Project Report

University of Southern Denmark

Faculty of Engineering

Maersk Mc-Kinney Moller Institute

4th Semester project

Course code - ST4-PRO

Project period: Feburary 1st - May 29th

Group ST01

Mathias Jeppesen Engmark	maeng20@student.sdu.dk
Oliver Heine	olhei20@student.sdu.dk
Anton Valdemar Dahlin Irvold	anirv20@student.sdu.dk
Nicklas Bruun Jensen	nickj20@student.sdu.dk
Christoffer Schurmann Krath	chkra19@student.sdu.dk
Kasper Stokholm	kasto16@student.sdu.dk

Supervisor

Drisya Alex Thumba	dath@mmmi.sdu.dk

II Title page

University of Southern Denmark

Faculty of Engineering

Maersk Mc-Kinney Moller Institute

4th Semester project software

Course code - ST4-PRO

Project period: Feburary 1st - May 29th

Authors

Mathias J. Engmark

Oliver Heine

Anton V. D. Irvold

Nicklas B. Jensen

Christoffer S. Krath

Kasper Stokholm

III Summary

At the start of the project a case from SDU I4.0 lab was received addressing an issue they encountered and wished to be solved. The issue was the ability to monitor and execute production sequences through a single software system for their drone production. The problem originates from their robots which are based on different communication protocols. Therefore, SDU I4.0 lab wishes the development of a single software system able to interface with the robots. Based on the issue at hand the following problem definition was written; "*The primary objective is to develop a component-based software system which interfaces with different assets using the communication protocols MQTT, SOAP and REST. When combined, the system can automate the production flow of drones specified by the user. Furthermore, to provide value to the assembly line through a simple UI dashboard and reporting for an end-user to interact with.*" The problem definition was further specified in Chapter 1.

The project's structure and planning was controlled through Scrum where Jira was used as a tool to visualize the planning of the project. An SRS document was composed to describe in detail how the system will be developed and how the software is intended to function. Requirements specifying the system's functionality was further developed through the SRS document and prioritized with the MoSCoW method. To further specify how the system should be developed in technical terms, diagrams from SysML was composed based on the requirements established from the SRS document.

Tests were conducted in order to verify and validate the developed project in relation to the composed requirements and the implementation of said requirements. The tests showed that the implementation of the system's requirements were mostly successful except for the requirements focused on shutting down the executing assets as this was not possible with the consigned resources. The cause of having these unfulfilled requirements stems from the delayed delivery of the technical documentation that contained the available functionality of the three assets and the technical limitations of the assets provided.

IV Preface

This report is written as part of a 10 ECTS course at the Maersk Mc-Kinney Moller Institute in conjunction with the 4th semester of Software Technology.

The report is written as documentation of a proposed software solution to SDU I4.0 lab's needs to their drone production line. The report describes the work towards the proposed solution carried out by the project group.

The report is written in a way that assumes the reader has at least the same knowledge as the authors have obtained throughout the fourth and previous semesters. The audience is mainly intended to be the supervisor and censor due to the project being a semester project that will be examined.

The project group would like to thank Drisya Alex Thumba for excellent and competent supervision throughout the project which has raised the professional aspect in relation to the development of the project. Lastly, SDU I4.0 lab shall receive great thanks for supplying a simulation and a lab tour of their drone production line.

V Table of contents

VI List of chapters

This report is structured chronologically and therefore, is meant to be read as such. Though it might be possible to read abstractions and still be able to understand the subject. It is expected that the reader has basic knowledge of the programming language Java and the specified framework Spring Boot and the communication protocols: REST, SOAP & MQTT

Chapter 1 - Introduction: Introduces the background and specifies the problem definition and stakeholders throughout the project
Chapter 2 - Software requirements specification: Describes how the system will be developed and how the software system is intended to function
Chapter 3 - Design: Explains the design choices for each layer through SysML diagrams
Chapter 4 - Implementation: Describes and explains the implementation for key use-cases
Chapter 5 - Validation & verification: Validates and verifies the functionality of the developed system, specified in chapter 2, through use-case testing and verification in UPPAAL
Chapter 6 - Discussion: Discusses the results from chapter 5 in relation to the problem definition
Chapter 7 - Conclusion: Concludes on the project and its results in relation to the problem definition
Chapter 8 - Perspectives: Takes the reader through a journey on what could have been done differently, obstacles throughout the development phase and what might be added to the system in a later development phase

VII List of figures

VIII List of tables

IX List of listings

X Editorial

This report is written as documentation of the project and its results, where each chapter in the report is divided between group members. The editorial in Table 1 does not necessarily correlate with the workload during the project in regards to development. All chapters include input from all group members.

Chapter	**Written by**
Summary	Oliver
Preface	Oliver
Introduction	Everyone
Software Requirements specification	Everyone
Design	Anton, Christoffer, Kasper, Mathias
Implementation	Nicklas, Oliver
Validation & verification	Anton, Nicklas
Discussion	Anton, Nicklas, Oliver
Conclusion	Christoffer, Kasper, Mathias
Perspectives	Kasper, Oliver

Table 1 Editorial for project contents

Chapter 1

Introduction

1.1 Background & motivation

The production line at the SDU I4.0 lab needs a software system to control and monitor the production of drones. This system consists of three main elements:

- An automated guided vehicle (AGV)
- A warehouse
- An assembly station.

Each of these technologies uses its own communication protocol - therefore the software system should be able to interface with these. This is a simulation of the real world where an assembly line can consist of assets from different manufacturers that are not immediately compatible.

Throughout the project, the group members will continuously learn about new concepts and technologies in the three courses connected to this project: *Component-based Software Engineering*, *Software Technology in Cyber-Physical Systems*, and *Design of Reliable Cyber-Physical Systems*.

1.2 Problem definition

The primary objective is to develop a component-based software system that interfaces with different assets using the communication protocols MQTT, SOAP, and REST. When combined, the system can automate the production flow of drones specified by the user. Furthermore, to provide value to the assembly line through a simple UI dashboard and reporting for an end-user to interact with.

- A simple UI to ease control and usability of the production line

- A component-based system providing interfaces for different protocols, allowing robots and their processes to communicate.
- Extensive documentation for the entire project, including but not limited to:
  - Requirement specification
  - Architectural and behavioral structure
  - Verification & Validation

1.2.1 Stakeholders

SDU I4.0 lab has been provided with three assets all of which use different communication protocols. The lab has requested a software system that can link the communication of the assets, provide one place to monitor the entire production, and for the end-user to be able to make and execute simple production sequences.

The internal product owner is a chosen group member whose main goal is to organize and structure the flow of the project. The entire group as a whole has the final decision on the prioritization of the project requirements.

Since there is no direct contact with the stakeholders during the project, the group is forced to self-analyze decisions.

1.2.2 Use and limitations

SDU I4.0 lab is the intended user of the software system. Therefore, the UI might be developed with the assumption that the user has some technical knowledge. An assumption that might degrade the level of user-friendliness compared to a non-technical person.

The system will only revolve around the following communication technologies:

- REST
- SOAP
- MQTT

That means that the system will not be compatible with other technologies without updating support. Furthermore, the system will only support the provided API methods associated with each asset. Further specifications on the APIs can be found in the *Technical Documentation* provided by the semester coordinator or in the Appendix D.

1.2.3 Concept definition

The following list contains an explanation for 'concept words' used throughout the project and will not be further specified when used.

- Asset
  - The hardware/robots that SDU I4.0 lab uses
- AGV
  - Automated guided vehicle. One of the three assets in the production line
- Technical documentation
  - The project proposal given by the semester coordinators describing what the SDU I4.0 lab is in the need of
- SPI
  - The SPI - Service Provider Interface is a specific implementation of an interface.
- MES
  - A manufacturing execution system (MES) is an information system that connects, monitors and controls manufacturing systems controls the flow of data through the system
- REST
  - REST stands for Representational State Transfer. Is a style of communication between systems on the web.
- SOAP
  - Simple Object Access Protocol - SOAP, is a messaging protocol used for exchanging structured information between web services
- MQTT
  - MQTT is a publish/subscribe network protocol.
- SRS document
  - Software requirements specification describing how the system will be developed and how the software is intended to function

Chapter 2

Software requirements specification

The software requirements specification, SRS, will describe how the system will be developed and how the software is intended to function.

2.1 Description

The background of the project and what assumptions and dependencies are necessary.

2.1.1 System description

The SDU I4.0 lab currently has a production line that produces parts for drones but all assets used in the production line uses different communication protocols. This system will provide interfaces to the production line enabling the possibility to control and monitor the production step by step. The system is made up of components that coordinate the production through timely communication with the external assets.

The system will be used by the SDU I4.0 production manager. Through a simple UI, the manager can create and execute a new production sequence for the production line and monitor relevant information from the production.

2.1.2 Assumptions and dependencies

Throughout this project, the group has made assumptions to ease the development of the system. The main assumption is that when finished, each of the blocks of the production line will still be dependent on the newly developed MES system to handle communication across production. The software system itself depends on a few technologies:

- Spring Boot Framework
- MQTT communication protocol

- SOAP communication protocol
- REST communication protocol

The software system also highly depends on the simulation of the physical robots and the reliability of these simulations.

2.2 Theory & methods

This chapter touches on some key aspects of the earlier parts of the project such as planning, project structuring, and different technologies used throughout the process.

2.2.1 Project Planning

For the planning of the project, the group decided to follow a Scrum workflow with Jira as the preferred tool to plan out sprints. The project was divided into three separate sprints each spanning roughly 3 weeks. During each of the group's work sessions, a quick Scrum meeting was held with the chosen Scrum master following up on the status of previous tasks and the future work that needed to be done. Issues in the backlog were divided evenly among the group members to get an evenly distributed workload.

2.2.2 Methods and technologies

This project has used a variety of methods and technologies, some of them were acquired through courses taught in the 4th semester. The technologies related to different courses will be described below

Component-based software engineering

The major technology used comes from the course *component-based software engineering*. We were taught different component-based frameworks, and have decided to implement our component-based architecture with the use of the Spring Boot framework. The main benefit is to ease the use of components as Spring Boot offers a native annotation-based registration of components and keeps track of the life cycle during run-time.

Design of reliable cyber-physical systems

The main method used to design of reliable cyber-physical systems is the extended version of UML, SysML. It allows for another way of describing the analysis and design process. As a tool to further verify the system and its requirements a simple model will

be developed through the use of UPPAAL and verified through the integrated verification tool in UPPAAL.

Software technology in cyber-physical systems

In software, it is important to specify how the general technology will be implemented and distributed. To accomplish that, key architectural principles from *software technology in cyber-physical systems* will be considered and used appropriately in correspondence to the defined requirements in Section 2.3. To establish a communication link between the customer and the developers a *software requirements specification document* will be performed, describing how the software will be developed and how it is intended to function.

2.3 System specification

Elicited requirements are prioritized through MoSCoW. Key requirements are further specified in a detailed requirements diagram and risks threatening the project will be listed and a mitigation composed.

2.3.1 Functional requirements

The below table, Table 2.1, specifies all functional requirements. These requirements have been prioritized in accordance with the MoSCoW-method. The functional requirements table will later be used to design the requirement diagram, which will be made using SysML.

ID	**Name**	**Description**	**Prioritization (MoSCoW)**
U01	Create a production	The user should be able to combine the functions of the assets in the production line to create a procedure based on parameters.	M
U02	Monitor production	The user should be able to see the states and parameters of the different assets in the production line.	M

U03	Execute emergency stop	The user or the system should be able to execute an emergency stop for the entire production line.	M
U04	Queue a production	The user should be able to queue a production to be executed later.	S
U05	Remove production from the queue	The user should be able to remove a production from the queue (ref. U02)	S
U06	Start an asset in the production line	The user should be able to start a specific asset in the production line after being stopped (ref. U05)	S
U07	Stop an asset in the production line	The user should be able to stop a specific asset in the production line.	S
U08	Restart an asset in the production line	The user should be able to restart a specific asset in the production line.	S
U09	Create production report	The user should be able to create a production report based on the production and its parameters details.	C
U10	Retrieve production history	The user should be able to check the information about the machine's previous activity.	C

Table 2.1 Functional requirement table

2.3.2 Non-functional requirements

Below is a table of the specified non-functional requirements, Table 2.2, these requirements are also prioritized according to MoSCoW.

ID	Name	Type	Description	Prioritization (MoSCoW)
QA01	Component-based software system	Modifiability.	The system components must be open to extension and closed to modification by using components.	M
QA02	Unit tests	Testability	The system should have unit tests to test the components.	S
QA03	Up-time of the system	Availability	The system could have an up-time of ideally $> 99\%$.	C

Table 2.2 Non-Functional requirement table

2.3.3 Detailed Requirements

In order to understand the identified requirements at a deeper level a detailed use-case description was developed on use-case U01 & U02 shown in Table 2.3 & 2.4. Use-case U01 & U02 will also be used throughout the report as examples as both are seen as critical use-cases based on their prioritizing in Table 2.1. Table 2.3 specifies the interaction taken by the user and the system in relation to the use-case U01 *Create production* and Table 2.4 described the steps required by the user and the system in order to fulfill the use-case U02 *Monitor production*

Name	U01 Create a production
Primary actor	Production-manager
Brief description	The user should be able to combine the functions of the assets in the production line to create a procedure based on parameters.
Pre-conditions	The application must be connected to the assets

Main flow	1. The **user** presses a "*Create new production sequence*" button 2. The **system** shows a new screen with an overview of the different asset-interfaces that are installed on the system 3. The **user** picks an asset from the list of installed assets 4. The **system** shows a list of the operations that the asset provides 5. The **user** picks one of the operations the asset provides 6. If the **operation** requires any parameters, then the **user** enters the input for the parameters 7. The **user** presses the "Add" button to add the operation 8. The **system** adds the operation to the production 9. If the **user** wants to add another operation Then 9a. The flow points 3 through 8 will repeat 10. The **user** presses "*Add to queue*" **«Includes»** *U04* 11. The **system** saves the new production sequence in a list of production sequences
Post-conditions	A new production sequence is saved
Cross references	U04 Queue production

Table 2.3 U01 Create a production [Detailed Requirement]

Name	U02 Monitor production
Primary actor	Production-manager
Brief description	The **user** must be able to see the states and parameters of the different assets in the production line
Main flow	1. The user presses a "*Monitor production*" tab 2. The system shows a view with the different assets, their states and their parameters, as well as the production sequence and the current operation in progress.

Table 2.4 U02 Monitor production [Detailed Requirement]

2.3.4 Requirements diagram

Throughout the project, the requirements for the software solution keep evolving. A system of keeping things neat and organized must be used. This is why a requirements

diagram is composed. The diagram helps keep the relations of the different requirements organized and easy to understand. This diagram shows how different requirements relate to one another, or if they extend each other. The diagram is mostly based on the functional diagrams specified in Table 2.1. A point worthy of mentioning is that this diagram also organizes what tests and activities satisfy which requirements. Some of the SysML notations are "extend" and "«refine»". These requirements are requirements that either further define a requirement or is a so-called "sub-requirement". Also, note that the requirement-type "DerivedReq" is a requirement that is made implicitly by another requirement. This is for example depicted in the Figure 2.1 at the QueueProduction requirement.

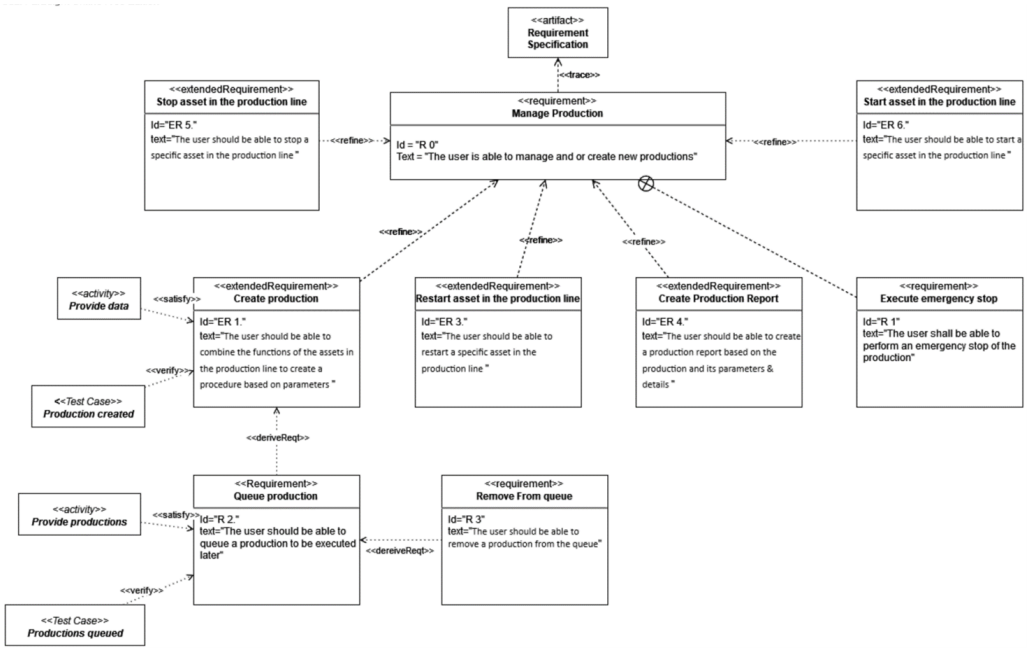


Figure 2.1 Requirement diagram

2.3.5 Risks

A project will sometimes encounter challenges capable of threatening the project's outcome. With good preparation it is possible to mitigate some of these risks.
The following table defines and describes risks that the group, project, and/or product may encounter.

ID	Name	Affects	Description	Mitigation

R01	Covid-19 backlash	Project & product, exam	A fourth wave of Covid-19 to occur in Denmark potentially closing university and preventing physical meetings.	Be sure not to meet if someone is sick. Have a backup plan in case. (Online meetings, sharing of relevant information)
R02	Code is lost	Project & product	If code is lost for some reason - by the wrong use of Git for example.	Everyone in the group is well informed about the use of Git so mistakes do not happen. Ensuring the existence of one "master" version branch on git, - that works - this can be avoided. If it so happens that the code is lost the group will try to roll back the git repository or continue working from an earlier branch.
R03	Constrained by technology	Product	If the technology used in the project is not sufficient to overcome the problems and requirements defined by the group.	Finding alternative ways to find solutions for problems or eventually changing requirements. Another migration could be to check the correlation between the type of project and the chosen technologies.

R04	'Group expert' leaves the group	Project & product	One group member who has more knowledge than the other group members in a specific field leaves the group prematurely before the project has ended.	Sharing knowledge and ideas throughout the entire process as well as keeping others up to date on everything. If the group expert leaves, the group will contact the expert to get some information. The group could also arrange for extra research in that area.
R05	Personal conflict	Project & product	Two or more group members engage in either a project-related or direct conflict that affects the productivity of the whole group. This effect is worsened if the leader of the group is engaged and unable to lead.	By keeping a friendly and open mindset in the group. By not getting too attached to any ideas. The group should elect a new leader if the leader struggles to focus because of the conflict.
R06	Delayed resources	Project & product	If the technical resources such as, but not limited to, the software simulation and technical documentation.	To ensure that the project is still worked upon with the material and knowledge available.

Table 2.5 Possible risks throughout the project

2.3.6 Risk assessment

To fully assess the risks described in section "Risks" it is necessary to consider the probability of a risk occurring as well as the severity of the risk should it happen.
The risk assessment will be calculated based on the probability times the severity:

$$Effect = Probability \bullet Severity$$

The probability and severity of each risk will be given a value between 1 & 5 - by the group – where 1 is the lowest and 5 is the highest. The column "note" will be used to explain the reasoning behind the risks assessment if found necessary.

ID	**Probability**	**Severity**	**Effect**	**Note**
R01	2	2	4	
R02	3	3	9	The severity is dependent on the code lost
R03	1	1	1	
R04	1	3	3	
R05	1	3	3	Due to the groups past the probability is considered low
R06	2	5	10	The severity is set that high due to the limitation and delays in the development of the specification is not known.

Table 2.6 Risk assessment

Chapter 3

Design

The project is developed with the use of Scrum to structure and manage the project's timeline, which also means that the defined requirements are subject to change. To mitigate the impact of such change the system is designed to be a component-based system developed with the use of the Spring Boot framework to allow for more flexibility during development and maintenance. The key benefit of using a component-based system is that each component can be developed, updated, or even scaled simultaneously and independently without impacting other components. The design is based on the earlier developed analysis of the system requirements described in the section 2.

3.1 Design

This section focuses on the different SysML diagrams developed to specify the different design choices made during development.

3.1.1 Block definition diagram

Figure 3.1 shows the initial block definition diagram of the system. The block definition diagram shows software in relation to hardware, as blocks, to get a better understanding of the structure of the system and each block's relation to one another. The Assembly Station-, AGV- and Warehouse blocks represent the physical assets and their protocols for communication. They communicate with the "Communicator" block. These blocks are part of the software as shown by the composite relationship to the MES. The assets have different states represented by an integer[1]. The presentation manager[2] oversees the UI, the report manager[3] oversees generating and getting reports and the production

[1]For a description of the states, see Appendix D
[2]The PresentationManager is called CoreInit in the implementation
[3]The ReportManager is called Persistence in the implementation

manager is in charge of creating, stopping, deleting, and queuing productions. This is not a complete representation of the components that the software system will implement, but an overview of the structure of the complete cyber-physical system. The implementation of the blocks will be further specified later on. The AGV-, Assembly Station- and Warehouse blocks are limited to a high abstraction and could be broken down further into parts that are not relevant to the system (MES) though the MES block will be broken down as seen in the internal block diagram, Section 3.1.2.

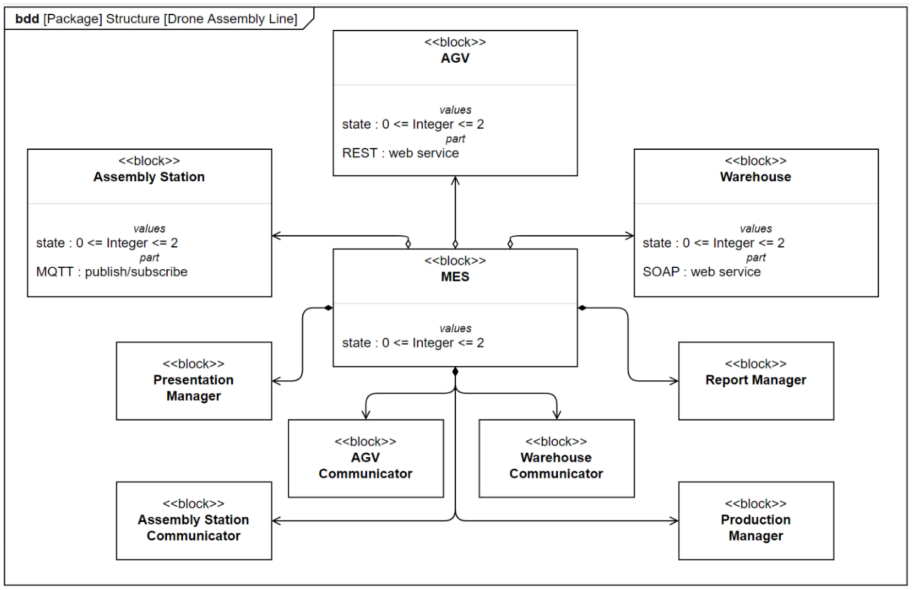


Figure 3.1 Block definition diagram

3.1.2 Internal block diagram

Figure 3.2 shows the internal block diagram, IBD, of the system. The IBD was created from the main block "MES" from Figure 3.1 to get a deeper understanding of the internal system. The diagram is split into six blocks, three managers and three communicators. The main block is the production manager. This takes input through flow ports from the AGV, Warehouse, and Assembly station - the input being the state of the machine. With these inputs, the production manager can execute several operations with the important ones being createProduction, deleteProduction & startProduction. These are executed through the interfaces of the other blocks, notably the communicators which then use the interface with the machines to execute said operations. The Report Manager provides an important operation: generateReport. This allows the system to generate a report of a specific production. The presentation manager takes input from the user through an

interface provided by the production manager and allows the user to execute supported operations. An example of how a user would interact with the system is described in Chapter 3.3.

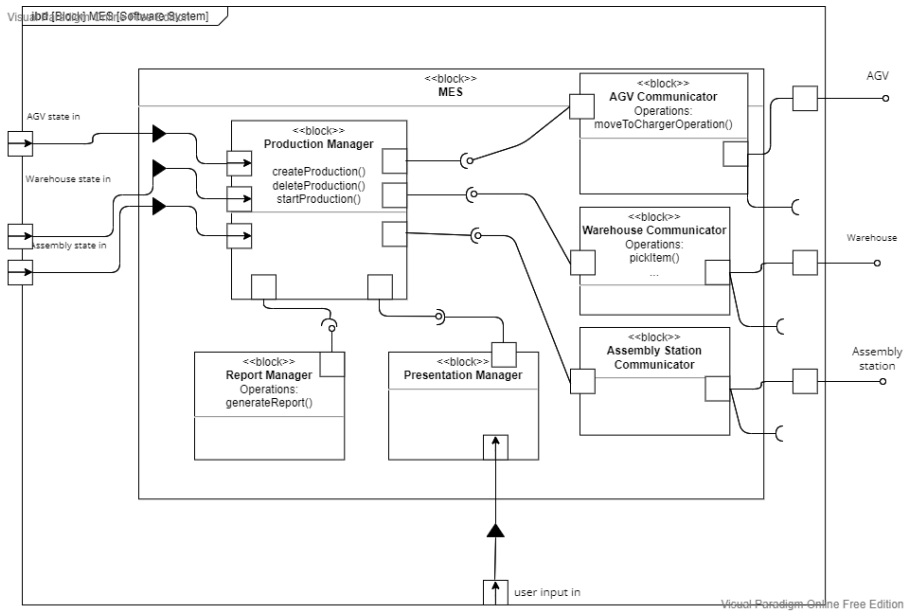


Figure 3.2 Internal block diagram

3.1.3 State machine diagram

The State machine diagram, seen in Figure 3.3, specifies the state transition the system performs in response to events. When starting the production line, it is essential that there is a connection to each asset in the system. This is depicted through timers after the first *idle* state where if the connection was not successful after 60 seconds or if the connection is declined, the system goes back to *idle*. If the initialization to all assets and therefore the connection is successfully established the system moves on. In case of state: *create production* or *Queue production*, the system responds accordingly and moves to *Operate robots*. From here, two possible responses can happen. Either the system begins a shutdown process and enters the *idle* state or it returns to the first state after initialization and waits for an event again.

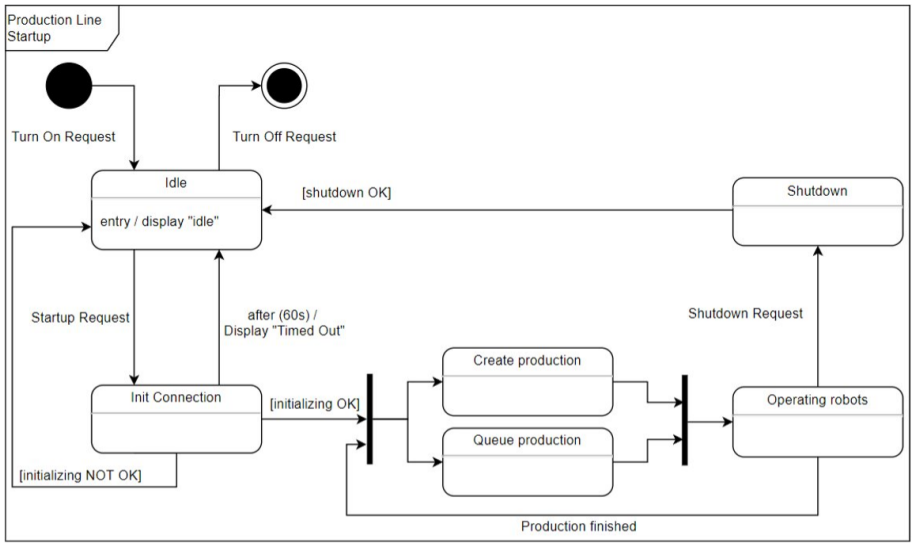


Figure 3.3 'Production System' - State machine diagram

3.1.4 Sequence diagram

To get a better understanding of the communication flow between components, a sequence diagram has been developed. One for each of the detailed requirements, U01 and U02 are seen in Table 2.3 & 2.4, due to the priority they have in the system which is based on the technical documentation and the prioritization in Table 2.1.

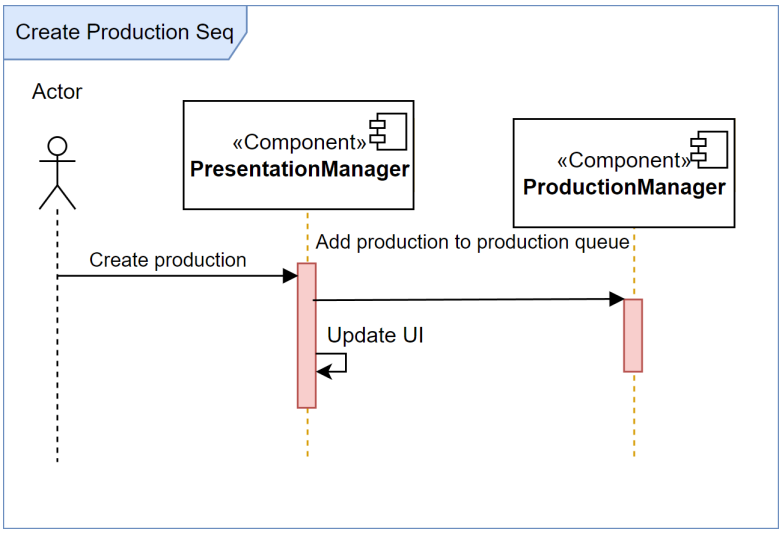


Figure 3.4 Sequence diagram of U01 - Create a production

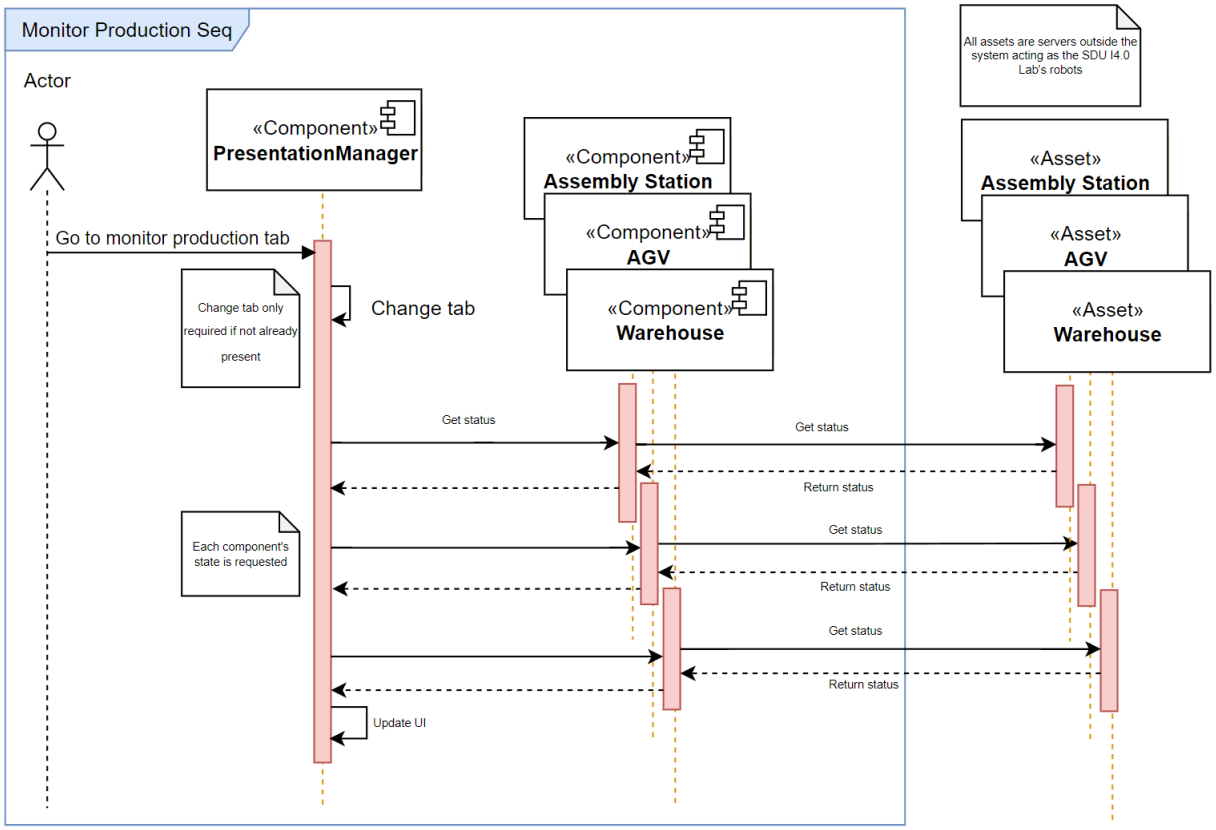


Figure 3.5 Sequence diagram of U02 - Monitor production

3.2 Design patterns

The MES acts as a distributed system that uses several different design patterns to communicate with the external asset servers. The AGV uses a Rest API from a web service to communicate and has several different endpoints to PUT and GET methods. These retrieve data and manipulate the state of the asset server. This is the request/reply pattern. In theory, this pattern allows for different clients to access the server at once. However, this is outside the requirements defined in Table 2.1. This is a relatively thick client, as the majority of the domain logic is done client-side which cannot be changed, as the server is provided along with the project proposal, however, a more thin client could make sense to provide a more sturdy interface to the asset. The Warehouse uses a SOAP protocol. This is in principle similar to the REST pattern, although it is better for the QA01, Table 2.2, as the asset becomes more open to extension and more closed to modification. This is because the SOAP methods are autogenerated from an XML file. This is not the case with REST. Lastly, the Assembly Station uses the MQTT protocol

which is an event-driven publish/subscribe pattern. In this case, both the asset and the communicator publish and subscribe to the broker topic. This makes it easy to extend the MES by adding more asset communicators, in case more clients to the program is needed. The MQTT protocol does make it more difficult to test (QA02 in Table 2.2) since the events can be difficult to reproduce because of the asynchronous methods.

3.2.1 System integration

In order for the system to act as a single connected system and to integrate to the independent assets, it is necessary to perform system integration. The system uses the horizontal integration pattern to fulfill the system integration. Figure 3.6 presents a component diagram describing the system's integration pattern. The production manager acts as the system's broker and through there other components can connect to the system.

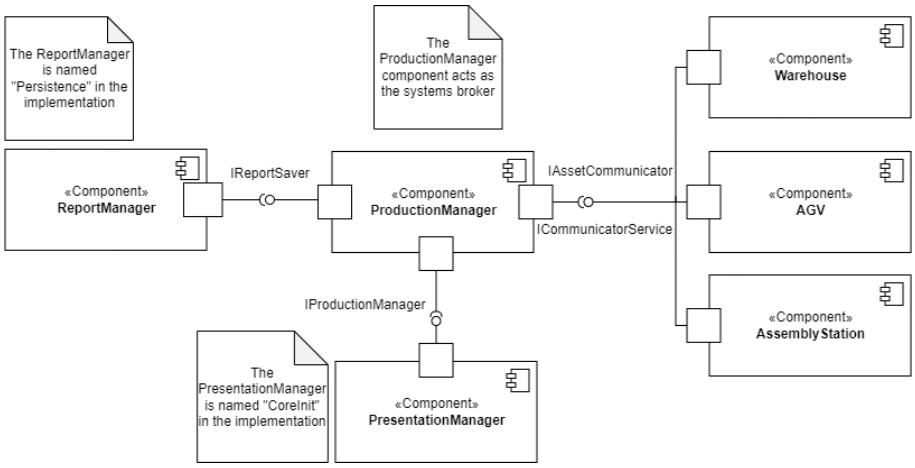


Figure 3.6 Component Diagram

3.3 UI- & UX-Design

The UI consists of three areas or tabs, one for monitoring, creating production sequences, and viewing past production history. These three tabs create a good basic functionality with the possibility to improve and optimize in the future. In this project, the interface is developed through JavaFX, a commonly used software platform designed for desktop- and web applications. As described previously in the System description in Chapter 2, the UI has to be simple yet effective. This is one of the reasons the functionality is split up. Also, the focus of the UI has not been to make the interface look pretty. The main focus has been to make it easy to navigate and minimize confusion and human errors.

It is also important to note that the end-user is expected to be familiar with the assets in the system and the UI is designed with this in mind meaning that elements of the design may not provide sufficient information for navigation and technical usage for users unfamiliar with the production line.

3.3.1 Monitor

This tab is where the operator can monitor all assets in the system and get an overview of which states each asset is currently in. Whenever an asset is active it is illuminated at the name of the asset with a green color. If not this is black. In the monitoring view, the operator also has the opportunity to start new productions. This can either be done as an entire production or a simple operation sequence. See Figure 3.7. It is also possible for the operator to show a production in more detail by highlighting one of the queues to see which operations the production is constructed of. Once an operation has been completed it is removed from the tab, and can now be seen in the Production History tab.

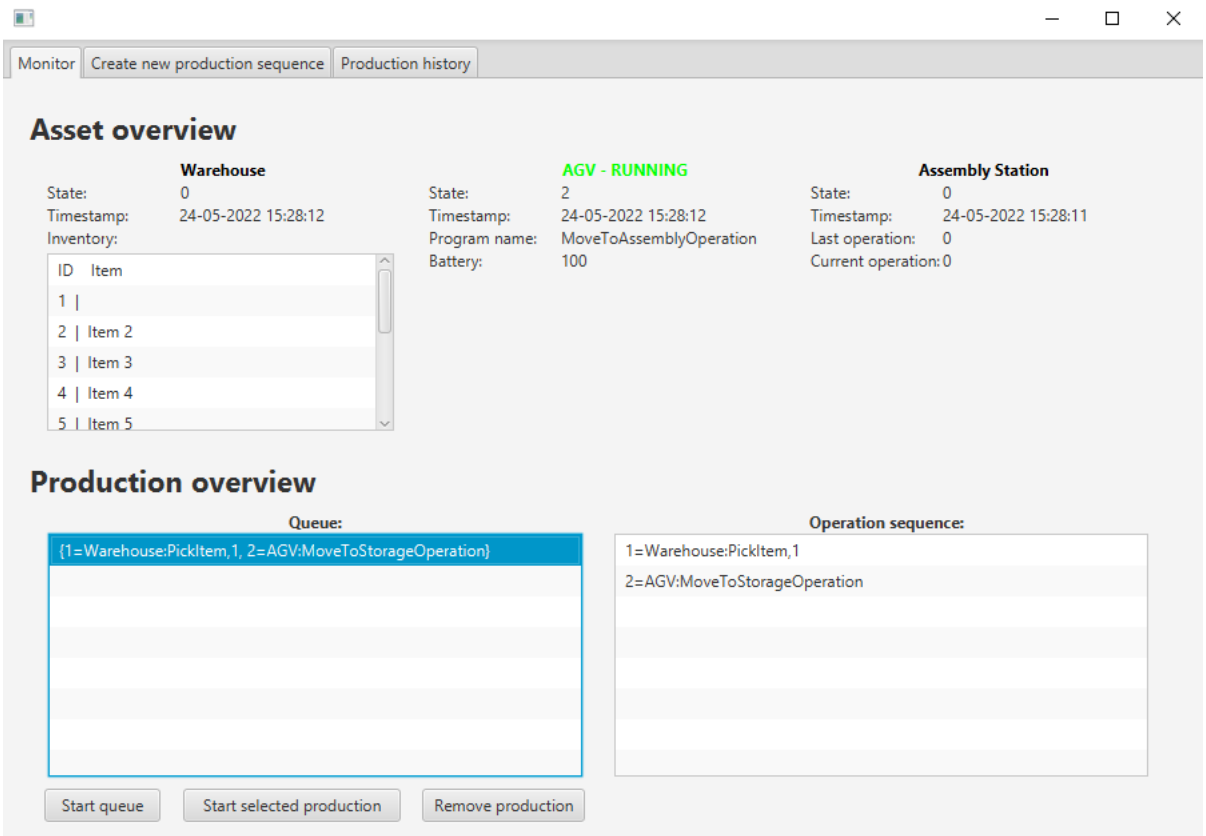


Figure 3.7 Production Monitor illustration

3.3.2 Create production sequence

In the *Create Production sequence* tab, the operator has the possibility of creating new operation sequences. An operation can be one of three types; a Warehouse-, AGV, or an Assembly station operation. Each asset has different operations of its own and the operator can, through the UI, mix and match operation sequences until satisfied. See Figure 3.8 for an illustration of the production sequence tab.

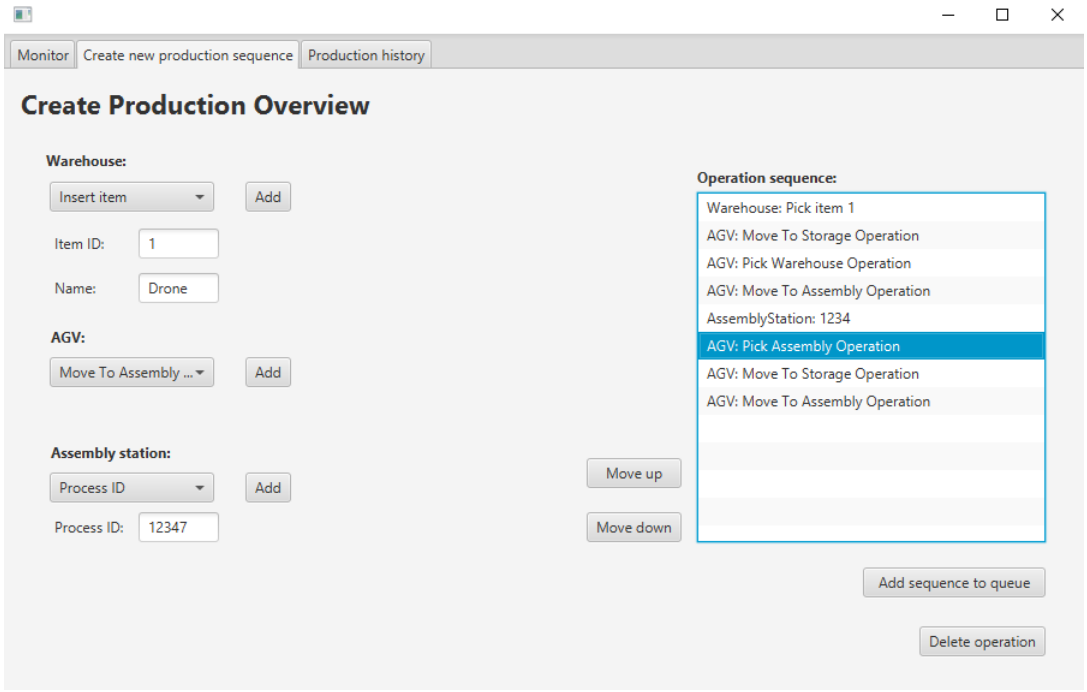


Figure 3.8 Production sequence illustration

3.3.3 Production history

The third and last tab called *Production history* shows the previously completed productions, allowing the user to determine what is needed for future productions. When a previous production is marked the operator can choose to add it to the current queue and start it if he so chooses. See Figure 3.9 for an illustration of this.

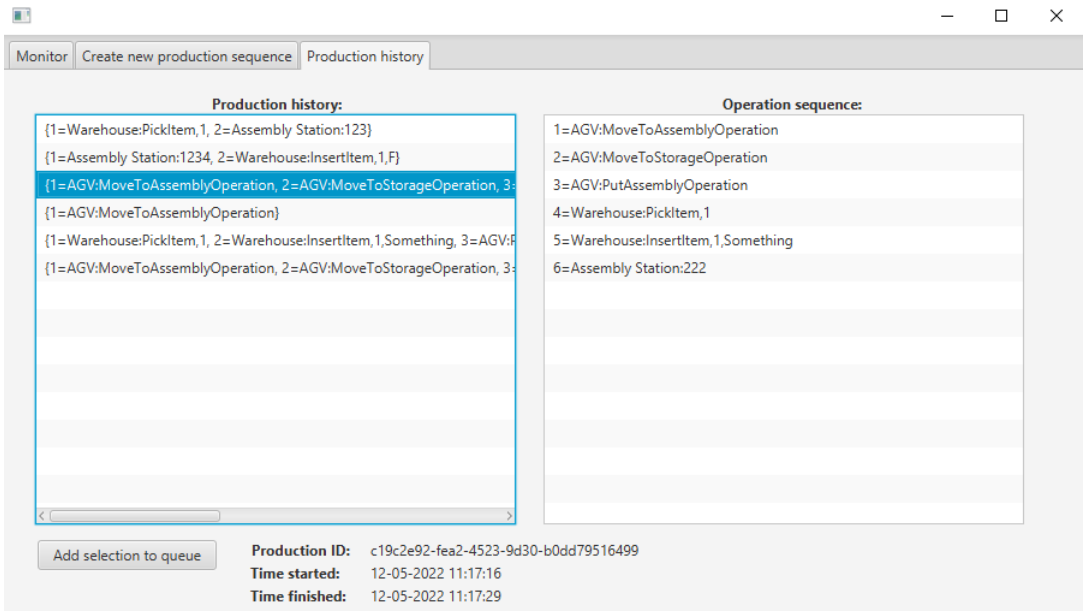


Figure 3.9 Production history illustration

Chapter 4

Implementation

4.1 Backend

This section will take a closer look at the backend implementation with a focus on the use-cases U01 Create a production and U04 Monitor production.

4.1.1 U01 - Create a production

The purpose of the *addAGVOperation* method shown in Listing 4.1 is to add the created AGV operation to the production. On line 60 a String variable called *selectedOperation* is saved with the value of the currently selected AGV operation in the application. On line 62 the *selectedOperation* variable is stripped from its white space characters and saved in a new String variable called *formatOperation*. This is done for the operation to be compatible with the input that the AGV asset requires. On line 63 the AGV operation is added to the production with a queue number in the form of the *queueCounter* variable. On line 64 the operation is added to the UI in the list of operations and lastly, the *queueCount* variable is incremented.

```
58 @FXML
59     void addAGVOperation(){
60         String selectedOperation = "AGV: " + choiceBoxAgv.
               getSelectionModel().getSelectedItem();
61         //Remove white space
62         String formatOperation = selectedOperation.replaceAll("\\s
               ", "");
63         operationQueueTreeMap.put(queueCount, formatOperation);
64         productionSequenceObservableList.add(selectedOperation);
65         queueCount++;
```

```
66      }
```

Listing 4.1: Code-snippet from the CreateProductionController of adding an AGV operation

The purpose of the *addWarehouseOperation* method shown in Listing 4.2 is to add the created warehouse operation to the production. On line 102 there is a switch case that differentiates whether the user has chosen the *Insert item* or the *Pick item* operation. This example will focus on the *Insert item* operation. On lines 104 and 105 an Integer variable called *itemID* and a String variable called *itemName* is saved with the values that the user has inputted. On line 106 a String variable called *productionInsert* is saved with the operation value including the *itemID* and *itemName* in a format that is compatible with the warehouse asset. On line 107 the *productionInsert* variable representing the operation is added to the production with the *queueCount* queue number. On line 108 the operation is added to the UI in the list of operations and lastly, the *queueCount* variable is incremented.

```
100 @FXML
101     void addWarehouseOperation(){
102         switch (choiceBoxWarehouse.getSelectionModel().
                getSelectedItem()){
103             case "Insert item":
104                 Integer itemID = Integer.parseInt(itemIDText.
                        getText());
105                 String itemName = nameText.getText();
106                 String productionInsert = "Warehouse:InsertItem,"
                        + itemID + "," + itemName;
107                 operationQueueTreeMap.put(queueCount,
                        productionInsert);
108                 productionSequenceObservableList.add("Warehouse:
                        Insert item " + itemName + ";" + itemID);
109                 queueCount++;
110                 break;
111             case "Pick item":
112                 Integer itemIDPick = Integer.parseInt(itemIDText.
                        getText());
113                 String productionPick = "Warehouse:PickItem," +
                        itemIDPick;
114                 operationQueueTreeMap.put(queueCount,
                        productionPick);
115                 productionSequenceObservableList.add("Warehouse:
                        Pick item " + itemIDPick);
```

```
116                 queueCount++;
117                 break;
118             }
119         }
```

Listing 4.2: Code-snippet from CreateProductionController of adding a warehouse operation

The purpose of the *addAssemblyOperation* method shown in Listing 4.3 is to add the created assembly station operation to the production. On line 70 a String variable called *processID* is saved with the user inputted process id. On line 71 a String variable called *formatProcessID* is saved with the same value as the *processID* variable but in a format that is compatible with the assembly station asset. On lines 72 and 73, the operation is added to the production and added to the operation list in the UI. Lastly, on line 74 the *queueCount* variable is incremented.

```
68  @FXML
69      void addAssemblyOperation() {
70          String processID = "AssemblyStation: " + processIDText.
                getText();
71          String formatProcessID = "AssemblyStation:" +
                processIDText.getText();
72          operationQueueTreeMap.put(queueCount, formatProcessID);
73          productionSequenceObservableList.add(processID);
74          queueCount++;
75      }
```

Listing 4.3: Code-snippet from CreateProductionController of adding an assembly station operation.

When the user has finished editing the production and its operations they will be able to add the production to the production queue. This is what the *addSequenceToQueue* method shown in Listing 4.4 handles. The for-loop on line 124 finds the service provider implementation of the *IProductionManager* service provider interface. Then on line 126 the *queueProduction* method is called with the production as a parameter. On lines 128-130 the operation map, the UI operation list, and the *queueCount* variable are all reset for the next creation of a production to be ready.

```
121 @FXML
122     void addSequenceToQueue(){
123         System.out.println("Adding sequence to queue...");
124         for (Map.Entry<String, IProductionManager>
                iProductionManagerEntry
```

```
125                 : context.getBeansOfType(IProductionManager.class)
                        .entrySet()) {
126             iProductionManagerEntry.getValue().queueProduction(
                    operationQueueTreeMap);
127         }
128         operationQueueTreeMap = new TreeMap<>();
129         productionSequenceObservableList.clear();
130         queueCount = 1;
131     }
```

Listing 4.4: Code-snippet from CreateProductionController of adding the production to the production queue.

This *queueProduction* method shown in Listing 4.5 takes the received production in the form of a Map and adds it to the production list queue.

```
120 @Override
121     public void queueProduction(TreeMap<Integer, String>
            productionMap) {
122             this.productionQueue.add(productionMap);
123     }
```

Listing 4.5: Code-snippet from the MesSpringClient class in the ProductionManager module. The code shows how the new production is added to the production list.

4.1.2 U04 - Monitor production

In order for the user to see the current values for any given asset, the UI requests the latest status on each asset. This is solved through threads so the update does not block other tasks in the system. The UI is updated every second which is specified on Line 86. The reference to the inner class, Line 85, can be seen in the Appendix A.

```
82 @Override
83 public void initialize(URL url, ResourceBundle resourceBundle) {
84
85     //Asset status threading - see inner class. SleepTimer for
           timer between updates
86     AssetUpdater assetUpdater = new AssetUpdater(1000);
87
88     Thread t1 = new Thread(assetUpdater);
89     t1.setDaemon(true);
90     t1.start();
91
92     warehouse_inventory_lst.setFocusTraversable(false);
```

```
93
94      q_lst.setItems(productions);
95
96  }
```

Listing 4.6: Code-snippet from MonitorController

Through the AssetUpdater inner class the *.getStatus* method, Line 197, is called on all instances requested through the *Spring Context*, Line 193, with the type *ICommunicatorService* which serves as an interface that ensures the presents of the *.getStatus* method. The data retrieved from the method-call is stored in a JSONObject, Line 197, and relevant data is fetched and sent to the UI.

```
193 for (Map.Entry<String, ICommunicatorService>
        iCommunicatorServiceEntry : context.getBeansOfType(
        ICommunicatorService.class).entrySet()) {
194
195      JSONObject json = null;
196      try {
197          json = new JSONObject(iCommunicatorServiceEntry.getValue()
                 .getStatus(assetManager));
198      } catch (JSONException e) {
199          e.printStackTrace();
200      }
```

Listing 4.7: Code-snippet from MonitorController

The concrete implementation of the *.getStatus* method is a bit different depending on which asset it is called upon, but it serves the same purpose; to fetch the current state from the assets. Listing 4.8 shows the implementation of the method on the AGV-asset. The implementation for the asset "Assembly Station" can be seen in the Appendix B and for the asset "Warehouse" in Appendix C.

```
38  @Override
39  public String getStatus(AssetManager assetManager) {
40      for (Asset asset : assetManager.getAssets(AGV.class)) {
41          String agvAPIUrl = "http://" + asset.getHost() + ":" +
                asset.getPort() + asset.getRequest();
42          HttpClient client = HttpClient.newHttpClient();
43          HttpRequest request = HttpRequest.newBuilder()
44                  .GET().header("accept", "application/json")
45                  .uri(URI.create(agvAPIUrl)).build();
46          try {
```

```
47              CompletableFuture<HttpResponse<String>> response =
                    client.sendAsync(request, HttpResponse.BodyHandlers
                    .ofString());
48              try {
49                  return response.get().body();
50              } catch (ExecutionException e) {
51                  e.printStackTrace();
52              }
53          } catch (InterruptedException e) {
54              e.printStackTrace();
55          }
56      }
57      return null;
58  }
```

Listing 4.8: Code-snippet from AGVControlSystem showing the getStatus method

Chapter 5

Validation & verification

The UPPAAL simulation & verification and unit & integration testing will be performed to validate and verify the functionality of the system. There will be no stress testing done because the system is only being used by one user at a time and the system is constrained to only run one production at a time to further lower workload.

5.1 UPPAAL verification & simulation

In order to test the developed system and see if it has the functionality required to satisfy the requirements from Section 2.3 a simulation was developed in UPPAAL to test a specific part of the project. This section focuses on formal verification and the graphical simulation of *Create Production* & *Queue production*. The application template of the simulation is seen in Figure 5.1 and the user template of the simulation can be seen in Figure 5.2.

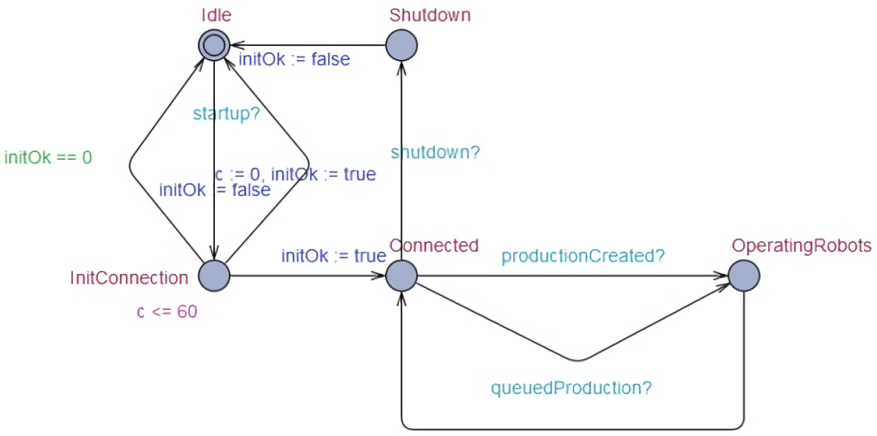


Figure 5.1 UPPAAL application template

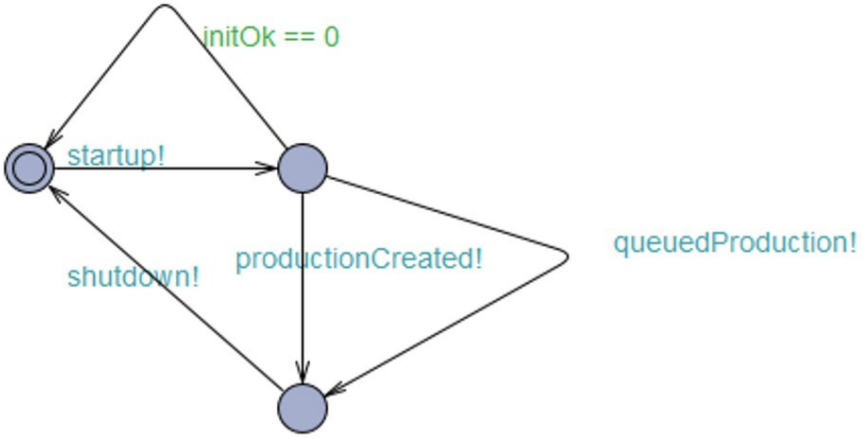


Figure 5.2 UPPAAL user template

As seen in Figure 2.1 *Create production* & *Queue production* are respectively connected to an activty block meant to satisy either of their requirements. In Figure 5.3 we verify that the simulation can successfully complete the *Create production* & *Queue production* transitions and transfer to the *OperatingRobots* state.

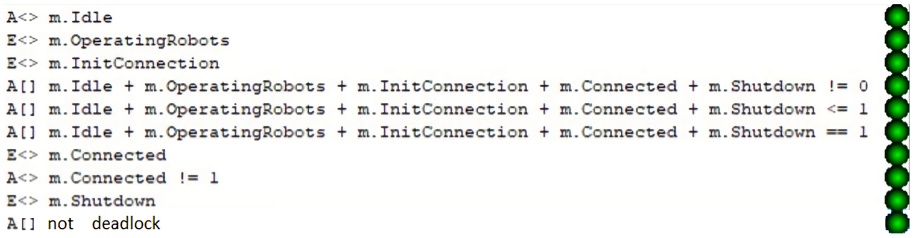


Figure 5.3 UPPAAL verification

Figure 5.3 also shows several other verification queries from the UPPAAL simulation. The m before the punctuation represents the automaton model and the text after the punctuation represents a state. The first verification query $A<>$ *m.Idle* checks whether the automaton m always for all execution paths eventually reaches the one state *Idle*. The $E<>$ *m.OperatingRobots* query checks whether the automaton m has one execution path that eventually reaches the one state *OperatingRobots*. The *A[] m.Idle + m.OperatingRobots + m.InitConnection + m.Connected + m.Shutdown == 1* query checks that there is only one active state for all execution paths. The *A[] not deadlock* query checks that the automaton does not reach any deadlocks in any of the execution paths.

5.2 Unit- & integration testing

The group has made an example of both unit tests and an integration test. This could be used to validate the functionality of the units and the system as a whole, when making changes to the code before deploying.

5.2.1 Unit tests

The group has made a unit test to test the AGV API by itself. This is done using JUnit 5. This AGVTest class starts the AGV Plugin and tests the different functions provided by the SPI, such as *getState*, *getStatus* and *performOperation*. An example of this can be seen below. The *testGetState()* function, Line 35, asserts that the state returned by the service provider is within the required range specified in the technical documentation, see Appendix D.

```
29      @BeforeAll
30      static void init() {
31          agvPlugin.start(assetManager);
32      }
33
34      @Test
35      void testGetState() {
36          int state = agvControlSystem.getState(assetManager);
37          assertTrue(state >= 1);
38          assertTrue(state <= 3);
39      }
```

Listing 5.1: Code-snippet for setting up the AGVTest class and running a test.

5.2.2 Integration test

Integration tests test the system as a whole, meaning it tests functionality from more than one component. In this case, the *startProduction* method from the *ProductionManager* component will be tested, as it affects multiple other components. It uses the *@SpringBootTest* notation to set up the *ApplicationContext*. The service providers are injected in to the SpringBootTest by scanning the base package in the TestConfig class. The ApplicationContext is tested in the contextloads function. In this function the asset plugins are started from the applicationcontext. After this, the startProduction function is tested. This is done by adding a production sequence and checking the status from the tested assets. This is shown in Listing 5.2.

```
53      @Test
54      @Order(2)
55      void testStartProduction() throws InterruptedException,
            JSONException {
56          TreeMap<Integer, String> productionSequence = new TreeMap
                <>();
57          productionSequence.put(1, "Warehouse:PickItem,1");
58          productionSequence.put(2, "Warehouse:InsertItem,1,Drone");
59          productionSequence.put(3, "AssemblyStation:1004");
60          productionManager.startProduction(productionSequence);
61          Thread.sleep(1000);
62          JSONObject json = new JSONObject(warehouseControlSystem.
                getStatus(assetManager));
63          JSONArray arrJson = json.getJSONArray("Inventory");
64          JSONObject inventoryItem = new JSONObject(arrJson.
                getString(0));
65          String content = inventoryItem.get("Content").toString();
66          assertTrue(content.equals("Drone"));
67
68          JSONObject json2 = new JSONObject(
                assemblyStationControlSystem.getStatus(assetManager));
69          String lastOp = json2.get("CurrentOperation").toString();
70          assertTrue(lastOp.equals("1004"));
71  }
```

Listing 5.2: Code-snippet for testing startProduction

The unit and integration tests are run and passes as seen below in Figure 5.4 and 5.5.

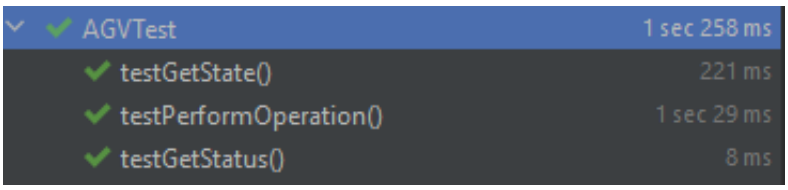


Figure 5.4 Unit test results: AGV

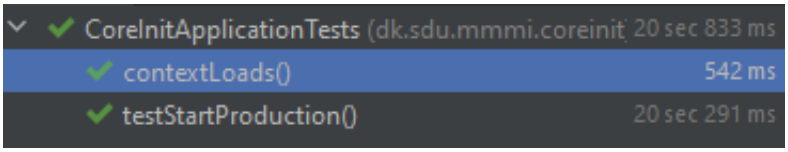


Figure 5.5 Integration test results

Chapter 6

Discussion

As written in the Sections 1.1 and 1.2, the aim and goal of this project was to develop a software system that interfaces up against different network protocols that each control a physical asset, which is part of a drone assembly line. This software system should decrease the workload of the operators that control the drone assembly line, by making it possible to create customized production sequences. The software system also functions as a log of all the operations performed by the assets, and generates a brief production sequence report for each production sequence.

This workflow allows for a lot of customizability and responsibility for the user of the system. The user has the freedom to create whichever sequence is necessary for the production of the drones, within the limitations of the asset interfaces and the technical possibilities of the physical assets. The production sequence will then be carried out exactly how it has been designed by the user, if all necessary production assets are online. However, the user has to ensure that the assets work in which ever way the production sequence is put together. This requires some basic technical background knowledge of the assets in the production line and how the different methods work together. This is expected according to the user target group defined in chapter 1.2.2.

The user interface was made simple according to the guidelines defined in the project description. The group concluded that it would not be necessary to create a web-based client-server application since the system currently is a single user system which uses the asset servers on localhost. The group decided that a web-based user interface would not provide any meaningful value with the current scope of the project. A JavaFX user interface provides the necessary features in order to create the required product, and integrates more easily into the Java application.

Even though it was a requirement from the project description to create a component-based application, it does make a lot of sense for a project like this. It provides the opportunity to easily replace any service provider implementations like the implementation of the different assets, while keeping the functionality the same, e.g. if the drone production line replaced an asset with another one that used a different network protocol, but had the same functionality, nothing would have to be changed except the specific asset component. The persistence component which has the responsibility of saving the reports, could also be swapped with another component. This could be the case if, for instance, the drone production line wanted the application to run without a database server or completely offline. In this scenario the reports could instead be saved in a JSON file or NoSQL database instead, by swapping out the Persistence component with another component that implements the persistence SPI.

The use of the Spring Boot framework was a very conscious decision as this is a flexible framework for component based applications. This makes it easy to not only create a component based application, but also create the database repository, or if, in the future, it would be necessary to create a web-based application, the Spring Boot framework makes it possible to easily create a REST API to use the functionality of the MES.

The group was limited by the functionality in the docker images provided by the project description. The docker images do not provide a lot of functionality for the assets, which makes it difficult for the group to create a less technical MES. I.e. features such as a GET request to get the supported operations for the AGV could also make the application more flexible when swapping out different components. The limited functionality of the provided docker images also caused some difficulties in respect to fulfilling the requirements of the system. An example of this would be the inability to implement the *U03 Execute emergency stop* requirement even though it has a *Must* prioritization, Table 2.1. The failure to fulfill this requirements is caused by the fact that the assets can not be stopped in the middle of execution. This fact also prevents the system from fulfilling the requirements *U07 Stop an asset in the production line* and *U08 Restart an asset in the production line*, Table 2.1. From the testing of the application it also seems that the asset servers are fragile and crash easily if a wrong request is sent thus requiring a restart of the docker containers, which is not always seen from the available state in their endpoints.

Having no direct contact with stakeholders such as an end-user, an investor or the provider of the task means that throughout development of the project, it was hard to get feedback on how a client would use the developed system. More specific require-

ments might have been developed and thus providing a better result if customer meetings had been arranged.

The group followed the agile Scrum method throughout the project. The project followed the problem definition from Section 1.2 and the requirements and analysis from Chapter 2. The project unfortunately experienced Risk R06, seen in Table 2.5, which impacted the development of the system, but due to the flexibility Scrum provides and by following the mitigation specified in Table 2.5 under R06, the impact of the risk was minimized. One aspect where the impact is seen the most is in the development of the software requirements specification from Section 2.3 where the requirements were elicited without any knowledge of the technical limitations of the system. The requirements *U03 Execute emergency stop* and *U08 Restart an asset in the production line* were not a possibility through the provided resource. This can be seen as an unnecessary work effort which removes time from other aspects of the development thus limiting the potential of the project.

Chapter 7

Conclusion

The developed system provides a solution for the production line at the SDU I4.0 lab through a component-based software which interfaces with the three assets in the production line using the communication protocols of MQTT, SOAP and REST. The solution provides means of monitoring and controlling the production of drones from a single system and provides value to the user through a graphical user interface which eases control and usability of the production line but still requiring the user to have a technical understanding of the production line.

The composed software requirements specification and the design of the system has created a foundation for the development of a prototype of a potentially more elaborate system that could handle the production of drones in the SDU I4.0 lab. Through formal verification techniques using UPPAAL as a tool of simulating requirements of the system it seems that the system functions as intended and satisfies most of the requirements specified early in the development. Using integration tests it is assured that external dependencies and that modules in the application are working correctly, and the use of unit-testing assures that each individual part of the program are functioning correctly. The system therefore correctly provides means of creating a production using the three assets in the system.

The system provides functionality of a degree to which is expected of a prototype-esque system would and provides the building blocks for developing a more advanced system. It provides the means of changing, adding and deleting different service providers in an easy manner to satisfy a potential need of scaling the system in the future and the Spring Boot framework in itself also provides a means of creating a more extensive persistence layer and provides means of developing a web-platform as well.

It can further be concluded that even though the project was influenced by the risk R06 as seen in Table 2.5, by following the mitigation specified for that risk the project was able to reach a complete state in relation to the problem definition. Despite delays for the delivery of the simulation accumulating to a total of one and a half month, the group managed to adjust the initial schedule and stay on track to deliver the project in an acceptable state. However, despite previously mentioned efforts, the finished results were still impacted.

Chapter 8

Perspectives

The developed software systems which goal is to interface with the three assets in the production line using the communication protocols MQTT, SOAP and REST, is specifically developed for SDU I4.0 lab only. Though, due to it being a component-based solution it is possible to either extend or change functionality to the already developed system, thus providing SDU I4.0 lab with the possibility to extend their production line or if any assets are upgraded, easily change functionality without effecting other components.

The system designed in this particular case solves one of many problems related to integration between different and independent software systems. The problem, that communication between applications may be limited or non-existing, is a very common issue and this project in particular tries to mitigate that issue by providing interoperability.

If this project was to be developed further, it would be interesting to find a potential end-user of the production line and have their inputs integrated into the design of the system. A lack of customer meetings during this project means that it was up to the group itself to specify requirements and an end-user would provide value to this specific iteration of the development phase.

Lastly, as Risk R06 indicated, the project suffered from delayed resources which impacted the workflow of the project. More specifically it impacted the development of the software system. If the delivery of the resources had not been delayed by one and a half month it would have been possible to focus on the development rather then mitigating the impact it had on the project thus resulting in a more focused development.

Appendix

A Inner class from Listing 4.6 - U04 monitor production

The full implementation of the AssetUpdater inner class.

```
161 public class AssetUpdater implements Runnable {
162
163     private long sleepTime;
164     private boolean running;
165     private String previousQueueContent = "";
166
167     public AssetUpdater(long sleepTime) {
168         this.sleepTime = sleepTime;
169     }
170
171     @Override
172     public void run() {
173         running = true;
174
175         while (running) {
176             Platform.runLater(new Runnable() {
177                 @Override
178                 public void run() {
179
180                     for (Map.Entry<String, IProductionManager>
                            iProductionManagerEntry : context.
                            getBeansOfType(IProductionManager.class).
                            entrySet()) {
181                         String currentQueueContent = String.
                                valueOf(iProductionManagerEntry.
                                getValue().getProductionQueue());
182
183                         if (!currentQueueContent.equals(
                                previousQueueContent)) {
184                             productions.clear();
185
186                             for (TreeMap<Integer, String> tm :
                                    iProductionManagerEntry.getValue().
                                    getProductionQueue()) {
187                                 productions.add(tm.toString());
188                             }
189                             previousQueueContent =
                                    currentQueueContent;
190                         }
```

```
191                     }
192
193                     for (Map.Entry<String, ICommunicatorService>
                            iCommunicatorServiceEntry : context.
                            getBeansOfType(ICommunicatorService.class).
                            entrySet()) {
194
195
196                      JSONObject json = null;
197                      try {
198                          json = new JSONObject(
                                iCommunicatorServiceEntry.getValue
                                ().getStatus(assetManager));
199                      } catch (JSONException e) {
200                          e.printStackTrace();
201                      }
202
203                      if (iCommunicatorServiceEntry.getKey().
                            equals("warehouseControlSystem")) {
204                          try {
205                              warehouse_state_lbl.setText(json.
                                    getString("State"));
206                              if (json.getString("State").equals
                                    ("1")) {
207                                  asset_warehouse_lbl.
                                        setTextFill(Color.color(0,
                                        1, 0));
208                                  asset_warehouse_lbl.setText("
                                        Warehouse - RUNNING");
209                              } else {
210                                  asset_warehouse_lbl.
                                        setTextFill(Color.color(0,
                                        0, 0));
211                                  asset_warehouse_lbl.setText("
                                        Warehouse");
212                              }
213                              SimpleDateFormat ft = new
                                    SimpleDateFormat("yyyy-MM-dd'T'
                                    HH:mm:ss");
214                              Date timestamp = ft.parse(json.
                                    getString("DateTime"));
```

```
215                         SimpleDateFormat ft2 = new
                                SimpleDateFormat("dd-MM-yyyy HH
                                :mm:ss");
216                         ft2.setTimeZone(TimeZone.
                                getTimeZone("GMT+4"));
217                         warehouse_ts_lbl.setText(ft2.
                                format(timestamp));
218                         warehouse_inventory_lst.getItems()
                                .clear();
219                         warehouse_inventory_lst.getItems()
                                .add("ID    Item");
220                         JSONArray arrJson = json.
                                getJSONArray("Inventory");
221                         for (int i = 0; i < arrJson.length
                                (); i++) {
222                             JSONObject inventoryItem = new
                                    JSONObject(arrJson.
                                    getString(i));
223                             warehouse_inventory_lst.
                                    getItems().add(
                                    inventoryItem.getString("Id
                                    ")
224                                         + "  |  " +
                                            inventoryItem.get("
                                            Content"));
225                         }
226                     } catch (JSONException e) {
227                         break;
228                     } catch (ParseException e) {
229                         e.printStackTrace();
230                     }
231                 }
232
233                 if (iCommunicatorServiceEntry.getKey().
                    equals("AGVControlSystem")) {
234                     try {
235                         agv_state_lbl.setText(json.
                                getString("state"));
236                         if (json.getString("state").equals
                                ("2")) {
237                             agv_asset_lbl.setTextFill(
                                    Color.color(0, 1, 0));
```

```
238                                 agv_asset_lbl.setText("AGV -
                                        RUNNING");
239                             } else {
240                                 agv_asset_lbl.setTextFill(
                                        Color.color(0, 0, 0));
241                                 agv_asset_lbl.setText("AGV");
242                             }
243                             SimpleDateFormat ft = new
                                    SimpleDateFormat("yyyy-MM-dd'T'
                                    HH:mm:ss");
244                             Date timestamp = ft.parse(json.
                                    getString("timestamp"));
245                             SimpleDateFormat ft2 = new
                                    SimpleDateFormat("dd-MM-yyyy HH
                                    :mm:ss");
246                             ft2.setTimeZone(TimeZone.
                                    getTimeZone("GMT+4"));
247                             agv_ts_lbl.setText(ft2.format(
                                    timestamp));
248                             agv_program_lbl.setText(json.
                                    getString("program name"));
249                             agv_battery_lbl.setText(json.
                                    getString("battery"));
250                         } catch (JSONException e) {
251                             break;
252                         } catch (ParseException e) {
253                             e.printStackTrace();
254                         }
255                     }
256
257                     if (iCommunicatorServiceEntry.getKey().
                            equals("assemblyStationControlSystem"))
                             {
258                             try {
259                                 as_state_lbl.setText(json.
                                        getString("State"));
260                                 if (json.getString("State").equals
                                        ("1")) {
261                                     asset_assembly_station_lbl.
                                        setTextFill(Color.color(0,
                                        1, 0));
```

```
262                                     asset_assembly_station_lbl.
                                            setText("Assembly Station -
                                             RUNNING");
263                                 } else {
264                                     asset_assembly_station_lbl.
                                            setTextFill(Color.color(0,
                                            0, 0));
265                                     asset_assembly_station_lbl.
                                            setText("Assembly Station")
                                            ;
266                                 }
267                                 SimpleDateFormat ft = new
                                        SimpleDateFormat("yyyy-MM-dd'T'
                                        HH:mm:ss");
268                                 Date timestamp = ft.parse(json.
                                        getString("TimeStamp"));
269                                 SimpleDateFormat ft2 = new
                                        SimpleDateFormat("dd-MM-yyyy HH
                                        :mm:ss");
270                                 ft2.setTimeZone(TimeZone.
                                        getTimeZone("GMT+4"));
271                                 as_ts_lbl.setText(ft2.format(
                                        timestamp));
272                                 as_last_op_lbl.setText(json.
                                        getString("LastOperation"));
273                                 as_curr_op_lbl.setText(json.
                                        getString("CurrentOperation"));
274                             } catch (JSONException e) {
275                                 break;
276                             } catch (ParseException e) {
277                                 e.printStackTrace();
278                             }
279                         }
280
281                     }
282                 }
283             });
284             synchronized (this) {
285                 try {
286                     wait(sleepTime);
287                 } catch (InterruptedException e) {
```

```
288                         System.out.println("Interrupted: " + Thread.
                                currentThread());
289                         running = false;
290                     }
291                 }
292             }
293         }
294 }
```

Listing 1: Inner class AssetUpdater from MonitorController

B Implementation of the getStatus method for the Assembly Station

Line 87 ensures that the message is not returned until it is set.

```
77  @Override
78  public String getStatus(AssetManager assetManager){
79      if(client == null){
80          init();
81      }
82
83      lock.lock();
84      try {
85          while (message.equals("")) {
86              try {
87                  notSet.await();
88              } catch (InterruptedException ignored){}
89          }
90      } finally{
91          lock.unlock();
92      }
93      return getMessage();
94  }
```

Listing 2: Code-snippet from AssemblyStationControlSystem showing the getStatus method

The message variable is updated through the messageStatus method. On Line 124 the **setMessage** method is called which updates the variable "message".

```
113 public void messageStatus(){
114     assert client != null;
115     client.setCallback(new MqttCallback() {
116
```

```
117         @Override
118         public void connectionLost(Throwable throwable) {
119             throwable.printStackTrace();
120         }
121
122         @Override
123         public void messageArrived(String s, MqttMessage
                mqttMessage) throws Exception {
124             setMessage(mqttMessage.toString());
125         }
126
127         @Override
128         public void deliveryComplete(IMqttDeliveryToken
                iMqttDeliveryToken) {
129             try {
130                 iMqttDeliveryToken.getMessage();
131             } catch (MqttException e) {
132                 e.printStackTrace();
133             }
134         }
135     });
136 }
```

Listing 3: Code-snippet from AssemblyStationControlSystem showing the messageStatus method

The message called within the messageStatus method from Listing 3

```
142 public void setMessage(String message){
143     lock.lock();
144
145     try {
146         this.message = message;
147         notSet.signal();
148     } finally {
149         lock.unlock();
150     }
151 }
```

Listing 4: Code-snippet from AssemblyStation showing how the lock is unlocked

C Implementation of the getStatus method for the Warehouse

The method **getInventory**, Line 44, is auto-generated with the tool *wsimport* and therefore not included further.

```
42 @Override
43 public String getStatus(AssetManager assetManager) {
44     return warehouseService.getInventory();
45 }
```

Listing 5: Code-snippet from WarehouseControlSystem showing the getStatus method

D Technical Documentation

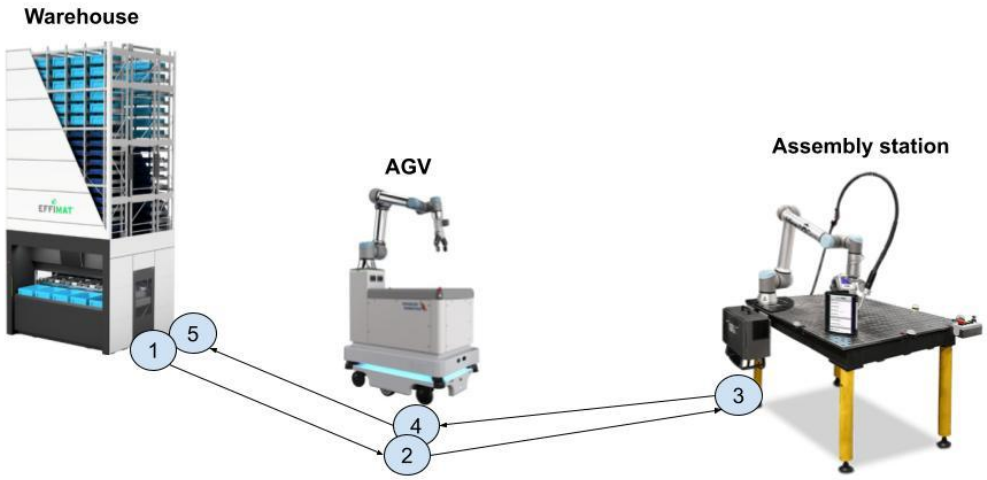
The documentation starts on the next page.

The simulated I4.0 lab

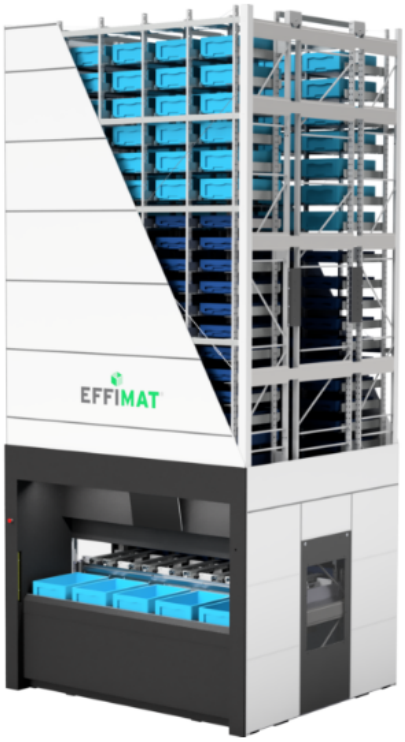
Software Technology 4th semester project

Introduction

This document will explain the features and functionality of the simulated lab assets that are involved as part of the (ST4-PRO) 4th semester Software Technology project, along with documentation explaining the implementation and use of the interfaces. Please refer to the project description for the project description and goals.



Warehouse



Asset communication

The warehouse in the SDU I4.0 lab lets users store parts and pieces of the current drone product line, while handling all of the internal shelf-mechanics and optimization algorithms internally on the PLC. It is typically operated by users through an onboard interface via a touch-screen mounted on the front of the machine. One of the goals of the Industry 4.0 initiative is easy connectivity to external systems using software, so the vendor's software has an implementation of a web service that communicates using SOAP as its messaging protocol.

> *"**SOAP** (Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP), although some legacy systems communicate over Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.*
>
> *SOAP allows developers to invoke processes running on disparate operating systems (such as Windows, macOS, and Linux) to authenticate, authorize, and*

communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms."

- *https://en.wikipedia.org/wiki/SOAP*

To communicate with the service, you will have to implement a component that can communicate with the SOAP service. For most programming languages you can find public libraries that greatly simplify this process, so look at the options and choose a fitting one for your solution.

Features and functionality

The simulated warehouse has a few basic features which are implemented and exposed by the interface. These make it possible to do basic box retrieval and inserts, as well as checking the current inventory of the asset, which also returns state information.
The warehouse consists of removable boxes, which are placed in non-removable trays inside the warehouse itself. For interaction with the API, you will be working with tray IDs and content names for pick and insert operations.

The SOAP endpoint for implementation can be found at **http://localhost:8081/Service.asmx**, which contains the WSDL that describes the details of requests and data structure.

Method	Description
PickItem(int trayId)	Request an item from the warehouse.
InsertItem(int trayId, string name)	Insert an item into the warehouse.

GetInventory	Return example:
	{
	"Inventory":
	[{
	"1": "Item 1",
	"2": "Item 2",
	"3": "Item 3",
	"4": "Assembly 1",
	"5": ""
	}],
	"State": 0,

	<pre>"TimeStamp": 12:34:56
	}</pre>

State information

State	Description
0	Idle
1	Executing
2	Error

AGV

Asset communication

In SDU's I4.0 lab we use AGVs to move parts between stations where transportation is not easily accessible via other means (conveyor belt etc.). AGVs provide a flexible solution to automated transportation, since they can handle collision, pathing and optimization of those paths as they are repeated.

The current AGVs in the I4.0 lab communicate using a REST API, which is widely regarded as a flexible web service that is easy to work with. Often regarded as the successor to SOAP, as it provides similar functionality but is simpler to work with.

> *"Representational state transfer (**REST**) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasizes the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems.*[1]

REST has been employed throughout the software industry and is a widely accepted set of guidelines for creating stateless, reliable web APIs. A web API that obeys the REST constraints is informally described as RESTful. RESTful web APIs are typically loosely based on HTTP methods to access resources via URL-encoded parameters and the use of JSON or XML to transmit data."

- *https://en.wikipedia.org/wiki/Representational_state_transfer*

Features and functionality

The AGV consists of a set of pre-programmed paths and programs, which is programmed on the teach pendant (controller/screen) of the AGV. Through a REST web service it is possible to execute these predefined programs and request status-data.

The web service of the simulated asset (localhost for local execution) is hosted on **http://localhost:8082/v1/status/**.
Currently, the implementation requires a user to load a program with a PUT request and execute the preloaded program with another PUT request - similar to loading a gun and then pulling the trigger in two separate steps. The `"State": 1` attribute must be present when loading a new program, like in the example below.

Load a program on the AGV:

```
{
    "Program name": "MoveToAssemblyOperation",
    "State": 1
}
```

Execute the loaded program by sending a simple JSON object that forces state to 2, starting execution, e.g:

```
{
    "State": 2
}
```

For each PUT request you send, the asset will return status information. To request status information without using a PUT request, you can send a GET request, which will return a similar JSON response without any programs being executed. Example of return object format:

```
{
      "Battery": 42,
      "Program name": "MoveToAssemblyOperation",
      "State": 2,
      "TimeStamp": 12:34:56
}
```

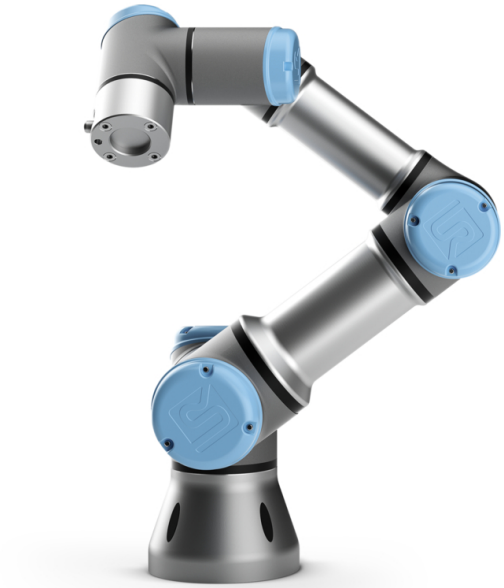
You can invoke the following methods by specifying the name in the PUT request:

Program name	Description
MoveToChargerOperation	Move the AGV to the charging station.
MoveToAssemblyOperation	Move the AGV to the assembly station.
MoveToStorageOperation	Move the AGV to the warehouse.
PutAssemblyOperation	Activate the robot arm to pick payload from AGV and place it at the assembly station.
PickAssemblyOperation	Activate the robot arm to pick payload at the assembly station and place it on the AGV.
PickWarehouseOperation	Activate the robot arm to pick payload from the warehouse outlet.
PutWarehouseOperation	Activate the robot arm to place an item at the warehouse inlet.

State information

State	Description
1	Idle
2	Executing
3	Charging

Assembly station



Asset communication

Assembly processes in the SDU I4.0 lab are handled in a variety of ways. There are automated processes where robots collaborate with other robots, collaborative processes where robots and humans work together to complete the task, and also processes that are handled by human hands only.

The processes involved in the current product line are strictly automatic, where robots work together to finish the assembly without manual input. We control these stations through a MQTT implementation, which is a scalable and lightweight solution that we can adapt to future requirement changes.

> *"**MQTT** (originally an initialism of Message Queueing Telemetry Transport) is a lightweight, publish-subscribe network protocol that transports messages between devices. The protocol usually runs over TCP/IP, however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT.[1] It is designed for connections with remote locations where resource constraints exist or the network bandwidth is limited. The protocol is an open OASIS standard and an ISO recommendation (ISO/IEC 20922)."*

- https://en.wikipedia.org/wiki/MQTT

Features and functionality

To call the methods available at the assembly station, you will need to implement a MQTT client that can talk to the broker which is hosted by the asset and can be accessed locally on port 1883. A typical URL would look something like **mqtt://localhost:1883/**.

The implementation of MQTT has three topics which you can subscribe to and publish messages to. Once subscribed to a topic, you should receive any updates that the asset publishes that topic.
Execution of the assembly process can be started by sending a process ID on the operation topic, whose value must be an integer. When using simulated assets you can specify any integer to start the assembly process.

You can publish and subscribe to the following topics:

Topic	Description

emulator/operation	Topic to execute programs.
	
	Input to start an operation must be JSON format, similar to:
	`{`
	`  "ProcessID": 12345`
	`}`
	
	To start an unhealthy assembly process you can publish “9999” to the topic. This can be used to verify correct error handling when subscribed to checkhealth.

emulator/status	The asset will broadcast frequent status messages on this topic in a JSON format, which is suitable for heartbeat/connection status etc.
	
	`Return example:`
	`{`
	`  "LastOperation": 1234,`

	<pre>"CurrentOperation": 2345,
	"State": 1,
	"TimeStamp": 12:34:56
	}</pre>

emulator/checkhealth	Result of the assembly process quality control. Result is published on the topic when the assembly process has finished, and will always be healthy for simulated processes unless forced.

State information

State	Description
0	Idle
1	Executing
2	Error

Deployment and usage

To run the services required for testing your software, ***you must have Docker installed*** on your machine to execute the associated images.
All the images will be publicly hosted on DockerHub, where you can pull them individually or use a docker-compose file to manage execution.
You can find an already defined docker-compose.yml file at https://github.com/ThMork/ST4-compose/blob/main/docker-compose.yml which will manage all the containers for you. Through your preferred terminal (cmd, powershell, git bash etc.) you can run the '**docker-compose up**' command in the directory of the file to initialize and run the images.

Note: If you have an m1 processor, you need to edit to docker-compose file to target the correct build. You can find the tag on DockerHub below. If you have no idea what this means, you likely won't need to change anything.

DockerHub images

- thmork/st4-warehouse
- thmork/st4-agv
- thmork/st4-assemblystation
- thmork/st4-mqtt

Content of docker-compose.yml

```yaml
version: '3.8'

services:
  mqtt:
    image: thmork/st4-mqtt:latest
    ports:
      - 1883:1883
      - 9001:9001

  st4-agv:
    image: thmork/st4-agv:latest
    ports:
      - 8082:80

  st4-warehouse:
    image: thmork/st4-warehouse:latest
    ports:
      - 8081:80
```

```
st4-assemblystation:
  image: thmork/st4-assemblystation:latest
  environment:
    MQTT_TCP_CONNECTION_HOST: "mqtt"
    MQTT_TCP_CONNECTION_PORT: 1883
```

Development tools

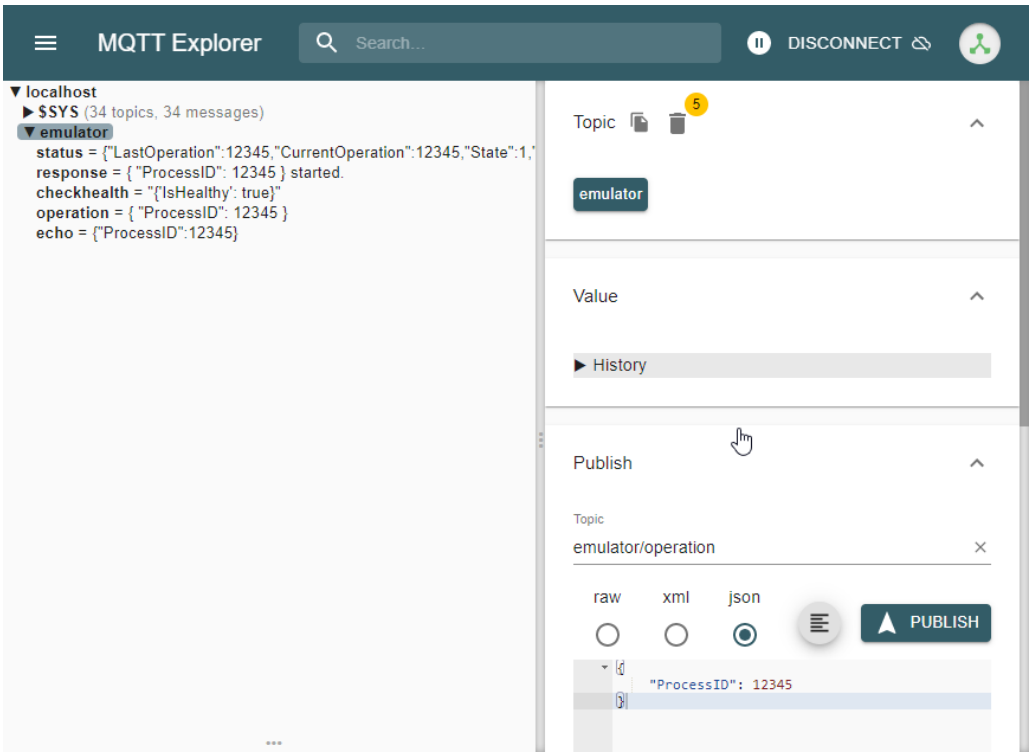
To help you verify the functionality of either containers or your code, and for exploring the APIs and functionality of these, you can use the following tools.

MQTT Explorer

Useful for exploring the MQTT broker, where you can see all different topics in use and publish messages on a given topic.

Download: http://mqtt-explorer.com/.

Example: Starting a process on assembly station from MQTT Explorer

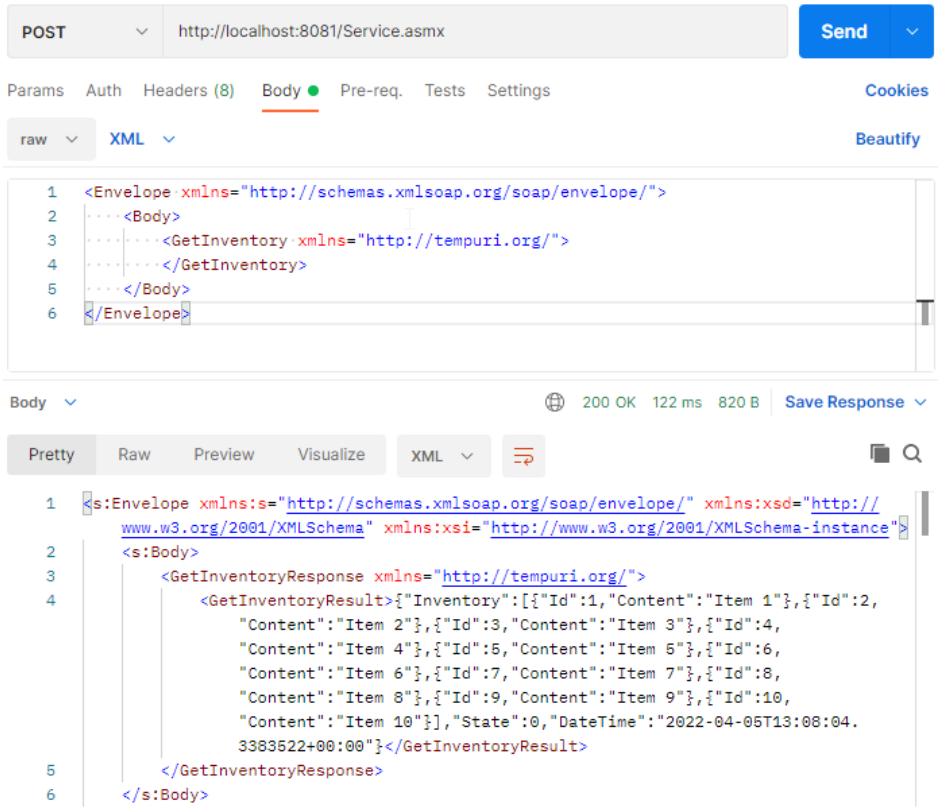


Postman

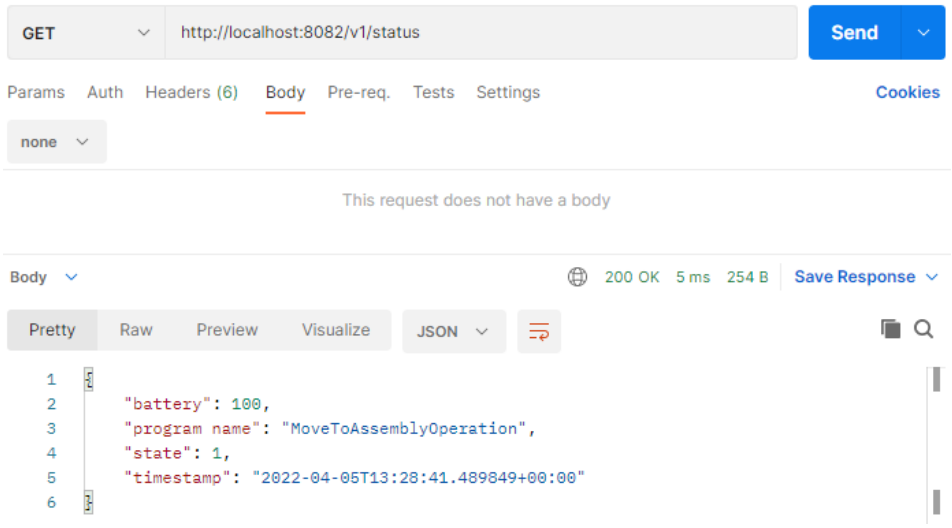
You can use Postman to send different types of http requests to an endpoint. In the ST4 project you can use it to verify the functionality of the AGV (get and put requests) and the Warehouse (construct raw xml objects to emulate implementation).

Download: https://www.postman.com/downloads/.

Example: Warehouse ‘GetInventory’ HTTP POST request



Example: AGV status information by HTTP GET request



Example: AGV load ‘MovetoAssemblyOperation’ by HTTP PUT request