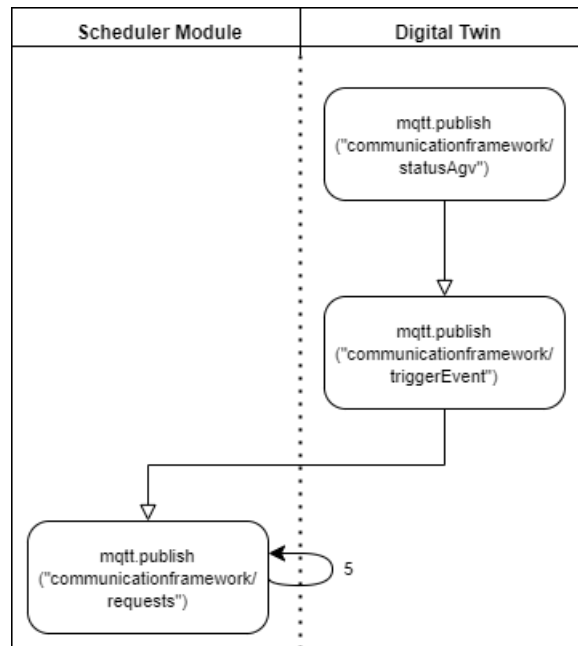


Figure 21: Initial MQTT sequence for Case 1.

In the second case again only internal triggers occur as no requests are submitted via the UI. The initial sequence of this case is exactly the same as in Figure 21. This means already 5 out of 10 requests in the database are scheduled after starting the CF. Because of the low charge of each AGV, all AGVs are forced to charge after their first processed request; a charge request is initialized and sent to the Digital Twin over topic "communicationframework/requests". This sequence is exactly the same as in Figure 20a). This accounts for 5 out of 8 occurrences of this sequence. The remaining ones are of processed requests after charging. This leaves us with 2 unscheduled requests. The system is unable to transport them as the charge after charging is still too low for its transport.

Case 3 starts with an empty database. The initial sequence starts off with sending the status of all AGVs (topic: "communicationframework/statusAgv") followed by a trigger (topic: "communicationframework/triggerEvent"). Because of the empty database, this is the end of the initial sequence. During runtime, five requests are entered via the UI. Each entered request follows the sequence of Figure 19a) except for one. This is an exceptional case. It follows the same sequence as in Figure 19a) but it skips the step of sending the AGV's status (topic: "communicationframework/statusAgv"). This exceptional case occurred right after the first request is submitted. The CF noticed that the status of all AGVs didn't change from the start of the simulation until submitting the first request. Although it is an effective feature because it saves sending abundant messages, this behaviour was unaccounted for. More tests were performed with an initial empty database. Upon submitting a request the CF displayed the same behaviour in all test instances. The sequence in Figure 20b) was followed 5 times. This sequence is started after completing a request and once again the database was empty therefore the Scheduler Module couldn't provide the Digital Twin with a new request as expected.

The MQTT communication validation shows that the CF's behaviour is as expected. It did show one abnormality, which doesn't have an impact on the CF. In fact, it would be a welcome extra feature. The Digital Twin and the rest of the CF are cooperating as wanted. Robustness is proved as each execution per case resulted in the same sequence occurrences.

6.2 Functional Design Validation including Software Assessment

In this section an evaluation is done regarding the functional design document which complies with ISO 25010 characteristics. ISO 25010 is used for software to design for its quality. It is known as a widely accepted norm. It has its own certification. This norm can also be used as a guideline when facing decision making. It contains two quality models with each its own characteristics. The first is Quality in Use: how well can the product be used by specific users to reach their goals? The second model is Product Quality. Together, they consist of 13 characteristics and 40 subcharacteristics.

The metrics of every characteristic always has been a complicated part. ISO/IEC 25010 prop-

Quality in use	Product quality	Product quality (cont.)
<ul style="list-style-type: none"> • Effectiveness • Efficiency • Satisfaction <ul style="list-style-type: none"> - Usefulness - Trust - Pleasure - Comfort • Freedom from risk <ul style="list-style-type: none"> - Economic risk mitigation - Health and safety risk mitigation - Environmental risk mitigation • Context coverage <ul style="list-style-type: none"> - Context completeness - Flexibility 	<ul style="list-style-type: none"> • Functional suitability <ul style="list-style-type: none"> - Functional completeness - Functional correctness - Functional appropriateness • Performance efficiency <ul style="list-style-type: none"> - Time behaviour - Resource utilization - Capacity • Compatibility <ul style="list-style-type: none"> - Co-existence - Interoperability • Usability <ul style="list-style-type: none"> - Appropriateness recognisability - Learnability - Operability - User error protection - User interface aesthetics - Accessibility 	<ul style="list-style-type: none"> • Reliability <ul style="list-style-type: none"> - Maturity - Availability - Fault tolerance - Recoverability • Security <ul style="list-style-type: none"> - Confidentiality - Integrity - Non-repudiation - Accountability - Authenticity • Maintainability <ul style="list-style-type: none"> - Modularity - Reusability - Analysability - Modifiability - Testability • Portability <ul style="list-style-type: none"> - Adaptability - Installability - Replaceability

Table 2: ISO/IEC 25010 model characteristics and subcharacteristics [12]

erties can not be determined using well-defined units like length, mass, time etc [11]. The attribute metrics can be application specific. Usually, it is unnecessary to plan for all characteristics. It should be determined compliant to the end product's requirements. For instance, why would security be assessed for a new digital calculator?

For each characteristic separately it is decided whether it is assessed in Table B1, which can be found in Appendix B. Green fields are used if a characteristic needs assessment and orange if not. How it is achieved for every characteristic is mentioned in the last column. Note that in this validation, the part *Quality in use* is skipped. This part is focused on the overall usability of users which currently can not be assessed.

A validation was done for each of the ISO 25010 characteristics mentioned in Table B1 by using validation and verification methods for addressing ISO 25010 characteristics [8]. Most characteristics are validated using text in this report. The complete validation of this part can be seen in Table 3. Each requirement mentioned in the functional design chapter is stated in this table and how it is achieved. The requirements set before designing the CF are all accounted for meaning the set of requirements is validated.

Requirement	How is it achieved?
Ability to initiate Use Cases (Functional suitability)	In Chapter 5.2 the behaviour of the proposed CF is set out. Each Use Case described can be initiated from the UI.
Self-Triggering	The CF is build to handle internal and external triggers. Each of them have their own MQTT topic ("communicationframework/triggerEvent" and respectively "communicationframework/externalTrigger"). Once the CF is started the internal triggers are produced by the CF itself and external trigger initiated from submitting a request (additional internal/external triggers can be defined.
User Error Protection (part of Usability)	The CF should prevent wrong data user input. This is achieved by defining certain rules in the URL routes inside the Flask application. See Appendix C.3 for these rules.
Interoperability (part of Compatibility)	An extensive I/O model is set up which is prove that all Modules are able to handle input from external Modules where needed. See Appendix C for this I/O model.
Recoverability (part of Reliability)	The external SQL database set up fully covers this requirement. In case of an error, the SQL stays untouched. All request's statuses are stored.
Maintainability	The provided scripts are full of print statements in order to track the CF's process flow. The MQTT messages can be traced using additional software (see Appendix G.1.2 for recommended software).
Adaptability (part of Portability)	The software documentation consisting of a I/O data model, Installation Guide and a User Manual assures: CF installability on other computers and adaptability of each separate module.

Table 3: Combining the functional requirements and ISO characteristics for the sake of validation.

6.3 Discussion

In Chapter 1 the aim of this research was stated: design a modular CF that can process incoming requests from multiple tenants using a Digital Twin. The designed CF should be the missing piece of the puzzle that is called the Brainport Industry Campus. Several decision support tools (among which single-load and multi-load scheduling algorithms) can be tested.

First research has been done on what is needed for creating such a framework. The different components that should communicate were first set up as a template (see Figure 2). This template is filled in with technical components and describes how their behaviour is structured.

The proposed CF delivers an environment where algorithms for similar situations can be tested in a dynamic environment that produces data which can be used to feed these algorithms. It has been setup in modular fashion which opens up the possibility of implementation in a real production environment. With the Digital Twin included, testing before implementation is made viable. Before implementation in a real life manufacturing environment a deployment plan has to be developed. How this exactly works is reserved for further research.

Its relevance came to light while progressing in this project. Numerous enquiries about the CF were made as word came out about the proposed CF. It is clear that the BIC benefits from this project. The OPAC group (Operations, Planning, Accounting and Control) of the Technical University of Eindhoven is preparing to go more in depth by setting up a taskforce that combines the BIC case with robotics research. This would enhance the Digital Twin approach of the real world.

More enthusiasm came from the academic world. Another former Master Student is interested in using the framework for his ongoing research. This research is focused on using Semantic Web for data integration with the focus on data obtained from sensors on machines and resources in a production environment. Using the Semantic Web could also merge the use of multiple FMS's into one common FMS. The CF is designed such that heterogeneous AGVs can be tested in a manufacturing environment.

Usage of the CF in a multi-tenant environment has been made clear throughout this project. Although the CF is designed for a multi-tenant environment, it could be used by singular companies where multiple employees could use the CF. E.g. an enquiry was done for using the CF in the logistics of hospital beds transported by AGVs.

The CF has also made name outside of the University. It is presented at the EAISI summit (Eindhoven AI Systems Institute) which was digitally attended by over 80 people.

7 Limitations and Conclusion

7.1 Limitations

Although the interest from the business and academic world is clear, the proposed CF still has its limitations. The setup has been made but more research awaits before implementing the CF. The topics for further research are listed below.

- *Modularity Validation*: the process of designing the CF was based on the concept of modularity. The extensive system documentation is proof of that. It still remains to be seen if it was enough. Further research should try and replace modules in order to fully validate its modularity.
- *Collision avoidance*: the Digital Twin software Perspective has a lot of potential in the manufacturing world. Although it is one of few that provides a dynamic environment that generates real-time data, its functionalities and UI are still under development. Collision avoidance is one of these functionalities that is currently being developed.
- *User verification*: currently every tenant that would initiate a use case is able to do so for each submitted request. Before implementing the CF in a production environment some sort of user verification needs to be established. Users should only be able to perform operations on their own orders.
- A thorough *validation of the process flow* inside of the Scheduler Module still needs to be executed. Although the process flow itself is mapped extensively in Chapter 5.2, in the validation chapter its behaviour is seen as a black box. This distributed software package makes it hard to validate the pieces of software separately. A full validation of specific communication within a module combined with the communication on system level is therefore left out of the picture. In the world of software development this would be a job done by a team of test engineers.

7.2 Conclusion

In this research an extensive CF is proposed in which algorithms can be tested in a Digital Twin capable of bi-directional communication. First the problem and scope of the project are defined followed by performing research into communication frameworks. The gained knowledge was applied and converted to fit the multi-tenant aspect of this project. The resulting CF is designed by bringing together multiple modules (User Interface, Database, Scheduler Module, Digital Twin and the MQTT communication protocol) and making them communicate with each other by generating communication interfaces between the modules. By implementing a highly customized Digital Twin capable of bi-directional communication through a scalable MQTT-broker, a strong foundation is built for increasing data generation from the shop floor. The modularity approach makes it possible to implement custom made factory layouts and custom algorithms enabling usage for multi-tenant (not only in the BIC) and single-tenant environments. These abilities make the CF align with the vision of smart manufacturing 4.0. Real time data exchange in order to optimize manufacturing processes is the biggest trend in the manufacturing business. The validation proves that the inter-modular communication is behaving as expected. The overall functionalities of the CF are validated which proves all required functionalities are implemented.

The research stands out because it is focused on a multi-tenant environment which currently is uncommon in the manufacturing business. The BIC is a unique factory whose vision is ahead of its time. If this vision proves to be successful the proposed CF could be highly desired by followers of the BIC's vision.

Another big advantage is the possibility of using heterogeneous AGVs. Currently each brand of AGV operates with their own FMS. The need for a common FMS despite its brand is a well known problem which receives much attention by researchers around the world. Because of the AGV heterogeneity, the CF diminishes this problem enabling the testing of AGVs with diverse capabilities.

The main research question proposed is stated in Chapter 3:

How to develop a communication framework to implement a scheduling algorithm for a heterogeneous AGV fleet in a multi-tenant environment?

This question is answered by setting up a framework with accessory modules. These modules are designed to communicate with each other passing data. By its modularity and complementary documentation the CF is set up to house custom made scheduling algorithms. The UI makes the CF usable by multiple tenants.

The answer to the main research question is answered with the help of the sub questions set up:

1. *What is needed for designing software?*
2. *How can all components communicate on the edges using different syntax?*
3. *How to assess the quality of this software?*

The first sub question is extensively answered by researching the components needed for the CF which later were filled in by describing the structure of the proposed CF. The second question is accounted for by stating how each module communicates with its environment by stating each module's communication protocol. The third question is handled in the validation by using the certified ISO 25010 norm for software assessment.

Future research is still awaiting. It is possible to deploy the framework in a real manufacturing environment, which replaces the Digital Twin. How this exactly works remains to be investigated. This research should specify what hardware capabilities are needed. The proposed CF is developed to run on a single computer. For large factories a technical deployment plan is needed.

Appendices

A Additional UML diagrams to Use Case 1 (Submit, edit and delete request)

A.1 Use Case 1 in detail

A. ADDITIONAL UML DIAGRAMS TO USE CASE 1 (SUBMIT, EDIT AND DELETE REQUEST)

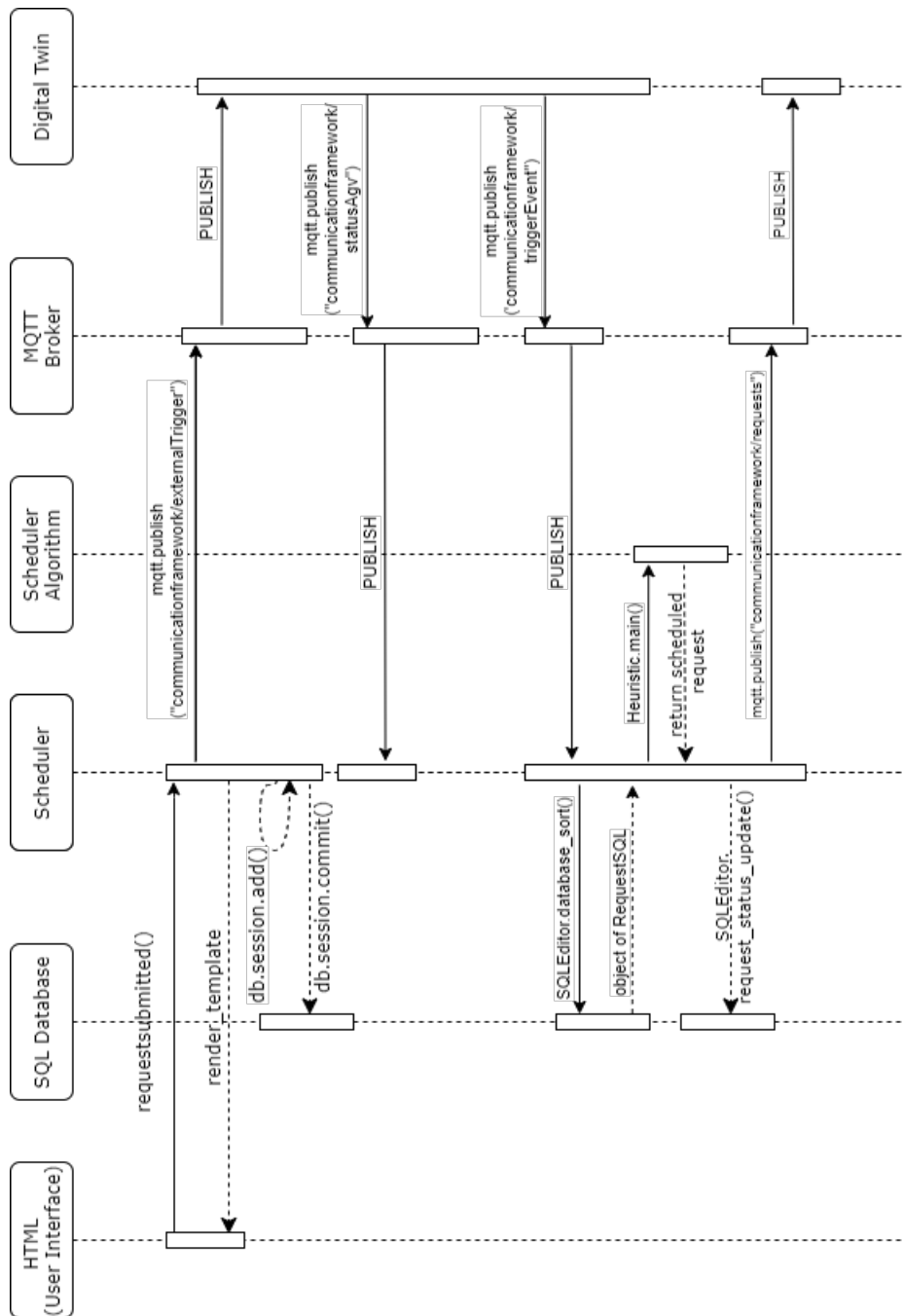


Figure A1: UML Sequence diagram in case of submitting a new request

A. ADDITIONAL UML DIAGRAMS TO USE CASE 1 (SUBMIT, EDIT AND DELETE REQUEST)

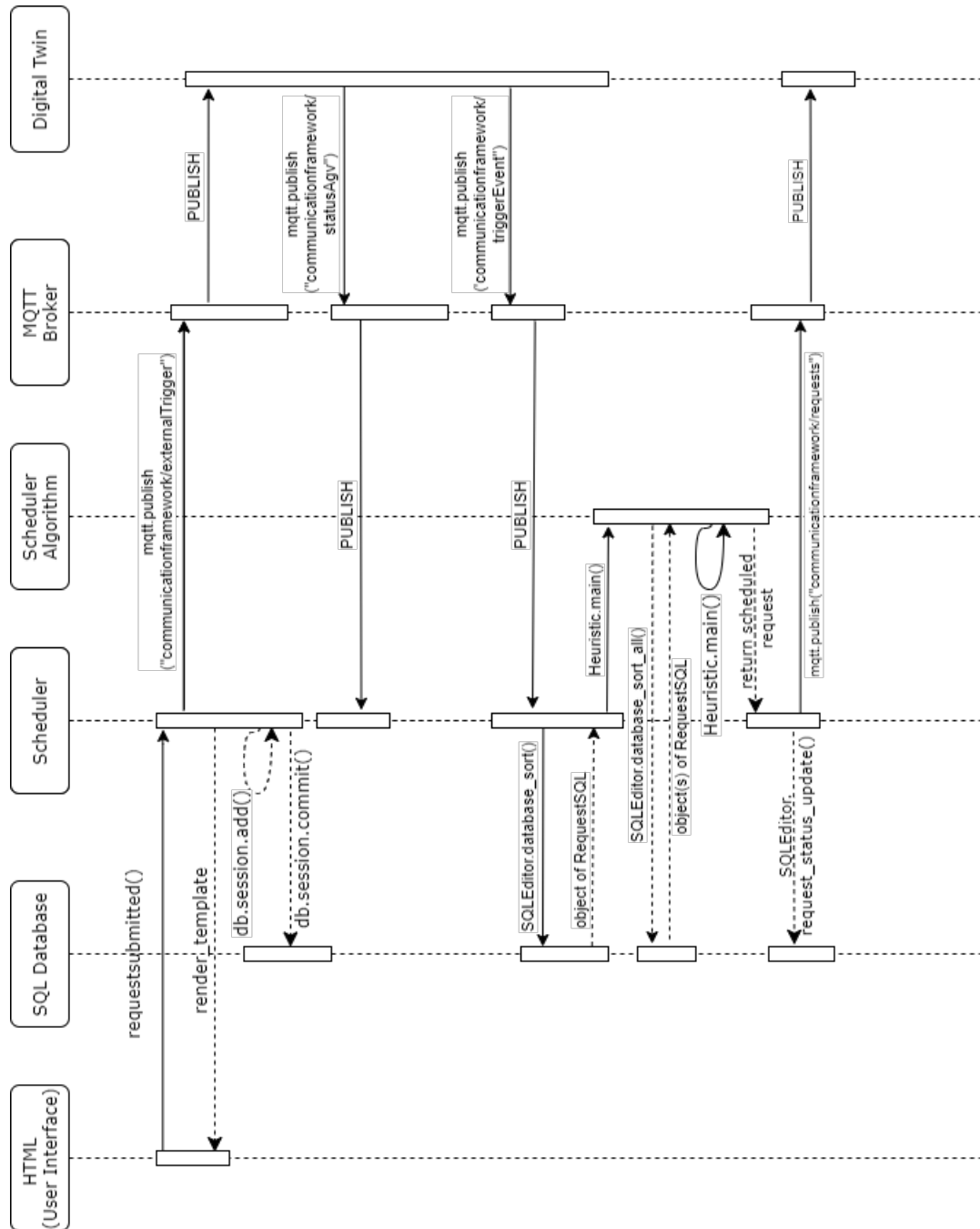


Figure A2: UML Sequence diagram in case of submitting a new request and first selected request exceed AGV's charge. It queries for all unscheduled requests in database and finds a suitable request.

A. ADDITIONAL UML DIAGRAMS TO USE CASE 1 (SUBMIT, EDIT AND DELETE REQUEST)

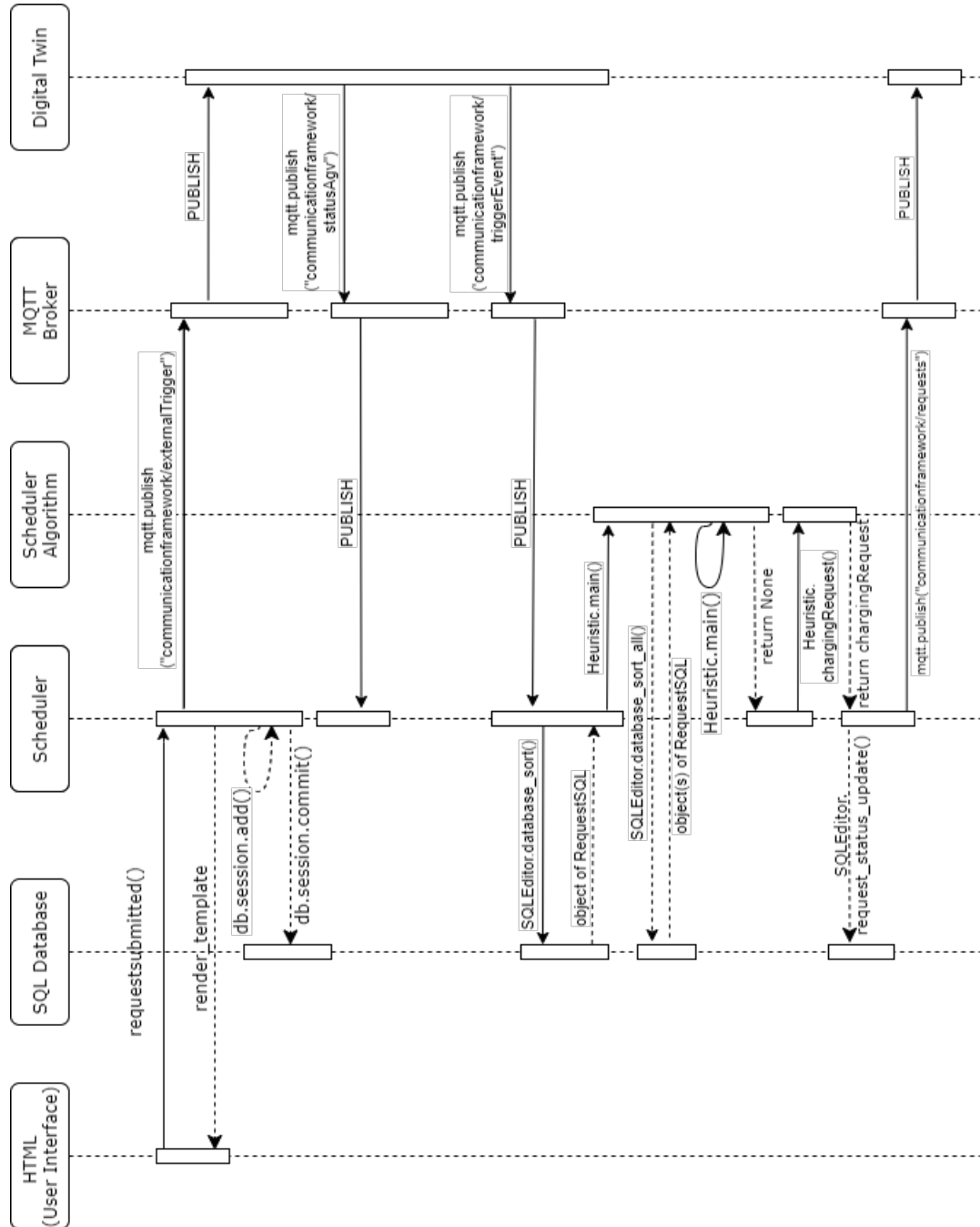


Figure A3: UML Sequence diagram in case of submitting a new request and no suitable requests are found that match the AGV's charge

A. ADDITIONAL UML DIAGRAMS TO USE CASE 1 (SUBMIT, EDIT AND DELETE REQUEST)

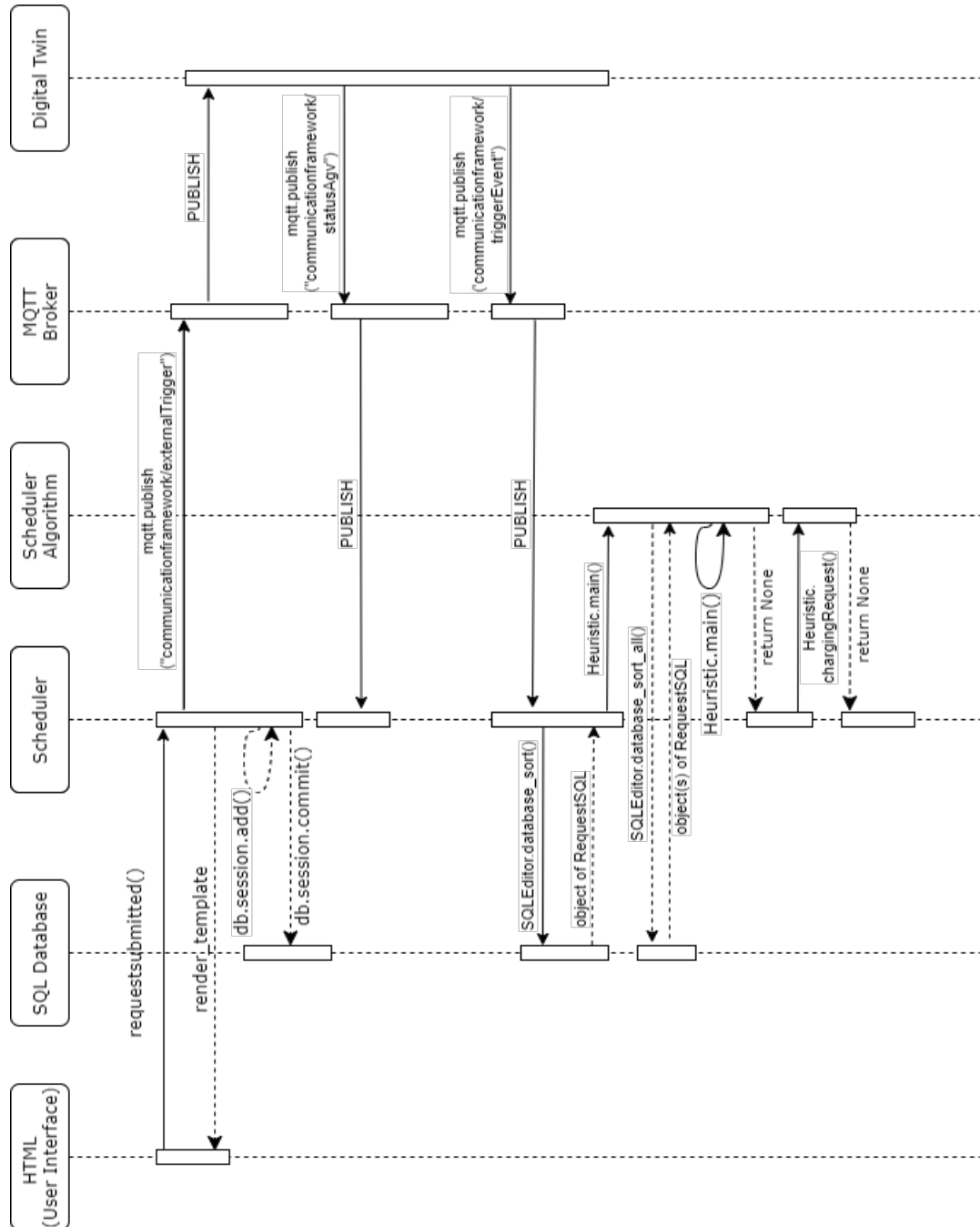


Figure A4: UML Sequence diagram in case of submitting a new request and no suitable requests are found that match the AGV's charge. The CF tries to generate a charging request but the concerning AGV already has reached the charge's upper threshold.

A. ADDITIONAL UML DIAGRAMS TO USE CASE 1 (SUBMIT, EDIT AND DELETE REQUEST)

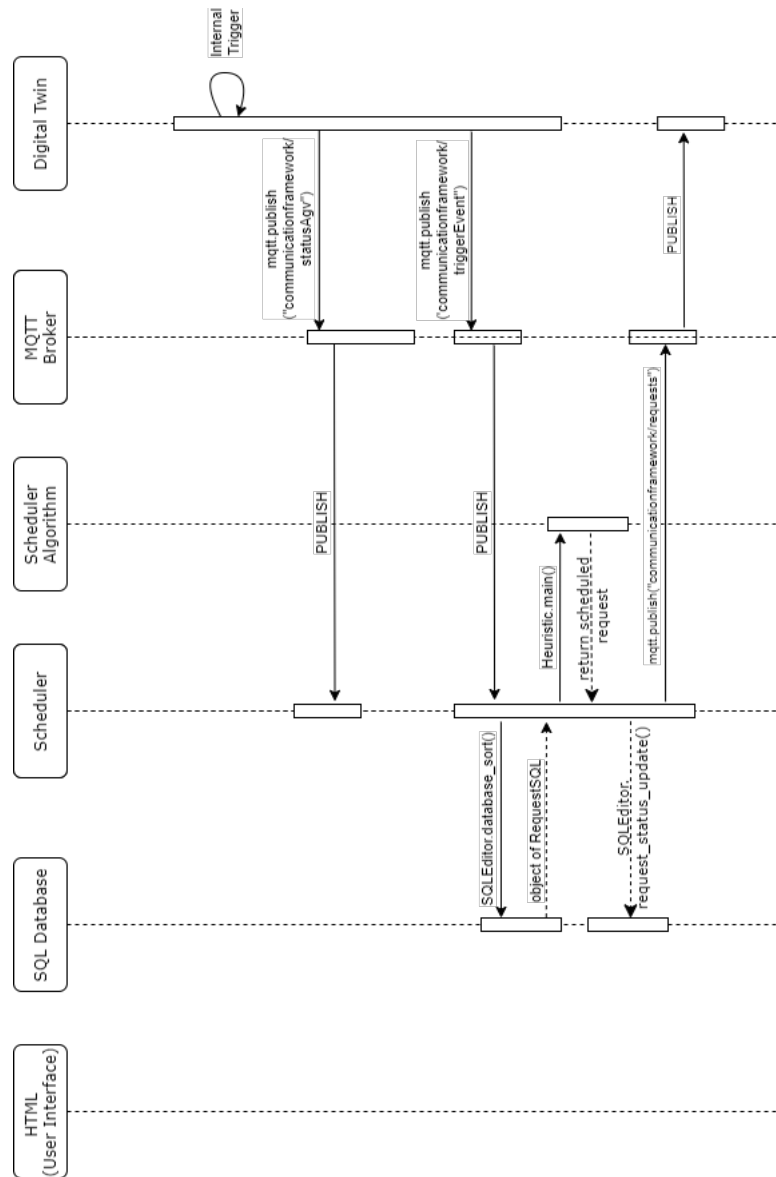


Figure A5: UML Sequence diagram in case of an internal trigger event occurs

B ISO 25010

Group	Item	How it is achieved
Quality in product		
Functional suitability	Functional completeness	Checklist functional requirements
	Functional correctness	Verification of each functional requirement
	Functional appropriateness	Two above imply appropriateness. See functional completeness and functional correctness.
Performance efficiency	Time-behaviour	Set a time goal.
	Resource utilization	Could become applicable if time behavior requires another resource utilization method i.e. parallel processing, scaling up or scaling out.
	Capacity	Testing that a set of input events can be handled within the agreed timeframe, to produce the desired output
Compatibility	Co-existence	Not applicable. The system is designed to work within its own (virtual) machine, with the interfaces as connection surfaces.
	Interoperability	Interface testing. <u>prove</u> that the system can handle input from other systems and produces output that can be used by other systems.
Usability	Appropriateness <u>recognisability</u>	Not applicable; After signing the functional design document, this quality item is implied.
	Learnability	Not applicable. Used by specialists.
	Operability	By nature of the system operability is not applicable. No user interface. No testing needed
	User error protection	All inputs can be tested for right format before running algorithm.
	User interface aesthetics	Not applicable. No interface is developed.
	Accessibility	Not applicable. Only used by specialists
Reliability	Maturity	First version of system. Not applicable, implied by other green characteristics.
	Availability	Not applicable. The availability of infrastructure is assumed.
	Fault tolerance	See User error protection.
	Recoverability	When no schedule is generated or an error occurs, the prior schedule and the <u>to be</u> scheduled request should be preserved.
Maintainability	Modularity	Not applicable. Code is modular from itself.
	Reusability	Not applicable due to modularity of code
	Analyzability	A simulation model could help, ledger
	Modifiability	Provide comments in code and strong System documentation.
	Testability	Not applicable. Testability is implied when other tests are done correctly
Portability	Adaptability	Prove the system is adaptable to the user's needs.
	<u>Installability</u>	An installation guide and user manual will suffice.
	<u>Replaceability</u>	Is the system capable of replacing mpdules?
Security	Confidentiality	Not applicable.
	Integrity	Not applicable, out of scope. See recommendations
	Non-repudiate	See ledger.
	Accountability	See ledger.
	Authenticity	See ledger.

Table B1: ISO/IEC 25010 model characteristics verified if test is needed

C Input/Output Data Model

In order to give more insight in how data is formatted and converted in and between modules a I/O data model is set up. For each separate module all needed information regarding the usage of data and internal functions. This document is intended for users that want to implement custom modules. It will become clear what data (including its format) is fed to each module (input) and what output is required for the CF to run smoothly.

C.1 Scheduler Algorithm

The algorithm implemented needs a few input parameters to run. In the algorithm there are 3 types of functions.

```
RequestToAGVNew, retrigger, agv = Algorithm.main(tempstatus, requestSort) (1)
```

```
chargingRequest = Algorithm.chargingRequest(tempstatus, Request): (2)
```

All functions take *tempstatus* as input. This parameter constitutes the status of all AGVs. This variable is gained from the Digital Twin and stored as a dictionary. It is built up as follow:

[Start of List
{	Start of first object
'AGVID': <i>i</i> ,	Unique identifier for an AGV with integer <i>i</i> being the <i>i</i> 'th AGV.
'CurrentLocationIsMachine': <i>loc</i> ,	'CurrentLocationIsMachine' denoting the current location of AGV <i>i</i> . <i>loc</i> stands for the representative integer of the corresponding location AGV <i>i</i> is at. For AGV's that are processing a request <i>loc</i> = -1.
'Charge': <i>cha</i> ,	Current charge with value <i>cha</i> of AGV <i>i</i> with type float
'MaxCharge': <i>mCha</i> ,	Maximum charge possible on AGV <i>i</i> with type float
'lowerCharge': <i>lCha</i> ,	Lower threshold for charge level. Below this level charging should take place. <i>lCha</i> has type float
'upperCharge': <i>uCha</i>	Upper threshold for charge level. If charging takes place, at least this level should be reached. <i>uCha</i> has type float
},	End of first input object
...	Any additional objects
]	End of List

Table C1: Data format built up for *tempstatus*

Function (1) is the main part of the algorithm. It takes an additional parameter as input: *requestSort*. This second input is defined by the request which has to be scheduled. This request is gained from sorting the SQL-database on the request with the delivery date first to come. This request can be called upon by:

```
requestSort = SQLEditor.database_sort() (4)
```

The *requestSort* parameter is an object of the class *RequestSQL* which is defined in the script *models.py*. It contains all necessary attributes. Table C2 shows what attributes can be found when a query (like (4)) is executed. More information about SQL operations can be found in chapter C.2.

requestSort	Returns object of <i>RequestSQL</i> .
requestSort.requestID	Returns ID of <i>requestSort</i> as integer .
requestSort.orderType	Returns <i>orderType</i> of <i>requestSort</i> as a string . This attribute can have 2 values: "transport" or "charging".
requestSort.pickUpDate	Returns earliest pick up date of <i>requestSort</i> as a string in the format "YYYY-MM-DD"
requestSort.endDeliveryTime	Returns earliest pick up time at pick up date of <i>requestSort</i> as a string in format "HH:MM"
requestSort.source	Returns the source node as integer
requestSort.destination	Returns the destination node as integer
requestSort.requestCost	Returns the requestCost as integer
requestSort.capacity	Returns the which type of AGV's are capable of transporting <i>requestSort</i> as string .
requestSort.status	Returns status of a request as string . Default value: 'unscheduled'. Should be update to 'scheduled' after it's scheduled by the algorithm.
requestSort.assignAGV	Returns AGV that is assigned to transport <i>requestSort</i> as an integer . This attribute has to be determined and written to SQL-database by the algorithm. Empty by default.
requestSort.EndTimestamp	Returns the actual end date and time of completed request as string in format: "YYYY-MM-DD HH:MM:SS". Empty by default. This attribute is filled in by the framework when it gets notified by a trigger.

Table C2: Custom class *RequestSQL* to use requests for storage and operations

The output of (1) is *RequestToAGVNew* which is the processed version of *requestSort*. The

C. INPUT/OUTPUT DATA MODEL

Algorithm should fill *requestSort.assignAGV* and *requestSort.status*. If succeeded to do so, the Scheduler Module formats it before publishing it to the MQTT-broker.

The second output parameter of (1) is *retrigger*. This parameter is currently used in case *requestSort* can't be executed because no AGV has sufficient charge. In case *retrigger* is equal to 1, the algorithm has found another request which can be processed by an AGV with sufficient charge. It does mean (1) should be invoked again.

In case the Idle AGV('s) has insufficient charge for any of the unscheduled requests, the algorithm returns *RequestToAGVNew = None* and *retrigger = 0*.

The remaining output parameter is *agv* which represents which AGV is assigned to the scheduled *RequestToAGVNew*. If *RequestToAGVNew = None* and *retrigger = 0*, the parameter *agv*'s value is arbitrary.

Function (2) is called when the last mentioned case occurs. In this case a charging request should be initiated. The input parameter *Request* is the charging request that has to be created. In order to align it for submitting to the Database as well, it should be stored as an object of class *RequestSQL*.

```
Request = RequestSQL(requestID=requestid ,
    orderType='charging' ,
    source=0,
    destination=0,
    status='unscheduled' ,
    assignAGV=agv)
```

Function (2) should assign the attributes *source*, *destination* and the *status*. Last mentioned has to be set to 'scheduled'. The other parameters should be provided. The parameter *requestID*, and *agv* are provided by the Scheduler Module. The request is also being submitted to the SQL database by the Scheduler Module.

This function can also be triggered when a trigger event occurs from within the Digital Twin. More about this in Chapter 5.2. **Note:** make sure to update the database if a custom made algorithm is used.

C.2 SQL Database

System incompatibility between the SQL database and the Scheduler Module can cause a lot of additional data converting. With the help of Object Relational Mapping (ORM) this problem is prevented. ORM helps in performing operations or queries in SQL initiated from the Scheduler module. This counts for custom made classes as well. A custom class made (called RequestSQL) in the Scheduler module (using Python) is mapped with an object instance of an SQL table.

When a query is done initiated from the Scheduler module to the SQL database, it will return an SQL table object. This object is recognized by the Scheduler module and stored as an object of the custom made class: RequestSQL. In script *models.py* the custom made class is defined:

```
class RequestSQL (db.Model):
    __tablename__ = "requests"
    requestID = db.Column(db.Integer , primary_key=True)
    orderType = db.Column(db.String , nullable=False)
    pickUpDate = db.Column(db.String)
    pickUpTime = db.Column(db.String)
    endDeliveryDate = db.Column(db.String)
    endDeliveryTime = db.Column(db.String)
    source = db.Column(db.Integer , nullable=False)
    destination = db.Column(db.Integer , nullable=False)
    requestCost = db.Column(db.Integer)
    capacity = db.Column(db.String)
    status = db.Column(db.String)
    assignAGV = db.Column(db.Integer , nullable=True)
    EndTimestamp = db.Column(db.TIMESTAMP, nullable=True)
```

All attributes listed are mapped to a column of the database db. By defining `__tablename__` this class is mapped to the database called requests. The *nullable* parameter defines if a particular attribute can be empty. The *primary_key* parameter is a unique identifier which helps quering for a specific database entry. For every attribute the data format is defined as well.

For operations on the SQL database, an SQL editor python script is generated. All func-

C. INPUT/OUTPUT DATA MODEL

tions that can be called upon will be explained. Direct SQL queries can be quite extensive and have their own format. The python package *sqlalchemy* contain these SQL queries in the python language. This can be seen as a data converter. In the framework no direct SQL queries are done. All queries are executed using *sqlalchemy*. For all use cases (submitting, editing and deleting requests) functions have been set up.

Before defining the SQL operations, all input parameters are declared:

Input	Definition	Data format
rID	Request ID	Integer
pud	Pick up date	String formatted as: 'YYYY-MM-DD'
put	Pick up time	String formatted as: 'HH:MM'
ed	End delivery date	String formatted as: 'YYYY-MM-DD'
et	End delivery time	String formatted as: 'HH:MM'
sou	Source	Integer
des	Destination	Integer
rc	Request cost	Integer
cap	Capacity	Integer
sta	Status	String. Base value when submitting a request: 'unscheduled'
aAGV	assignAGV	Integer. Base value when submitting a request: None

Table C3: All input parameters needed to invoke SQL operations

Submit request to SQL database

```
SQLEditor.sql_submitRequest(rID , pud , put , ed , et , sou , des , rc , cap , sta ,aAGV) (5)
```

This function is currently called when a user enters a request via the UI (more about UI, see Chapter C.3).

Edit Request in SQL database

<code>SQLEditor.sql_submitEdit(rID, pud, put, ed, et, sou, des, rc, cap)</code>	(6)
---	-----

In this function only the *rID* can't be empty. If any of the other input parameters are non-empty, they will be changed in the SQL database where the request ID is *rID*. Empty input parameters will be left untouched.

Selecting next request for scheduling

<code>SQLEditor.database_sort()</code>	(7)
--	-----

This function sorts the database by first its status. It should select a request which is not scheduled yet by the algorithm. Secondly it will sort the database on ascending end delivery date and time. Calling function (7) gives a request which has the earliest End delivery date and status 'unscheduled'.

Update request status

<code>SQLEditor.request_status_update(rID, sta)</code>	(8)
--	-----

Currently 3 states for requests are defined. The status is 'unscheduled' if the request is not yet run through the algorithm. If the request is already fed through the algorithm, its status is 'scheduled'. When the request is carried out, its status is 'processed'.

Note: If an own algorithm is used in the CF, function (8) should be used to update the status of the request from 'unscheduled' to 'scheduled'.

This function is also invoked by the current simulation model. If a request is completed, the simulation sends a MQTT message over the channel "communicationframework/request-Status" containing the input parameters *rID* and *rStatus*.

Deleting a request

<code>SQLEditor.sql_submitDelete(rID)</code>	(9)
--	-----

If a request is to be deleted from the SQL database function (9) can be called. The deletion of a request only needs a request ID *rID* as input. The deleted request is returned as an object of the *RequestSQL* class.

Finding a request

<code>SQLEditor.sql_find_request(rID):</code>	(10)
---	------

If a particular request has to be obtained from the SQL database, function (10) can be called. This function is used for use case 3 (see Chapter 5.2.6). It returns the queried request is returned as an object of the *RequestSQL* class.

Add timestamp

<code>SQLEditor.add_endTimestamp(rID)</code>	(11)
--	------

If a request is finished a timestamp should be added to check whether the request is completed in time. The following function takes care of that. Just like with update request status, this function is called upon when the MQTT topic “communicationframework/requestStatus” receives a message where request with *rID* and *sta* is ‘processed’. This message is initiated from the simulation model whenever a request is completed. The same *rID* is used to add a timestamp to.

C.3 User Interface: HTML Pages

All communication between the User Interface (UI) and the scheduler module is established by HTTPS connection. This requires defining methods for every possible Flask route (all possible URL’s the user could end up in). Method possibilities are GET and POST.

Input

Every page is rendered from the Flask application using the function *render_template()*. This function requires 1 input which is the template you want to render. This template should

be an HTML-page. As additional info, variables can be passed on from Flask concluding the function:

```
render_template('page.html', variablename=variablename) (12)
```

Output

The only output is regarding all user input over the forms. Every variable input by the user can be obtained in Flask using the function (13)

```
request.form.get("fieldname") (13)
```

where *"fieldname"* is the name of the input variable defined in the HTML page. All user input is verified for correctness. This is done in the Flask application in the concerning URL route. It checks whether all necessary information is submitted. The dates and times input are checked if the specific values aren't already passed. If installed correctly the CF limits the source and destination fields to the machine locations defined in prespective.

C.4 MQTT Broker

The remaining modules are the scheduler module and the digital twin. The MQTT broker doesn't need explanation as this module only serves as communication mediator between the two aforementioned remaining modules. In this subchapter the subscribed MQTT topics and the data passed on via the topics will be set out for both modules.

C.5 Digital Twin (Prespective)

The only possible communication with Prespective is over MQTT. The inputs and outputs will be defined with the help of the used MQTT-topics.

Input

The digital twin is subscribed to two topics on which it receives data.

1. Subscribed topic: "communicationframework/requests" Over this channel all requests that are processed by the algorithm will be published. The outputted request *RequestToAGVNew* of the algorithm is an object of *RequestSQL*. Before sending it over MQTT the object is reduced to *RequestToAGV* only containing the information that the Digital Twin needs.

C. INPUT/OUTPUT DATA MODEL

```
RequestToAGV = {  
  "RequestID": RequestToAGVNew.requestID ,  
  "OrderType": RequestToAGVNew.orderType ,  
  "AssignAGV": RequestToAGVNew.assignAGV ,  
  "Source": RequestToAGVNew.source ,  
  "Destination": RequestToAGVNew.destination  
}
```

This variable is converted to JSON format and received in this format by Prespective. The Digital Twin is able to read the JSON objects and store it as an object of a custom made struct *RequestStruct*.

2. Subscribed topic: “communicationframework/externalTrigger” This channel is used if a trigger event occurs from outside the Digital Twin. For instance when the SQL database doesn’t hold unscheduled requests, an AGV is available and a new request is submitted. A trigger should be send to initiate the sequence of events needed to schedule this incoming request. The MQTT message over this channel should be a simple “1” in the form of a string to trigger this sequence of events.

Output

The digital twin has 4 MQTT output topics to which it publishes messages.

1. Topic: “communicationframework/statusAgv” All AGV’s statuses will be send over this topic. They are sent in the same format as *tempstatus* as defined at the input for the scheduler algorithm. Prespective doesn’t support to send the dictionary format over MQTT. Instead it is built up the same but converted into a string. The scheduler module is able to breakdown this string to its wanted format (dictionary).
2. Topic: “communicationframework/splineDistances” This topic is only used on initiation of the framework. All distances between machines/charging stations are stored as a dictionary in the Digital Twin. There also is a key:value pair included for the amount of machines (‘machineAmount’: int *mAmount*). Before sending it to this topic, the complete dictionary is converted to a string. All key values for the distance between machine *i* and machine *j* are named: ‘*i_j*’ (*i* underscore *j*).

3. Topic: "communicationframework/triggerEvent" Whenever a trigger event occurs in the Digital Twin, a trigger message is sent over this topic. This trigger message contains an integer. If a trigger event occurs the integer is 1. Receiving the message 1 over this topic triggers the algorithm to schedule the next request. The Digital Twin can also send a 2 over this channel. If so, this means an AGV's charge is below its threshold. Perspective only sends a message over this topic when an AGV is done with an assignment. Immediately after sending 1 or 2 over this topic, a 0 is sent for resetting the variable.
4. Topic: "communicationframework/requestStatus" If an event occurs which alters the status of a particular request, this channel is used. A custom class requestStatus in the digital twin is sent consisting of:

```
public class requestStatus
{
    public int rID;
    public string RequestStatus;
}
```

The message is read by the scheduler module as a dictionary with keys 'rID' and 'RequestStatus' with values respectively an integer and a string. This topic is used when an AGV is done with an assignment to edit the status of request *rID* in the SQL database to 'processed'.

C.6 Scheduler Module (Flask application)

The flask application is the center piece of the framework. Almost all communication is passing through or is initiated by the flask application. First the MQTT communication is set out. All subscribed topics and topics to which flask publishes to will be explained containing the right data formats. Later in this subchapter the remaining communication is documented.

Input (over MQTT)

All output MQTT topics of the Digital Twin are the topics to which the flask application is subscribed to. This also means that the data formats sent over these topics is already handled. A short description per topic is given.

1. Subscribed topic: “communicationframework/statusAgv” All needed real time information regarding the AGV’s statuses is received over this topic. This data is stored in the variable *tempstatus*. It is obtained by the Scheduler Module and used for the algorithm. See data format at C.1.
2. Subscribed topic: “communicationframework/splineDistances” As said before, this topic is only used once on initiation of the framework. The distance between all machines are received as a string and immediately stored as a dictionary (see Chapter C.5). This dictionary is currently used in the provided algorithm. Upon receiving this dictionary, it is stored into a file *MachineDistances.pkl* using the function *pickle.dump(dictname,filename)*. For using this dictionary, *pickle.load(dictname,filename)* is called and stored in a variable.
3. Subscribed topic: “communicationframework/triggerEvent” If the trigger message over this topic contains the integer 1, the algorithm is triggered to schedule a request. If the trigger message contains a 2, the algorithm is triggered to create a charging request. For both cases multiple requests could be scheduled depending on the availability/needs of the AGV’s.
4. Subscribed topic: “communicationframework/requestStatus” See C.5 for more information.

Output (over MQTT)

All input MQTT topics of the Digital Twin are the topics to which Flask publishes messages to.

1. Topic: “communicationframework/requests” All scheduled request are published by the Scheduler Module over this topic. Data format is RequestToAGV converted into a JSON object before publishing. See Chapter C.5 for the particulars of the data format.
2. Topic: “communicationframework/externalTrigger” In case of a trigger event happening outside of the digital twin, a string containing “1” is published on this topic. This triggers the sequence of events for scheduling a request. Now only used when a new request is submitted by a user.

For every subscribed channel in the Scheduler Module, a callback function is defined. This means when a message is received over a subscribed channel, this function is instantly called. They are defined in the main script *SchedulerModule.py*. Beneath the form is defined with the example of the 'communicationframework/statusAgv' topic. The attributes *client*, *userdata* and *message* are sent with the MQTT message.

```
@mqtt.on_topic('communicationframework/statusAgv')
def handle_mytopicAGV(client, userdata, message):
```

All communication over MQTT is covered. The remaining communication is set out below. It is already covered before but from the external modules' viewpoint. The remaining communication is defined by referring to these external viewpoints as it coincides with the communication as seen from the Flask module.

All communication with the SQL database is done with the functions covered at C.2.

All communication with rendered HTML pages is covered at C.3.

D Prespective Scripts

In Chapter 5.1.4 the overall structure of the Prespective software was set out. It consists mainly of components and scripts. Some scripts were standard from the software but others had to be reformed drastically. In this chapter an elaboration is done on these scripts. The software is built on reading script in C. All script will have the *.cs* extension, standard for C.

First the standardized script provided by Prespective will be explained. These scripts only needed some parameter adjustments. In Prespective these scripts are loaded as components.

In Figure 7 the structure tree of the components can be seen.

D.1 Standard Prespective Scripts

For each script a short description is given. Additionally, the adapted parameters are stated. The concerning standardized scripts are:

- DES Controller
- Pre Logic Simulator
- DSpline
- DESCue

D.1.1 DES Controller and Pre Logic Simulator

As seen in Figure 7 the component *DESController* is the parent component that is armed with the scripts *DES Controller* and *Pre Logic Controller*. These scripts are focused on the simulation attributes of the Digital Twin.

The overall simulation attributes changable are depicted in Figure D1. This window is called the inspector view in Prespective. It is a UI window for scripts. The parameters visible are already declared in the script and adjustable from this window. The only parameter touched is the *Time Multiplier* which speeds up the simulation speed.

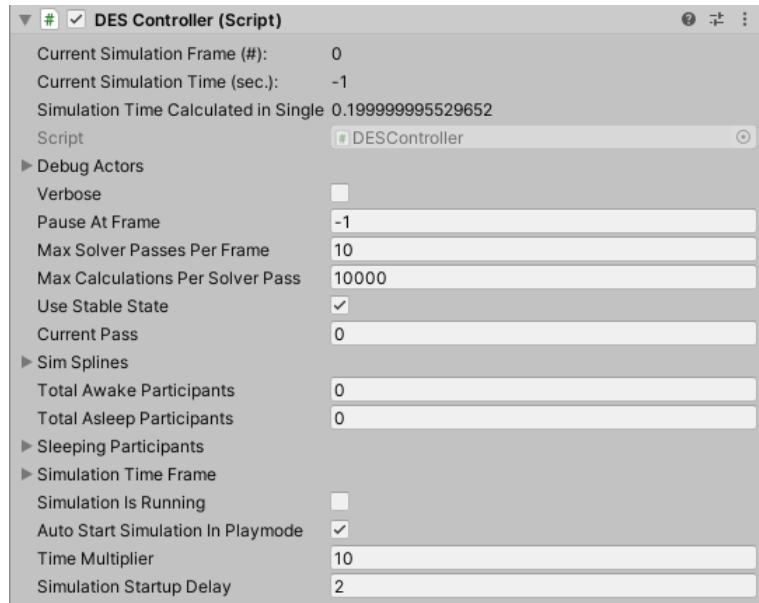


Figure D1: Component parameter window for *DES Controller*.

The *Pre Logic Simulator* is used for initialization of the MQTT adapter in Prespective. The parameter window is shown in Figure D2. In the subwindow Adapter Settings the main features of the adapter are declared. They encompass the communication protocol (set to MQTT), the IP address and port where the MQTT broker is located (locally hosted) and the QoS level set to Exactly Once (equal to QoS = 2, see Chapter 2).

Important for this script is to set the component containing the script *mqttScript.cs*. It is now attached to an object called communicationframework. This object doesn't have a visual representation in the Digital Twin but is of great value for the MQTT communication. The object communicationframework should always be assigned as Logic Component. More information about *mqttScript.cs* can be found in Chapter D.2.

Note: at the bottom of the *Pre Logic Simulator*'s parameter window a button called *Export Policy* is displayed. Whenever a change is made to *Pre Logic Simulator* or *mqttScript.cs*, this button has to be pressed. It compiles a policy regarding the MQTT communication protocol.

D.1.2 DESCue and DSpline

These scripts are imported with the component when inserted. These components are inserted when designing a layout. Designing a selfmade layout is explained in Chapter G.

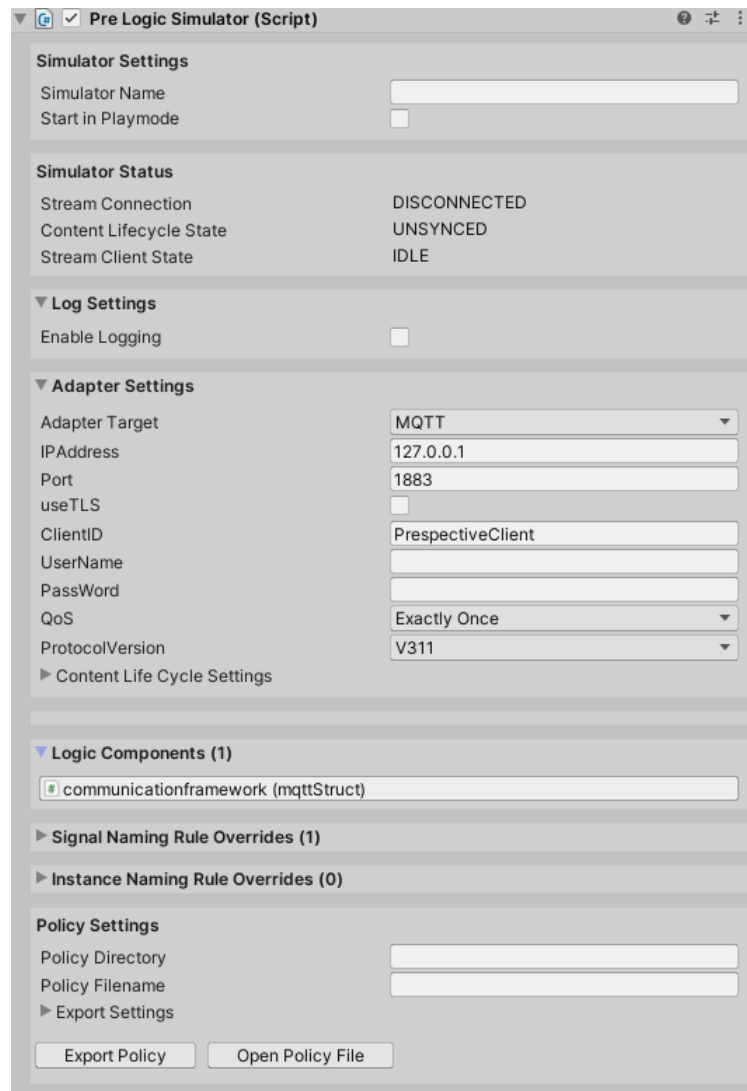


Figure D2: Component parameter window for *Pre Logic Simulator*.

D.2 Adapted Prespective Scripts

Like mentioned before, in Prespective scripts are adapted to fit the requirements. The following scripts were adapted:

- *AGVManagerInstructor.cs*
- *AGVActor.cs*
- *mqttStruct.cs*
- *getDistanceMatrix.cs*

D.2.1 AGVManagerInstructor.cs

As mentioned in Chapter 5.1.4 the *Instructor* is the replacement of the real life Fleet Management System (FMS). It controls all *Actors* (AGVs). The *Instructor* script has 5 main functionalities:

1. Resolve AGV Assignment: this function is converting the request's details. It for instance checks which machine location matches the source and destination and which AGV is to be used for a specific request.
2. Assign AGV to Machine: the converted information of the first functionality is given to the concerning AGV.
3. Create request sequence: the data is broken down into understandable steps. For a transport request it says: first go to the source machine, wait for 3 seconds and then go the destination machine. For a charging request it states: go to the destination machine and then wait until the charge reached the upper threshold.
4. On AGV done with assignment: this function specifies exactly what should be done after an AGV is done with its assigned request.
5. Distance matrix struct: only used on initiation of the CF. This struct is to be filled in by the function defined at Chapter D.2.4.

For more in-depth information please go through the scripts. They are provided with comments.

D.2.2 AGVActor.cs

The script for the *AGVActor* component is where all AGV's attributes are initiated and updated like velocity, charge and AGV position. It's script is built up in three parts. The first

part is the definition of all used variables. The initial values are declared in this part as well.

The second part is a function called *OnSimulationStart*. As the name suggests, some calculations are done at the start of the simulation.

The last part is for updating variables using function *OnSimulationFrameUpdate*. The parameter currently updated per frame are AGV's charge, position, status and location. Don't mistake location for position. The position is the real position in cartesian coordinates. The location denotes at which machine location the AGV currently is at. If transporting this attribute is -1 .

For every AGV this script is currently the same. For differences in AGV attributes, this script can easily be adapted.

D.2.3 mqttStruct.cs

All information for the communication over MQTT is defined. This script is inseparable connected to the component *communicationframework* as depicted in Figure 7. The naming of this component is of great importance. The name is the first part of all MQTT topic names. Its structure looks like: *communicationframework/....*. The part after slash is defined in the script *mqttStruct.cs*. The main functionalities are set out below.

- Signal definition: For each in- or output regarding to MQTT is defined. Its name, direction (in- or output) and supported data format are defined. For instance one channel is named "requests" which brings the eventual topic name to:
"*communicationframework/requests*".
- For every MQTT-channel that is coming into Prespective (**note:** defined as OUTPUT in *mqttStruct.cs*) a callback function is defined. In this way the Digital Twin knows what to do upon receiving a message on the specified topic. Do note that defining the callback is the equivalence of subscribing to a specific topic.
- All functions regarding to publishing information to a specific topic are specified in *mqttStruct.cs*. Currently the AGV's status, triggers, request updates and sending machine distances are defined as functions.
- MQTT data structs: for each in- or outgoing topic a struct is set up to store the information. With this technique Prespective can assign ingoing data to an object of

one of these structs. It also formats outgoing structs which can be converted by the Scheduler Module.

D.2.4 getDistanceMatrix.cs

The function for calculating the distances between all machines stations including charging stations is defined in this script. This script is called upon in the *OnSimulationFrameUpdate* function in script *AGVManagerInstructor.cs* when the simulation is in its first frame. The structure of the distance matrix is defined in this script is as well (see Chapter D.2.1).

E Scheduler Module's Scripts

In this chapter all used scripts in the Scheduler Module will be described on their behaviour and usage. A visual notation can be seen in Figure E1. Each script is provided with comments explaining certain features.

E.1 SchedulerModule.py

In order to fire up the CF, this script is to be executed. It serves as the central script of the *Scheduler Module* component. The main functionalities of this script are:

- From this central script the Flask application and the MQTT client are initiated. All configuration regarding to these features are done in the first lines of the script.
- With initializing Flask, it also means the URL scheme (Flask routes) for the UI is defined in this script (see Chapter 5.1.1). For each route the possible returned HTML pages are defined and its behaviour upon reaching a specific route.
- Presence of functions regarding to MQTT. There are specific function on what to do when the Flask MQTT Client is connected and when a message is received on a specific topic. In these functions specific behaviour is defined when one of these events occur. For more information see Chapter C.6.

E.2 SQLEditor.py

This script is used to perform operations on the SQL database. For all functions with explanation see Chapter C.2.

E.3 Algorithm.py

In order to fully test the CF a simple algorithm is implemented. The algorithm checks for a specific request which AGV with sufficient charge is nearest by. For specifics on how this is executed, check the comments in the script. The required usage of this algorithm can be found in Chapter C.1.

E.4 Models.py

In this script the custom made class *RequestSQL* is defined. With the help of ORM this custom made class is compatible for usage in Python and in the SQL database. For more information of how this works see Chapter C.2. .

E.5 Initialization Files

The concerning files that need to be run on initialization:

- *Scheduler_initialization.py*: used for initializing all charging stations. They need to be seperated from machine stations.
- *updateMachineDistances.py*: when the Digital Twin is started up for the first time, it will send the distances between the machines and charging stations. After the first start, this script has to be executed. It will store the distances between all machines in a dictionary which is now used by *Algorithm.py*. For more information about the data format see Chapter C.5.
- *csvToSql.py*: This particular script can be used when the user wants to use a pre-filled database. These can be delivered from a CSV-file (Comma Seperated Values). This an excel extension. A test file is provided called *requests.csv*. The name of the CSV-file to be inserted needs to be specified in this script. Running it will insert all requests in the CSV-file to the SQL database.

E. SCHEDULER MODULE'S SCRIPTS

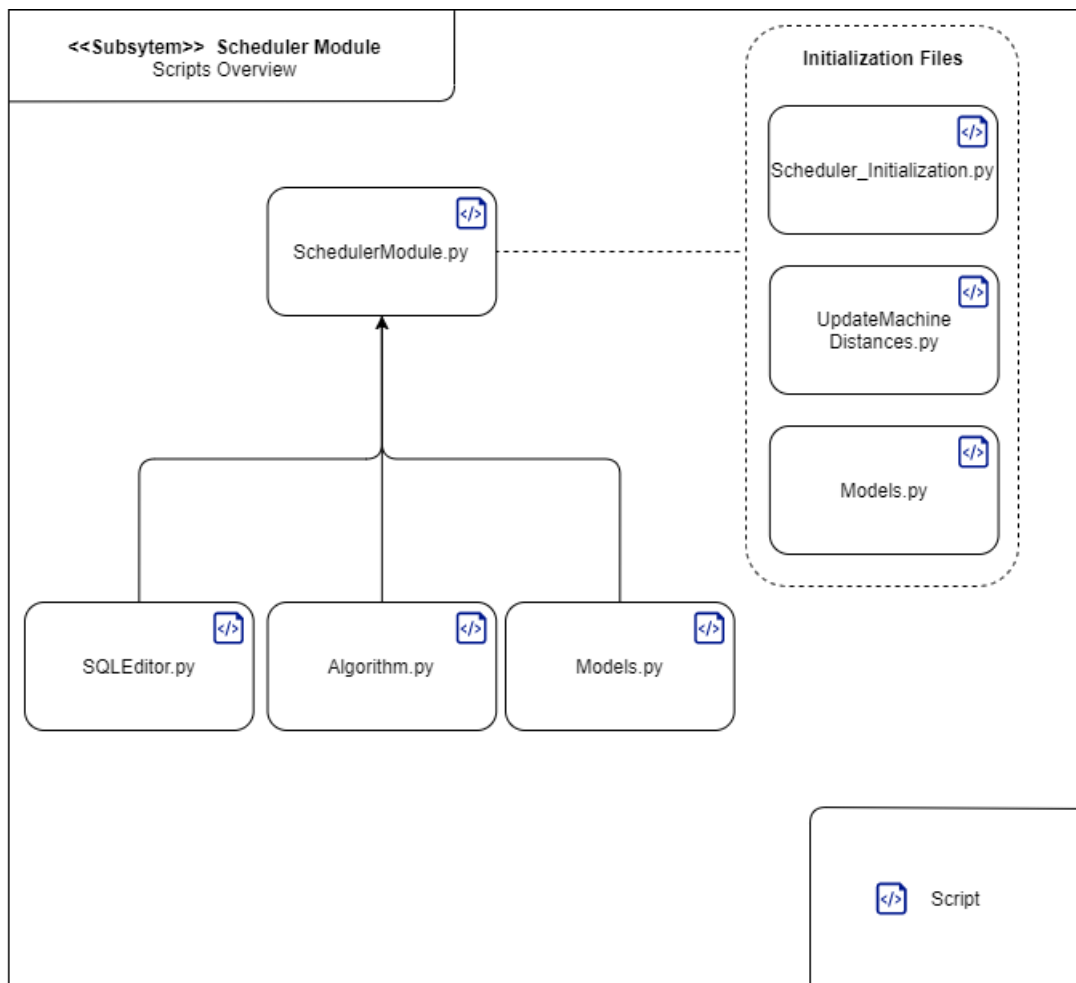


Figure E1: Script structure of the Scheduler Module

F Installation Guide

F.1 Introduction

In order to fully use the CF a step wise explanation on how to install each piece is set out in this chapter. Each component of the CF is handled separately. This guide is focused on using the default configurations applied to the BIC case explained in Chapter 1. For adaptations like using another algorithm or Digital Twin layout, the User Manual is recommended (see Chapter G).

The components covered in this guide are:

- Prespective Digital Twin software for Unity
- Visual Studio Code
- PostgreSQL
- Mosquitto MQTT Broker

F.2 Purpose of Document

This guide serves the installation proces of the MQTT-based communication framework including a Digital Twin. It brings together all applicable components and how to install them.

F.3 Prespective Digital Twin Software

The software of Prespective is still being developed further every day. Do note that later versions of Prespective can interfere with functionalities set in earlier versions. This means there is no guarantee if a later version of Prespective is installed. The version used in this CF is 2020.1.58.3.2.

This software is built upon a real time development platform called Unity. For the full installation follow this link: [Prespective Getting Started - Installing Unity Prespective](#). It is a clear step-by-step video for installing both Unity and Prespective.

F. INSTALLATION GUIDE

F.4 Visual Studio Code

The setup of the Scheduler Module including Flask is done on the platform Visual Studio Code. The current version used is 1.51.1. Make sure you also have the Python extension. This is available in the marketplace in VS Code (see Figure F1).

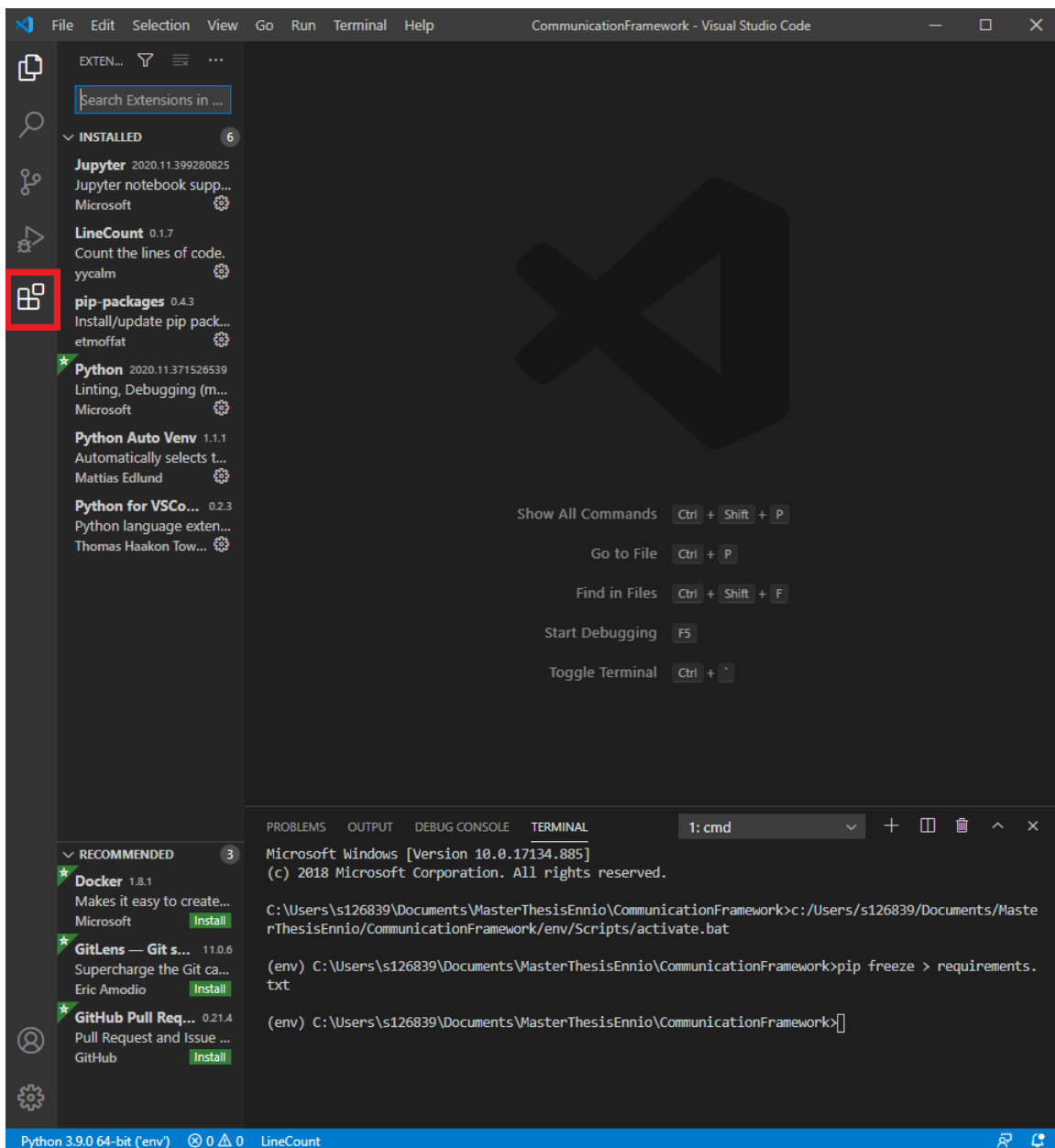


Figure F1: VS Code Marketplace in red rectangle

The next step is to check whether you have installed the *pip* installer. It should already be installed when installing VS Code. If not, download it from the market place.

In order to install all required packages, a virtual environment has to be created and activated. VS Code documentation on Python explains how to do this: VS Code virtual environments.

VS Code is almost ready to install the required packages for the CF. First use the command:

```
pip install virtualenvwrapper-win
```

In order to install all required packages use the command:

```
pip install -r requirements.txt
```

This text file is provided in the CF's software package. Check if all packages are installed correctly using *pip list*.

F.5 PostgreSQL

Downloading the PostgreSQL database can be done via their website:

<https://www.postgresql.org/download/>. The current version used is 13.0. Do make sure when installed that the windows path for Postgres tools is set correctly. Specifically this means adding the bin directory to the path system variable.

The next step is to connect the Scheduler Module with the SQL database. In the script *SchedulerModule.py* the right URL has to be provided. After the importing of packages the Flask app configurations are defined. For the connection, the following variable in *SchedulerModule.py* has to be adapted accordingly:

```
app.config["SQLALCHEMY_DATABASE_URI"] = "postgresql+psycopg2://{username}:{password}@localhost/{dbname}"
```

In the curly brackets the PostgreSQL username and password need to be entered. This also has to be done in the script *SQL_Initialization.py*.

Note: the variable *dbname* is the overall database name which consists of tables. The table is where the actual requests are stored.

Now everything is set regarding to the SQL Database. When the overall CF is executed for the first time, the database table *requests* is initialized. This database will be used for storing every request. For pre-filling the database see Chapter G.

F.6 Mosquitto MQTT Broker

As already mentioned in Chapter 5.1.5, a locally hosted MQTT broker is used. The open source broker called Mosquitto is used for this purpose. This application is installed as a Windows Service. It is running on the background without a UI.

It can be downloaded via their website. The broker is connected to using a MQTT broker URL (127.0.0.1 for locally hosted) and a port (1883 set as default port). The two clients connecting to it are the Scheduler Module and Prespective. They are both already set to the right broker URL and port (127.0.0.1 and respectively 1883).

G User Manual

This manual is for users who want to use the CF for analysis and/or optimization of similar like cases. First use the Installation Guide in Chapter F to prepare the CF for its first use. The default case (BIC case) is set out on how to use it. Later on in this chapter, a description is given on initializing your own requests dataset, how to implement a custom made algorithm and how to generate your own factory layout.

G.1 Monitoring Processes

G.1.1 MQTT.fx

For getting to know the MQTT message sequencing an additional software could be helpful. The MQTT.fx application can be used to check all MQTT traffic. This software is designed to initialize an extra client to the set up MQTT Mosquitto broker. Just as with the Scheduler Module, a connection can be made to the broker using a MQTT broker URL (locally hosted: 127.0.0.1 and default port 1883).

MQTT.fx lets the user subscribe/subscribe directly to any channel. All messages over subscribed topics will be shown as can be seen in Figure G1.

G.1.2 Database insight

In order to get more insight in the process it is highly recommended to keep track of the database while running the CF. In order to do so open a command prompt. Use the following command to open up the SQL shell:

```
psql -U {username}
```

After this first command, the command prompt asks for your password for the locally hosted SQL database. If everything is set up according to the Installation Guide you are ready to start up the CF for the first time. If accepted, it suppose to show the the window depicted in G2.

To see the content of the *request* table, use the following commmand:

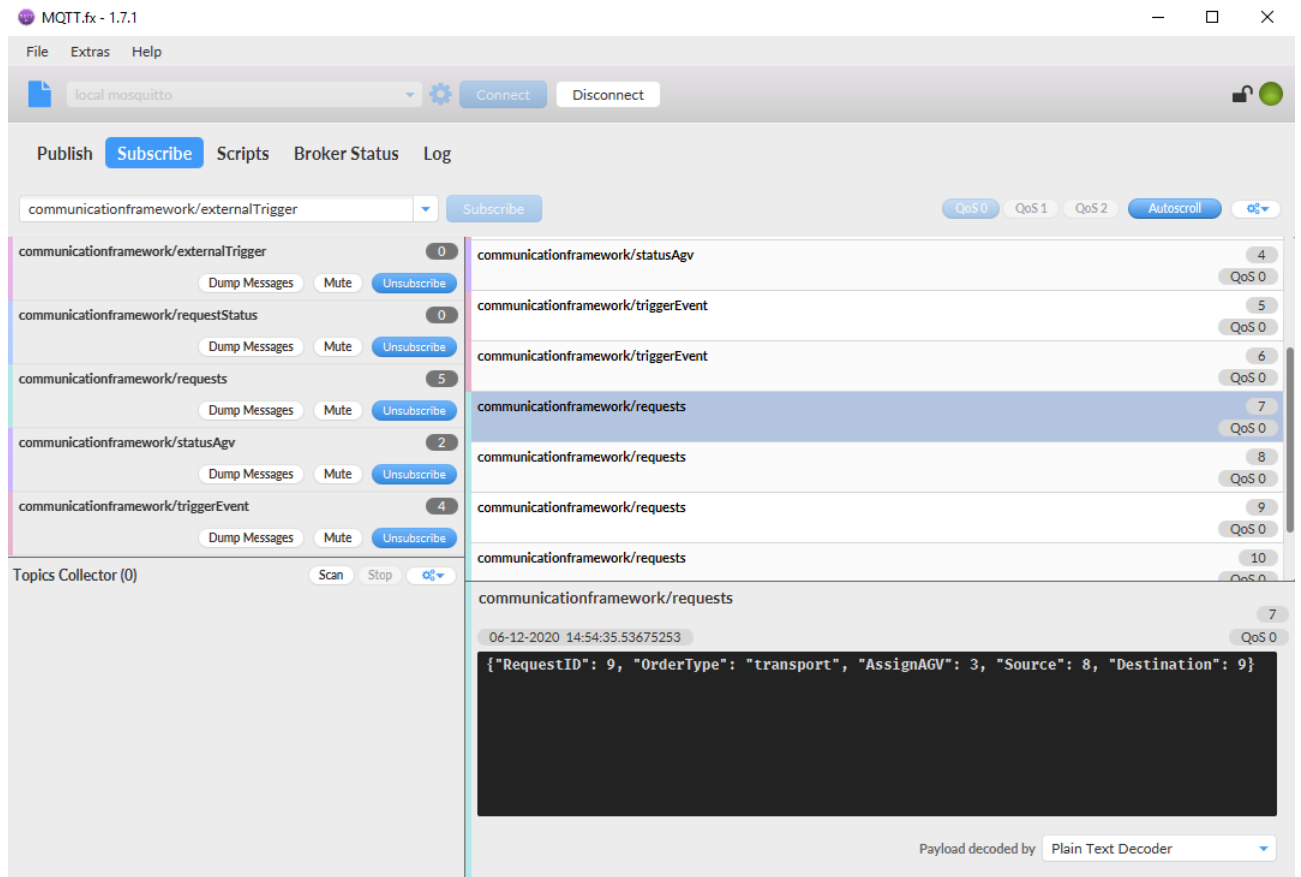


Figure G1: Screenshot of MQTT.fx’s User Interface

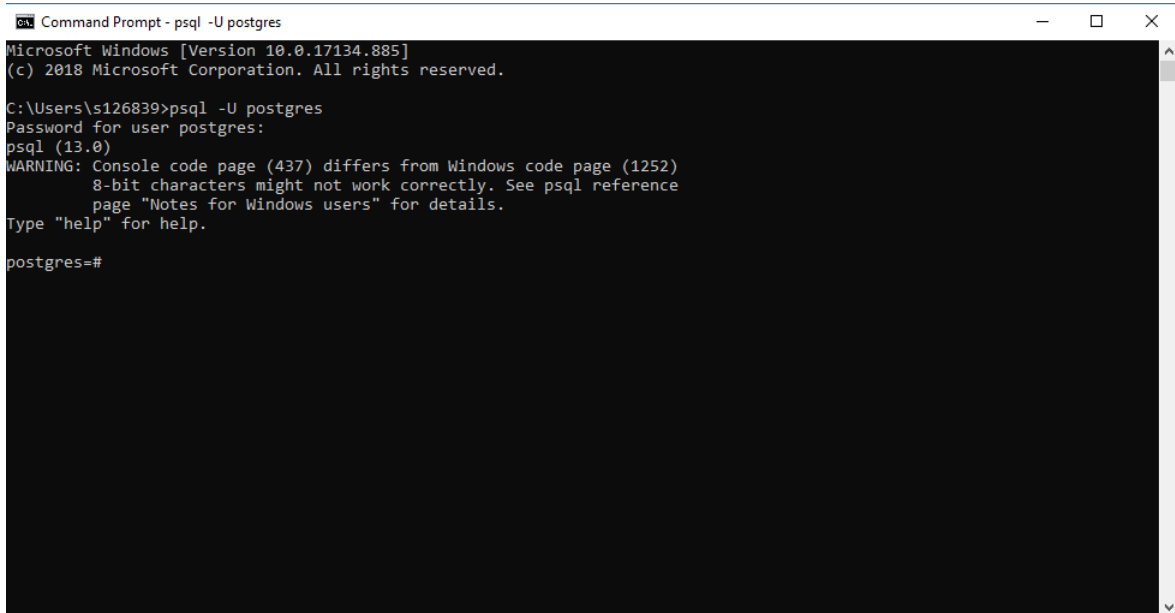
```
SELECT * FROM requests;
```

Do note that this table doesn’t exist before the first run. After the first run, the *requests* table is initialized. This command can be used during runtime or after running the CF.

G.2 Getting Started

When starting up the CF there always exist a sequence in which module to start first. Always first start the Scheduler Module by running the script *SchedulerModule.py*. On a successful start up, VS Code should give the window depicted in G3.

Note: If it is the first time this module is executed, stop the module after a successful first run. The SQL database should now be initialized and ready for use. Now try the commands

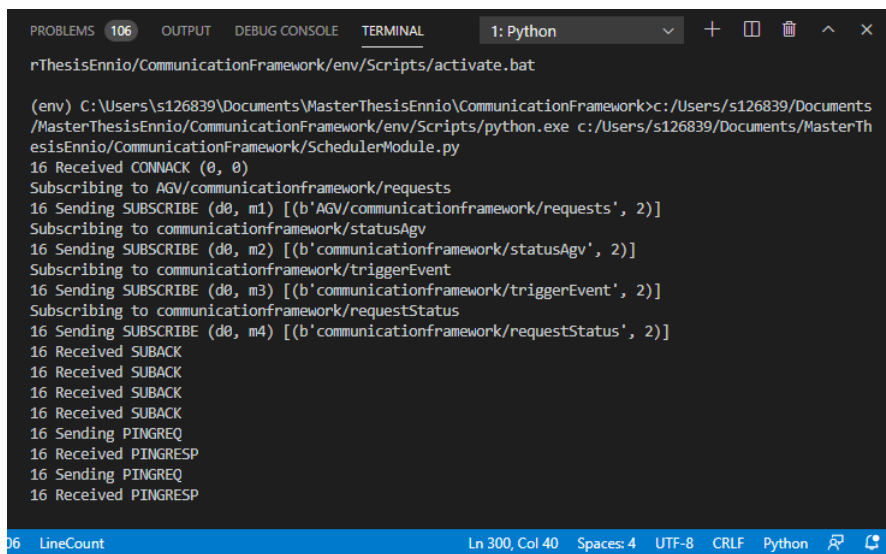


```
Command Prompt - psql -U postgres
Microsoft Windows [Version 10.0.17134.885]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\s126839>psql -U postgres
Password for user postgres:
psql (13.0)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=#
```

Figure G2: Screenshot of successful SQL verification



```
PROBLEMS 106 OUTPUT DEBUG CONSOLE TERMINAL 1: Python
rThesisEnnio/CommunicationFramework/env/Scripts/activate.bat

(env) C:\Users\s126839\Documents\MasterThesisEnnio\CommunicationFramework>c:/Users/s126839/Documents/MasterThesisEnnio/CommunicationFramework/env/Scripts/python.exe c:/Users/s126839/Documents/MasterThesisEnnio/CommunicationFramework/SchedulerModule.py
16 Received CONNACK (0, 0)
Subscribing to AGV/communicationframework/requests
16 Sending SUBSCRIBE (d0, m1) [(b'AGV/communicationframework/requests', 2)]
Subscribing to communicationframework/statusAgv
16 Sending SUBSCRIBE (d0, m2) [(b'communicationframework/statusAgv', 2)]
Subscribing to communicationframework/triggerEvent
16 Sending SUBSCRIBE (d0, m3) [(b'communicationframework/triggerEvent', 2)]
Subscribing to communicationframework/requestStatus
16 Sending SUBSCRIBE (d0, m4) [(b'communicationframework/requestStatus', 2)]
16 Received SUBACK
16 Received SUBACK
16 Received SUBACK
16 Received SUBACK
16 Sending PINGREQ
16 Received PINGRESP
16 Sending PINGREQ
16 Received PINGRESP
```

Figure G3: Screenshot of a successful start of script *SchedulerModule.py*.

explained in Chapter G.1.2. This should give you the window as seen in Figure G4.

In the second run, again the Scheduler Module should be enabled first followed up by starting the Prespective Simulation using the start button at the top of the screen. It could take a

G. USER MANUAL



```
Command Prompt - psql -U postgres
Microsoft Windows [Version 10.0.17134.885]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\s126839>psql -U postgres
Password for user postgres:
psql (13.0)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

postgres=# SELECT * FROM requests;
 requestID | orderType | pickupDate | pickupTime | endDeliveryDate | endDeliveryTime | source | destination | requestCost | capacity | status | assignAGV | EndTimestamp
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(0 rows)

postgres=#
```

Figure G4: Screenshot of successful SQL initialization.

few seconds for the simulation to start as it has to initialize the MQTT protocol on every run. After a successful run without any errors in the Console of Prespective (see Figure G5) stop both Prespective and the Scheduler Module. The Scheduler Module should have received the distances between all machine stations.

Note: The Scheduler Module probably gives an error (on the second run) as it wants to start scheduling requests without knowing all distances between the machines. When both Prespective and the Scheduler Module are stopped after the second run, open and start the script *UpdateMachineDistances.py*. Now all distances between machines are known and stored to use in the CF.

There is one action left. In the script *SchedulerModule.py* a few code adjustments have to be made:

- Line 34-36 are stating the importing of packages. Uncomment these lines (see Figure G6.)
- Line 69 has to be uncommented (see Figure G7).
- In lines 318-319 the subscription for the topic "communicationframework/splineDistances" is stated. Comment these lines (see Figure G8)

Everything is set for running the default BIC case.

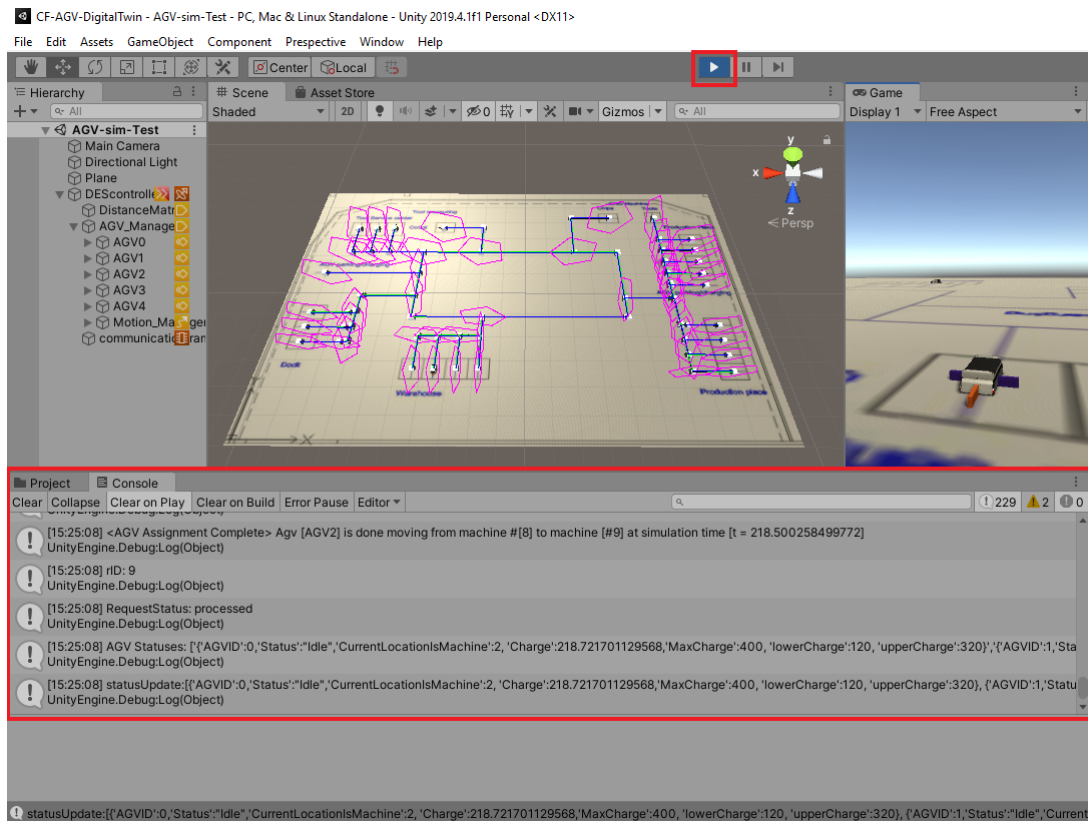


Figure G5: Perspective UI with in the top red square: play button. Lower red square: console.

```

32 #Uncomment these imports after initialization
33 #-----
34 # import Algorithm
35 # import InitializationFiles.Scheduler_Initialization
36 # from InitializationFiles.Scheduler_Initialization import chargeStations, amountChargingStations,
37 #-----

```

Figure G6: Lines 34-36 have to be uncommented.

```

68 from models import * # this import has to be done after the SQLAlchemy object initialization since
69 # import SQLAlchemy

```

Figure G7: Line 69 have to be uncommented.

```

305 @mqtt.on_connect()
306 def handle_connect(client, userdata, flags, rc):
307     print('Subscribing to AGV/communicationframework/requests')
308     mqtt.subscribe('AGV/communicationframework/requests', qos=2)
309     print('Subscribing to communicationframework/statusAgv')
310     mqtt.subscribe('communicationframework/statusAgv', qos=2) #topic made by Prespective
311     print('Subscribing to communicationframework/triggerEvent')
312     mqtt.subscribe('communicationframework/triggerEvent', qos=2)
313     print('Subscribing to communicationframework/requestStatus')
314     mqtt.subscribe('communicationframework/requestStatus', qos=2)
315
316     #Comment these lines out after initialization
317     #-----
318     print('Subscribing to communicationframework/splineDistances')
319     mqtt.subscribe('communicationframework/splineDistances')
320     #-----
321

```

Figure G8: Lines 318-319 have to be commented.

G.3 Initialize CF with own request dataset

The CF can be used with an empty dataset and generating own requests using the HTML page UI. It is also possible to fill the dataset beforehand. The data can be provided in a CSV-file (excel extension). An example dataset is in the CF's software package as well called *requests.csv*.

Opening up the example dataset shows how the dataset should be delivered. On the top row all request attributes are visible (for explanation on these attributes see Chapter C.2). Each parameter is inserted in the same order as these header row is sorted. The parameters *assignAGV* and *EndTimeStamp* should be empty in the SQL database. In the file *requests.csv* they should contain a value. Its values are arbitrary.

When the database with the request table is set up correctly, a CSV-file can be entered using the script *csvToSql.py*. Two parameters have to be adapted in this script.

The variable *csvFilename* should be edited to the filename you want to convert into the SQL table. Check if your CSV-file is in the right directory (same directory as *SchedulerModule.py*). The second parameter is the variable *conn* which is an object of the connection this script tries to make with your SQL database (see below).

```
conn = psycopg2.connect("host=localhost dbname={dbname} user={username}
                        password={password}")
```

Note that the variables in between curly brackets have to be edited. They speak for themselves.

Once the parameter editing is done, run the script and your custom dataset is inserted to the requests table. The arbitrary values of *assignAGV* and *EndTimeStamp* are converted to empty fields.

G.4 Implement custom algorithm

The current algorithm holds two main functions: assigning an AGV to an inserted request and creating a charging request. The system is now build upon the names of these functions called *Algorithm.main()* and *Algorithm.chargingRequest()*. Use the same names when implementing

a custom made algorithm to prevent adapting other scripts.

Note that this software package doesn't include decision making about request and charging assignment. This is to be defined by the custom algorithm. For embedding the custom algorithm into the CF the I/O data model stated in Chapter C.1 has to be read carefully.

G.5 Generate custom factory Layout

Like mentioned before, the CF can serve as a template for similar like cases. It is possible to use the CF with another factory layout. In Prespective the objects *Plane* and *Motion Manager* need to be emptied (see Figure G5). Now the Prespective model lacks *Splines* (routes) and *Cues* (decision points). Designing your own layout can begin.

In order to design your own layout the official Prespective documentation is recommended: Custom Factory Layout Generation.

This manual explains how to design your own layout with *Splines* and *Cues*. Note that all machine stations (including charging stations) should be provided with a *Des Cue* script and the junctions of *Splines* with a *Junction Cue* script. For each *Cue* a cue instructor has to be set. This can be done in the inspector window of that particular *Cue* (see Figure G9). Click and drag the *AGV_Manager* object on the left (in the Hierarchy window) to the cue instructor field in the inspector window. This has to be done for each *Cue*.

The charging stations also have to be provided with the script *ChargingStation.cs* which can be found in the Scripts directory in the project window (left from the console window, see Figure G5).

In the *AGV_Manager* object all machine locations have to be defined. Click on the *AGV_manager* object and go to the inspector window. Go to the Machine Locations field and drag all machine objects in.

Note: the charging locations should be dragged in at the end of the list. The CF is built in such a way that it can identify the charging locations when located at the end of this field. In the Scheduler Module the script *Scheduler_initialiation.py* has to be adapted. The variable

amountChargingStations has to be set accordingly.

The layout is ready. Now the AGVs can be put in place. Drag the *AGV* objects on a spline. Upon starting the simulation the AGV will look for the closest machine location and will start from there.

The last step is to click on the *AGV_Manager* object and look in the Inspector window for the button: *Re-index Motion Web From Selection*.

More information and an in-depth explanation for generating custom factory layouts can be found in the work of Albers, M (2020, chapter 4) [13].

G.6 AGV Adjustments

The number of AGVs can be edited easily. Delete the *AGV* object if there are too many. For more AGV's just copy and paste the *AGV* object. Don't forget to rename it accordingly.

In order to edit AGV parameters, click on the *AGV* object in the Hierarchy window. In the Inspector window the adjustable parameters show up. Adjust them to your likings and save the project. This can also be done in the *AGVActor.cs* script itself.

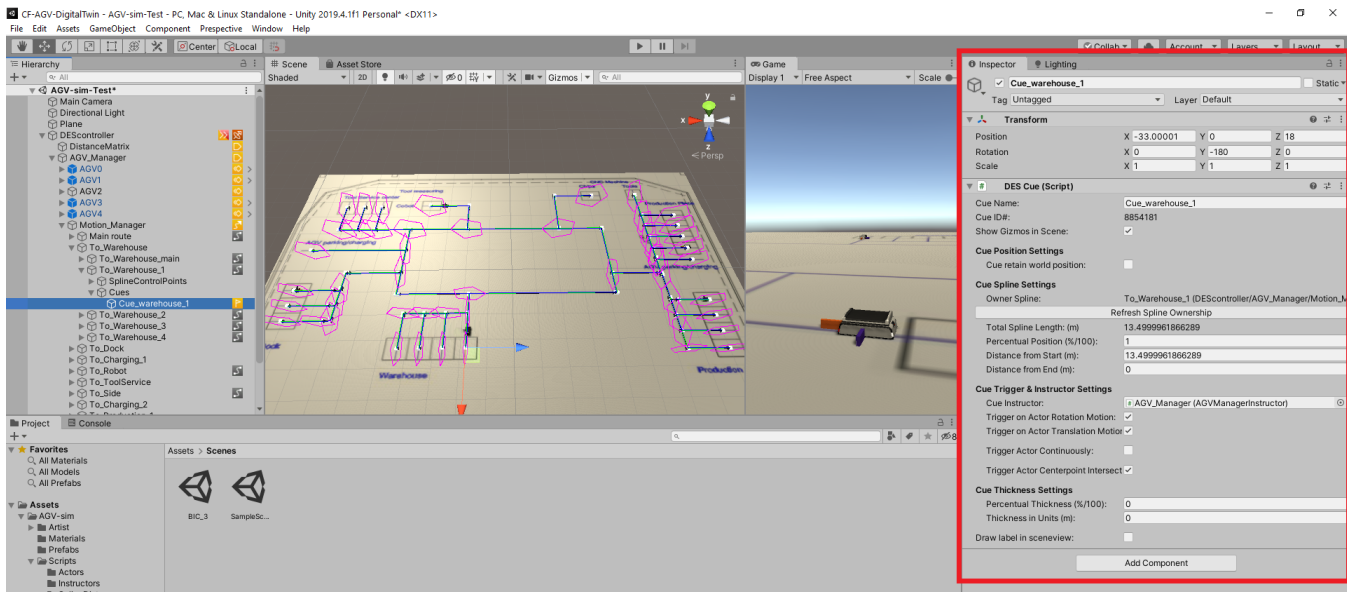


Figure G9: Inspector Window Perspective

H Scientific Code of Conduct

Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conductⁱ.

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct

Date

28-01-2021.....

Name

Ennio Thijssen.....

ID-number

0810786.....

Signature



.....

Submit the signed declaration to the student administration of your department.

ⁱ See: <http://www.tue.nl/en/university/about-the-university/integrity/scientific-integrity/>
The Netherlands Code of Conduct for Academic Practice of the VSNU can be found here also.
More information about scientific integrity is published on the websites of TU/e and VSNU

Bibliography

- [1] O. Lohse, S. Krause, C. Saal, and C. Lipp, “Real Time Reaction Concept for Cyber Physical Production Systems,” *2020 3rd International Symposium on Small-Scale Intelligent Manufacturing Systems, SIMS 2020*, pp. 8–12, 2020. 2, 5
- [2] S. Mayer, C. Arnet, D. Gankin, and C. Endisch, “Standardized framework for evaluating centralized and decentralized control systems in modular assembly systems,” *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, vol. 2019-Octob, pp. 113–119, 2019. 3, 5, 9
- [3] R. Erol, C. Sahin, A. Baykasoglu, and V. Kaplanoglu, “A multi-agent based approach to dynamic scheduling of machines and automated guided vehicles in manufacturing systems,” *Applied Soft Computing Journal*, vol. 12, no. 6, pp. 1720–1732, 2012. 5
- [4] M. Hoffmann, *Smart agents for the industry 4.0: Enabling machine learning in industrial production*. 2019. 5
- [5] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, 2017. 6, 22
- [6] D. Yi, F. Binwen, K. Xiaoming, and M. Qianqian, “Design and implementation of mobile health monitoring system based on MQTT protocol,” *Proceedings of 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference, IMCEC 2016*, pp. 1679–1682, 2017. 6

- [7] A. E. Khaled and S. Helal, “Interoperable communication framework for bridging RESTful and topic-based communication in IoT,” *Future Generation Computer Systems*, vol. 92, pp. 628–643, 2019. 6
- [8] I. Mendoza, M. Kalinowski, U. Souza, and M. Felderer, “Relating Verification and Validation Methods to Software Product Quality Characteristics: Results of an Expert Survey,” in *Lecture Notes in Business Information Processing*, vol. 338, pp. 33–44, 2019. 12, 41
- [9] F. Alhumaidan, “A Critical Analysis and Treatment of Important UML Diagrams Enhancing Modeling Power,” *Intelligent Information Management*, vol. 04, no. 05, pp. 231–237, 2012. 16, 24
- [10] J. Erasmus, *The Application of Business Process Management in Smart Manufacturing*. 1st ed., 2019. 19
- [11] J. Estdale and E. Georgiadou, “Applying the ISO/IEC 25010 Quality Models to Software Product,” *Communications in Computer and Information Science*, vol. 896, no. January, pp. 492–503, 2018. 35, 41
- [12] J. Visser and S. I. Group, “Beoordeling van productkwaliteit met ISO 25010,” 2013. 41
- [13] M. Albers, “Automated Milling Line Throughput Simulation,” tech. rep., Technical University Eindhoven, 2020. 87