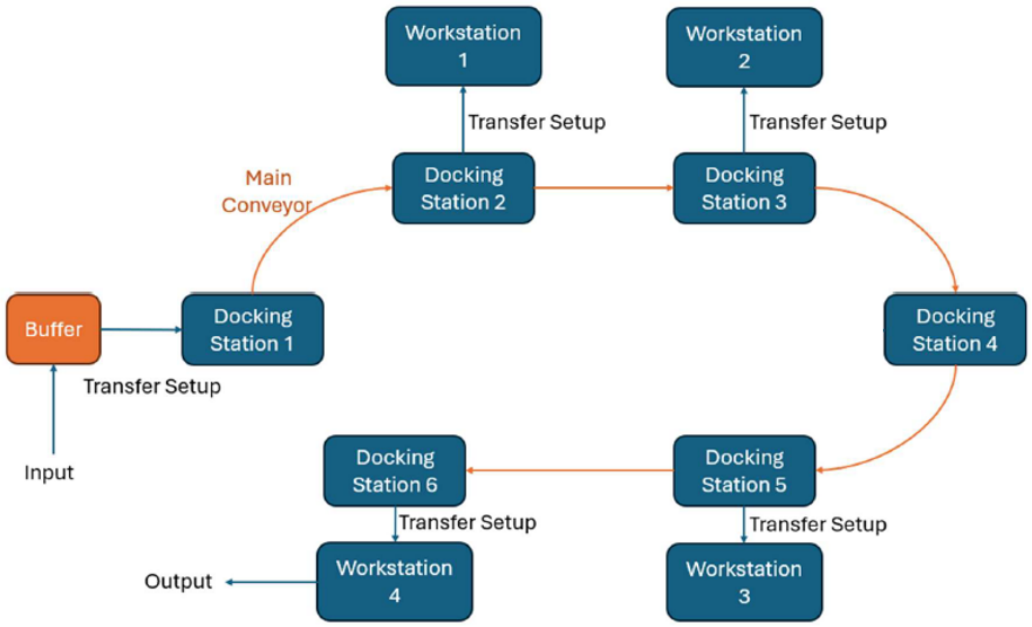


Documentation to Digital Model: A Python Based Simulation of a Drone Assembly Line

Developing, verifying and validating a Python-based digital model of a drone factory, from PLC documentation, for future Digital Twin

Master's thesis in Production Engineering

JOYEL JOSEPH JOBI
JIAHAO LI

DEPARTMENT OF INDUSTRIAL AND MATERIAL SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Documentation to Digital Model: A Python Based Simulation of a Drone Assembly Line

Developing, verifying and validating a Python-based digital model of a drone factory, from PLC documentation, for future Digital Twin

JOYEL JOSEPH JOBI
JIAHAO LI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Industrial and Material Science
Division of Production Systems
Chalmers University of Technology
Gothenburg, Sweden 2026

Documentation to Digital Model: A Python Based Simulation of a Drone Assembly Line
Developing, verifying and validating a Python-based digital model of a drone factory, from PLC documentation, for future Digital Twin
JOYEL JOSEPH JOBI
JIAHAO LI

Supervisor: Silvan Marti, Chalmers University of Technology
Examiner: Björn Johansson, Chalmers University of Technology

Master's Thesis 2026
Department of Industrial and Material Science
Division of Production Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Drone Factory Model Layout.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Documentation to Digital Model: A Python Based Simulation of a Drone Assembly Line
Developing, verifying and validating a Python-based digital model of a drone factory, from PLC documentation, for future Digital Twin
JOYEL JOSEPH JOBI
JIAHAO LI
Department of Industrial and Material Science
Chalmers University of Technology

Abstract

Digital simulation models serve as the foundational component of Digital Twins for enabling real-time monitoring, predictive analysis and data driven decision making across production systems. As industries aim to evolve from Industry 4.0, the demand for flexible, transparent and cost efficient simulation tools continues to grow. Traditional DES platforms offer strong modelling capabilities, but are often limited by reduced customizability and challenges in integrating real-time data streams. To address these limitations, this thesis develops a Python based digital model of a semi-automated drone assembly line and evaluates its feasibility as a foundation for a future Digital Twin. The model was implemented using a modular architecture that replicates the physical system's conveyor, docking-station logic, timing dependencies and PLC document workflows. Verification was performed against the factory's PLC documentation and validation was conducted through scenario based experiments and cross comparison with a similar model in Siemens Plant Simulation. The python based digital model developed successfully reproduces both structural and behavioural characteristics of the real system, achieving an average accuracy of 87% similarity in production time performance and 84% average accuracy in station cycle time, though station cycle time performance varied between 60% to 98% in various scenarios. These findings demonstrate that Python provides a customizable and transparent platform for developing Digital Twin simulation models in discrete manufacturing environments.

Keywords: Digital Twin, Python Simulation Model, Discrete Event Simulation, Model Validation and Verification.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Silvan Marti, whose continuous support, guidance and encouragement have been invaluable throughout the completion of this thesis. His insights and feedback greatly strengthened the quality of our work.

We would also like to extend our heartfelt thanks to Sven Ekered for providing the essential technical details and clear explanations regarding the Drone Factory setup, which played a crucial role in shaping our understanding of the system.

Our sincere appreciation goes to our examiner, Björn Johansson, for giving us this wonderful opportunity to deepen our knowledge and for his constructive evaluation of our work.

We would further like to thank Anders Ankén, Student Guidance Counsellor for the MSc Production Engineering programme, for his continuous support and for helping us navigate our academic journey successfully.

Lastly, we are deeply grateful to our families and friends for their unwavering support, patience, and encouragement throughout this learning journey.

This report has been prepared with the support of AI tools such as Gemini and Copilot, which were used to refine grammar, clarity and structure.

Joyel Joseph Jobi and Jiahao Li, Gothenburg, June 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AS	Assembly Station
CFD	Computational Fluid Dynamics
DS	Docking Station
DES	Discrete Event Simulation
DT	Digital Twin
FEA	Finite Element Analysis
HMI	Human-Machine Interface
IoT	Internet of Things
KPI	Key Performance Indicator
LAN	local Area Network
PLC	Programmable logic controller
RFID	Radio Frequency Identification
SFC	Sequential Function Chart
WS	Work Station

Contents

List of Figures

List of Tables

1
Introduction

This chapter gives an overview of the project scope starting with the current trends in the field followed by Aim and Research questions.

1.1 Background

The rapid digitalization of manufacturing has accelerated the transition toward the interconnected and smart production environments envisioned in Industry 4.0.[5] These industrial transformations emphasize real time data exchange, cyber-physical connectivity and the integration of advanced information technologies into production systems. As noted by Li et al., smart manufacturing has become the central direction of global industrial development, driven by national strategies such as Germany's Industry 4.0 and China's Made in China 2025, which highlight the need for digital, networked, and intelligent production capabilities[6]. Within this context, the Digital Twin has emerged as a key enabler, providing a high-fidelity virtual representation of physical systems that supports real-time monitoring, prediction, and optimization.
Simulation has long played a foundational role in manufacturing analysis, enabling engineers to evaluate system behaviour, test alternative configurations, and reduce operational risks. Traditional simulation models, however, often operate as isolated digital representations with limited or no automated data exchange[7]. As Kritzinger et al. and Martinez et al. emphasize, much of the existing literature still falls within the categories of Digital Models or Digital Shadows, where integration with physical systems is partial or unidirectional, and fully realized Digital Twins remain scarce in practice . This gap highlights a broader challenge: while simulation is widely used, its evolution toward fully synchronized, bidirectional Digital Twin architectures is still in its early stages[7][8].
At the same time, the rise of Python-based simulation offers new opportunities for flexible, transparent and customizable modelling. Unlike commercial DES platforms, Python uses modular architectures that can be tightly integrated with data channels, IoT devices and analysis tools essential for digital twin development. But there is very limited researches exploring PLC derived python simulation models as practical foundations for Digital Twins in manufacturing. This thesis project is aligned to analyse this approach by answering the three research questions later discussed in section 1.3.

1.2 Aim

This project aims to develop, verify and cross-validate a Python based digital model of a semi-automated drone factory, derived from PLC documentation, to serve as a foundation for future Digital Twin development.

The study focuses on three main objectives: developing a python simulation model from a PLC document, verifying the simulation model using a structured verification approach and cross validating the model with a DES model. By addressing these aspects, the thesis seeks to bridge the gap between traditional simulation practices and emerging Digital Twin oriented modelling approaches directly in Python.

1.3 Research Questions

To explore the feasibility of using Python based digital models for Digital Twins, the following three research questions were formulated. This section introduces the research questions and discusses the relevance with current scope in the field. These research questions are answered in Results and Discussion section.

1.3.1 Research Question 1

Identify the benefits and trade-offs of using a Python based Digital Simulation model over commercial DES tools for this drone factory.

Traditional DES tools are powerful but often closed, proprietary and difficult to extend. Python, on the contrary offers modularity, transparency and good integration potential required for Digital Twin evolution. Understanding these benefits establishes why Python is a meaningful alternative in modern manufacturing simulation. This research question intends to compare the functionality, efficiency and financial perks associated with Python simulation model over Traditional techniques and DES models.

1.3.2 Research Question 2

How accurately can the component logic, timing dependencies and mission workflows be translated from a PLC document into a modular Python simulation architecture?

Smart manufacturing systems involve complex, dynamic interactions that must be represented faithfully for a model to be useful [6]. When models are manually constructed from system logic, the implementation procedure itself becomes a potential source of error. Ensuring that each step of the modelling process is accurate is therefore essential for producing a reliable digital model that reflects the intended system behaviour. This guarantees that the Python model forms a reliable basis for further verification and Digital Twin development. This research question intends to test the accuracy of devised implementation procedure by comparing functioning of the digital model with the PLC document.

1.3.3 Research Question 3

How can the Python simulation model be verified and validated using PLC documentation variables and comparison with a DES model?

Verification and validation are critical to ensure that a Python based model behaves consistently with established DES tools, which are widely trusted in industry. Validating the model with the current industry standard provides a structured pathway to assess

accuracy, reliability, and Digital Twin potential. This question focuses on creating a method to verify and validate the simulation model based on current industry standards.

2
Theory

This chapter discusses the definitions of different concepts used throughout the project. The concepts are explained using simple examples and figures. Finally it introduces the characteristics of Drone Factory model that is developed in the project.

2.1 Digital Twin

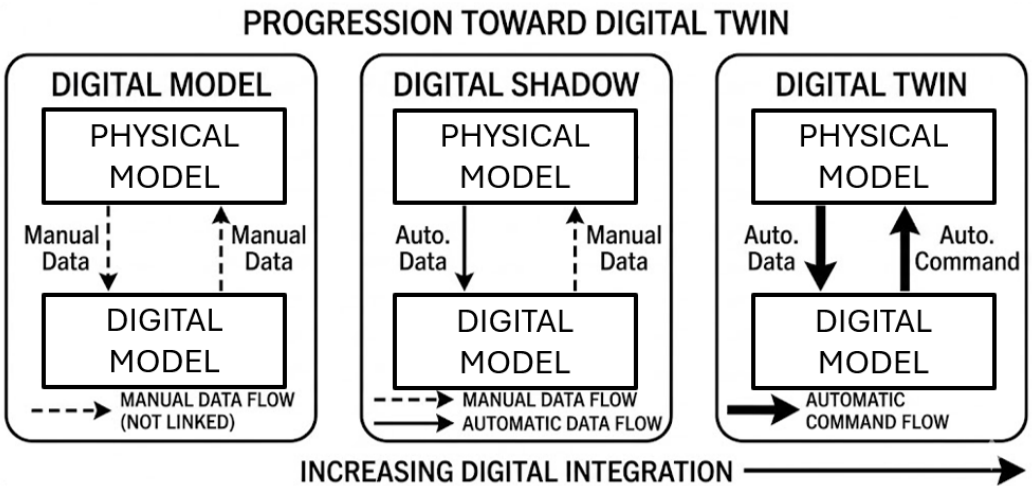


Figure 2.1: Comparison of Digital Model, Digital Shadow and Digital Twin.[1][2]

In this context, there are essentially three stages of virtual model representations based on data synchronization, they are: Digital model, Digital shadow and Digital twin see Figure 2.1. A digital model is just a 3-dimensional digital blue print or plan of a physical or hypothetical entity generated using a computer. The model is manually updated by the user. Digital shadow is an active digital representation that reflects the changes of a physical system in the virtual model automatically, here the data flow is single directional from the physical system to virtual system. A true digital twin is a real time depiction of a physical object or system. Both the digital model and its physical entities are always in synchronization through automatic bi-directional data exchange. In traditional digital twin systems this inter connection is enabled via technologies like IoT, LAN, 5G, WiFi[1][2].

The concept of digital twin can be explained in detail with the example of a simple drone factory. The drone factory has PLC layer, that controls the components and subsystems

like conveyor, actuators, Pneumatic system, RFid system. And this drone factory PLC system is connected with the digital model through the connection layer. For a simple conveyor operation: The PLC system sends a signal to the conveyor motor, and as the conveyor moves the sensors in the conveyor system sends a state transition signal back to the PLC layer. These I/O signals in the PLC layer are accepted for updating the digital model. Similarly, when a change is made in the digital model it is send as input signal to the PLC system for implementing in the real model.[3]

An intelligent digital twin system has the capability to sustain a mutual symbiotic relationship with its real world counterpart[9], such that it can modulate (monitor, diagnose, predict and adapt) its real world counterpart in a self sustaining environment. These features of digital twin makes it a promising technology to enable other manufacturing concepts like predictive maintenance, bottleneck detection, product life cycle management and so on[10][11].

The python based simulation model aligns with the early stages of digital twin progression (as shown in figure 2.1). The digital model does not receive real time data from the assembly line, all the data are input manually. The python model can be developed based on the logical and physical behaviour of the drone factory, so that it can evolve into a digital shadow and later a digital twin[8]. According to the four stage framework on AI-enabled throughput bottleneck analysis, python based simulation model also has scope to enable AI based bottleneck prediction and production management support [12].

2.2 Discrete Event Simulation

There are variety of simulation techniques used for simulating digital models based on the field of application like FEA, CFD, DES to name a few. In this project we will focus on DES approach for cross validation and a direct model simulation approach in python.

DES is a technique used to simulate digital models, in the form of state changes that occur as the outcome of events at discrete points in time. It does not follow a continuous simulation time, instead it just depicts state changes in model based on the logic conditions of an event, then records these changes and schedules the next chain of events in the form of loop until an end condition is met.[13][14]

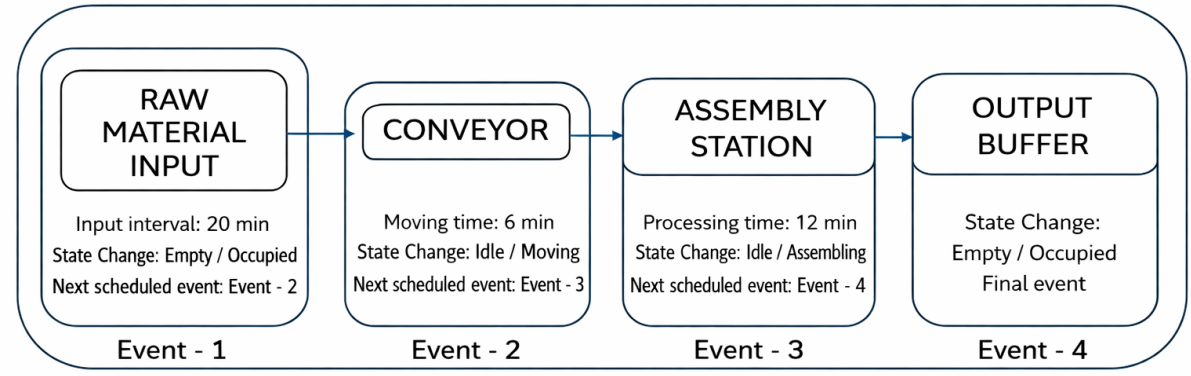


Figure 2.2: A simple Drone factory example.

The working of DES concept can be understood with example of Drone factory see Figure 2.2. In this example we consider the assembling of a single product. The drone factory consist of a Raw material Input, conveyor, assembling station and an output buffer. Raw material input time, Transportation time and Processing time are the various boundary conditions. The simulation starts with executing the conditions associated with the first event which is material input time and shows the state of the component, then it schedules the next event which is Event-2. In the second event conveyor moving time is the conditional variable and the state changes from idle to moving for the provided time, later event-3 is scheduled. This is process is repeated until the final scheduled event is executed.

Discrete Event Simulation (DES) models are extensively used across manufacturing, health, aerospace industries to represent event based simulation and decision making within digital twin researches [15] [8] [16]. Since, DES is considered as the accepted approach for digital modelling in the field, it can be used as a reference for comparing with the python based simulation model. Siemens Technomatix Plant Simulation is a standard DES software used in manufacturing and production industries. As it is reliable and suitable for the project scope, we have used it to cross validate the developed model.[17].

2.3 Simulation in Python

This part provides a introduction and comparative theoretical analysis of modern architectural approaches used to develop computer simulation engines in the Python programming language. Specifically, it explores how developers bridge the gap between continuous physical processes and discrete-event frameworks to build real-time applications like industrial Digital Twins and Internet of Things (IoT) ecosystems[18].

Python is an open source programming language.Due to its object oriented programming features it is much easier to model and develop complex systems and functionalities[19, 18]. The user can easily break down system components into classes and use them according to need. The extensive user base and community, provides numerous shared libraries that can be used for specialised tasks like simulation, scientific computing, machine learning, visualisation, data handling to name a few[20]. Being an open source software it is almost free for every application. Developers can easily experiment and develop their own tools and software with it.

SimPy is a popular, general-purpose discrete-event simulation library in Python that controls time advancement using an event-stepped engine[21]. Active components within the simulation are implemented as Python generator functions and managed by an environment class using a global event queue. While SimPy natively supports real-time simulation, it is designed strictly for discrete events and offers no built-in features for continuous systems[22]. To handle mixed discrete-continuous environments, developers must build custom, abstract class wrappers around SimPy to serve as an event interface that manages continuous states alongside the global event list.

In modern Digital Twin and IoT environments, systems are often classified by how their internal states progress over time. Some systems behave continuously where variables change and progress are smooth and without interruption, while discrete systems update their state only at specific, event-driven moments see section 2.2. When both behaviours are implemented together in a system, it becomes a Mixed Discrete–Continuous (MDC)

system.[23] Because digital computers operate on discrete steps rather than true continuity, they cannot directly process continuous smooth changes. Instead, continuous behaviour must be approximated by evaluating the system at selected time intervals. In cases where the underlying physical relationships are relatively simple and linear, these time-stepping routines can be made more efficient through predictive methods. Here, the future state can be calculated directly from the current one, allowing the simulation to jump to the exact moment when a variable reaches a threshold rather than iterating through every small time increment.

The Mixed Discrete-Continuos (MDC) system is typically approached in two main architectural ways:

- In a continuous-loop setup, time advances through a single time-stepped simulation cycle, and discrete events are handled as conditional updates inside that loop. But, this approach forces the entire system to run at a small, fixed step size determined by stability requirements, which can become computationally expensive for larger or more complex models.[23]
- The alternative is a discrete-engine approach, where time progresses through an event-driven algorithm and continuous processes are embedded within this framework. Here, continuous components are wrapped in abstract classes that expose a unified event interface. This interface groups continuous–discrete interactions into four functional categories: perturbation events, which represent sudden external changes to the continuous state; probe events, which request an immediate state update up to the current simulation time; generated events, which are emitted when a continuous variable crosses a defined threshold; and wakeup events, which are internally scheduled callbacks that allow the continuous component to resume execution at its next predicted or fixed update point.[23]

Together, these mechanisms allow continuous dynamics to coexist cleanly within a discrete event simulation environment.

Simulating connected, real-time IoT systems or industrial digital twins introduces a set of practical challenges related to data transfer, cyber security and software execution. Modern continuous engines shift away from legacy client-only desktop architectures toward distributed model federations hosted via remote web servers. Using a lightweight Web Server Gateway Interfaces such as the Flask micro-framework makes the system as a modular service.[24] But this may increases the chance for data breach. Another challenge arises with the shared memory and read-write allocations. The Read threads and Write threads has to access same memory location exclusively for proper functioning, this conflict can be solved by implementing a Fair Reader-Writer Lock structure. This synchronization primitive ensures that multiple readers can comfortably share the critical data block, but cleanly isolates and halts active reading loops when the write head needs access.[24]

2.4 Drone Factory Model

The Model developed in this project is based on the Drone factory facility available at Chalmers campus in Lindholmen, Gothenberg. It is a semi-automated assembly line. The internal document “Program code, WS Conveyor SII-Lab”[3] and the existing physical model were used as reference for developing the python model. The physical and functional characteristics of the model are discussed in detail with respect to the layout

diagram.

Figure 2.3: Photo of the Drone Factory facility in SII-Lab, Chalmers campus Lindholmen, Gothenberg

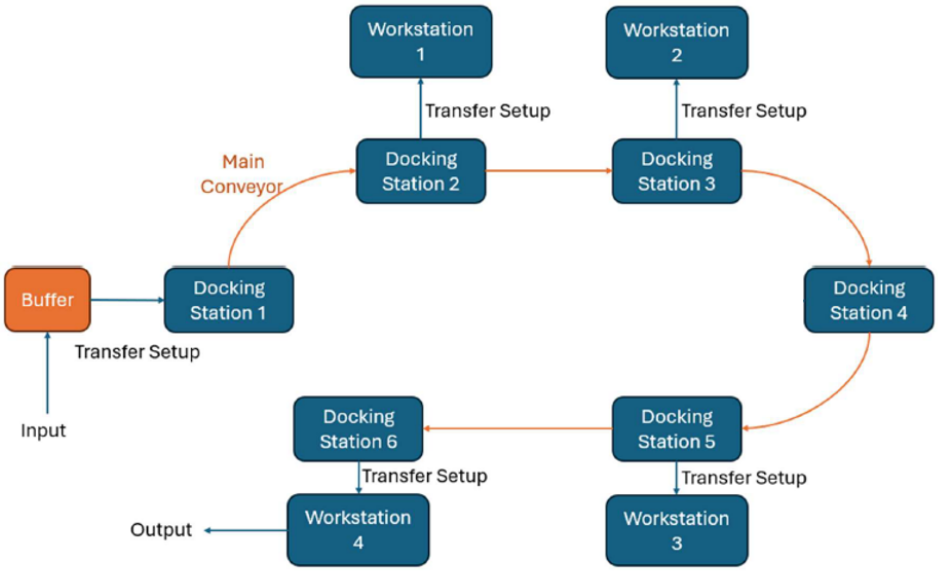


Figure 2.4: Layout of Drone Factory Model. Based on the Drone Factory Program Code document. [3]

2.4.1 Model layout and Components

The drone factory model consist of several interconnected subsystems like main conveyor, transfer, buffer, docking, workstation and control systems to support the assembly flow.

Transportation of pellets are automated based on the HMI inputs but the assembling process is solely dependent on the workers efficiency. The main components and details are organised into the Table 2.1. A detailed table with all the System Components, Interfaces, Functions and State changes are included in the Appendix A.1 System Architecture

- **Main Transport System**. The Main Transport System uses a motor-driven conveyor fitted with sensors, queue stops, and pallet-position detectors to control how materials move through the line. [3]
- **Transfer and Docking System**. Vertical movement between levels is handled by the Elevator, which includes the lift itself and a set of tracking sensors that monitor its position and loading state. For directing pallets across different routes, the docking System relies on a cross transfer conveyor supported by fixed stops, queue stops and transfer stops that scans, identifies and controls the path of the pallet.[3]
- **Buffer System**. To prevent blockages in production line, the Buffer system uses its own conveyor with embedded sensors, along with queue, fix and transfer stops, to temporarily hold pallets before releasing them back into the main flow.[3]
- **Workstation**. The workstation includes height adjustable worktable with sensors, assembling parts and a HMI panel.
- **Control system**. The control layer consist of an HMI interface, a Soft PLC system, an RFID-based tracking setup, a pneumatic system and a Mission Dispatcher. It synchronizes and handles the functioning of the drone factory.[3]

Table 2.1: Drone Factory Components

Components	Details
Main Transport System: Main Conveyor, Conveyor motor with sensors, Queue stops	Speed range 20-100%, Max speed 0.18m/s, Ramping time 2s, Total length 14.5m, 2.0 to 3.5m between each Docking station)
Buffer System: Buffer Conveyor, Sensors, Queue stops	Capacity 4 pallets, Raw Material Input every 120 to 900 seconds
Transfer Setup: Elevator and Cross-Transfer belt	Transfer time 3 seconds
Assembly Stations	4 stations, Manual Operation, Processing time 120-900 seconds
Docking Stations	6 stations
Pallet	Dimension 0.25 X 0.25 m, RFID, Station number, My mission
Controls Layer	HMI Interface, Soft PLC System, Rfid System, Mission Dispatcher, Pneumatic System

2.4.2 Control Architecture

The figure 2.5 shows an overview of hardware network layout. The main Control unit of the factory consist of a Soft PLC X2 Control. The PLC operates on CODESYS V3.5 and is connected to an HMI panel running ix 2.40 SP5. The sensors and actuators are connected to an I/O Link Master. The main PLC uses Crevis NA-9286 I/O module to access the sensors through the Link Master. EtherCAT and Ethernet are used to connect the hardware components like PLC, I/O module and other peripheral devices. This control architecture supports both automated and manual commands when required.

HMI panels present at workstations provide the operators with overview and control of the whole assembly line. The operators can easily assign automated mission sequences, manually control conveyor and pneumatic functions, control the table height, test I/O signals from sensors and actuators, assign delay times at queue stops and define buffer quantity. (see appendix A.2 HMI Panel Functionalities.)

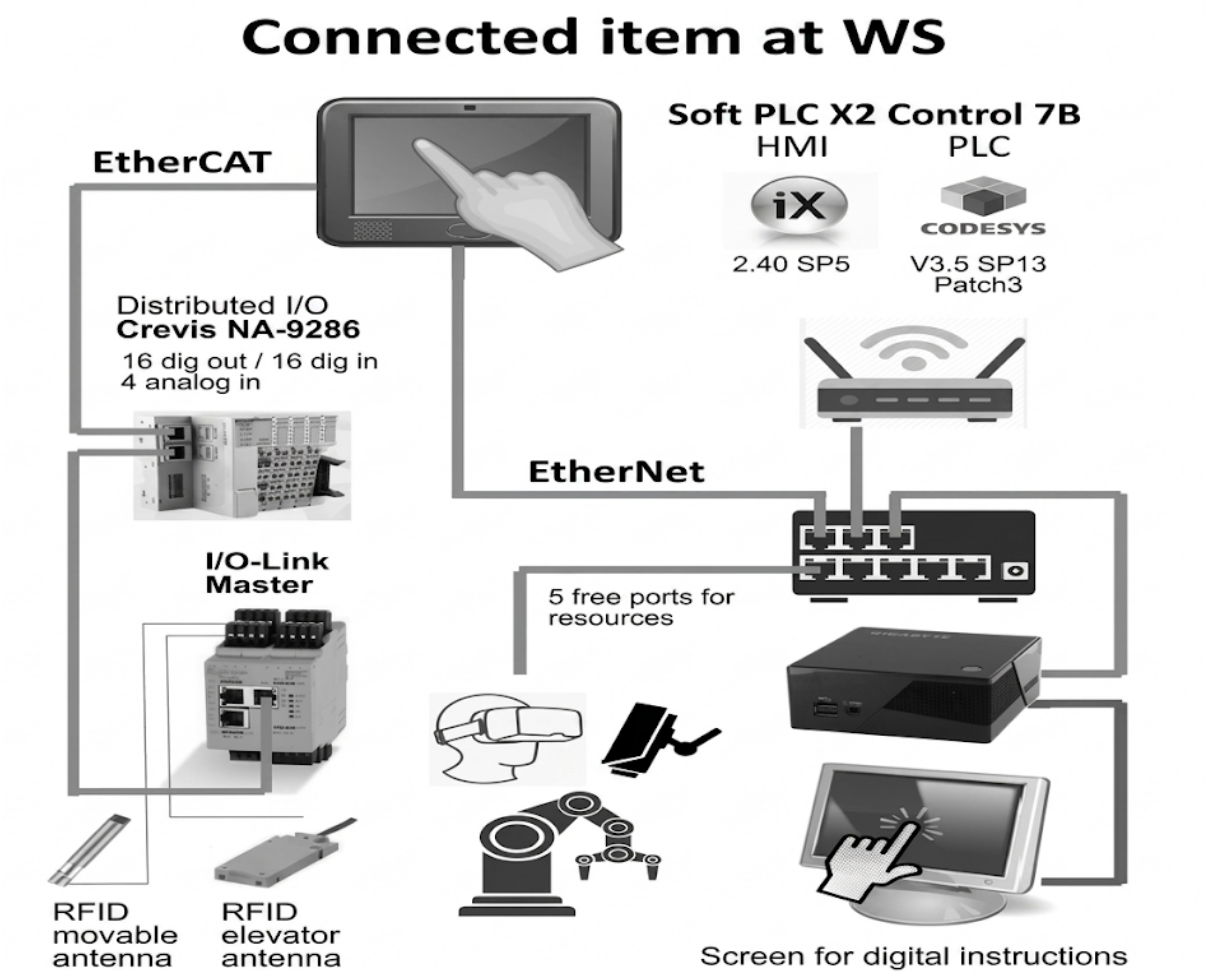


Figure 2.5: Hardware and Network Layout of the drone factory from the program code manual [3].

2.4.3 Time constraints and Work flow

The production process follows a strict sequence of time constraints and dependencies. The Work flow explained here can be observed in accordance with the Layout Diagram see Figure 2.4. The process begins with the Entry stage, where raw material pallets are input at an interval of 120 to 900 seconds into the Buffer. Pallets exit the Buffer station through the Docking Station. Exit/Entry at Buffer involves lift and lateral transfer which takes 3 to 5 seconds. It is then conveyed along the main line to next docking station, here it reaches a queue stop and held for a 0.5-second RFID reading stabilization delay. [3]

During the next stage (Station Entry), the workflow splits based on the assigned mission: Pallets to be transferred to the Assembly Station (WS Branch) undergo a 3-second lift

and transfer followed by a configurable mandatory wait of up to 4.0 seconds. And the pallets that needs to skip the Docking station is directly allowed to pass. An auto pass timer of 5 seconds is activated at Docking station to prevent interlocking. [3]

In the Operation stage, the Assembly Station requires 120 to 900 seconds of manual processing depending on operator efficiency. It also includes a temporary buffer branch that holds items automatically until the assembly process is over.

The next stage (Station Exit) the docking station manages the return flow, allocating up to 10 seconds for workstation pallets to transfer out and lift down (triggering an error if exceeded). Pallets merge back onto the main line to leave the docking area. The assembled product then reaches the Quality Check station and final product is obtained here concluding with a configurable safety wait of up to 1.0 second before the sequence clears. [3]

3
Methodology

This section includes all the working procedure that is followed throughout the project timeline. An overview of project plan is initially discussed followed by detailed description of the various Phases.

3.1 Model Development Framework

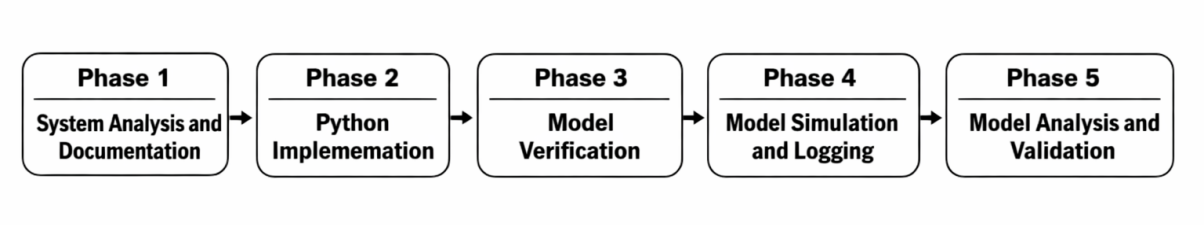


Figure 3.1: Project Methodology followed for Model Creation. Inspired from Robert G. Sargent's paper [4]

Model Development involved Five Phases. Phase 1 Mapping the whole system and formulating all the details into a single Document file. Phase 2 Implementing the model in Python, based on the System Documentation. Phase 3 Verifying that the model components behave similar to the real system components and the script is error free. Phase 4 Simulating the model over extended periods and successfully creating log files. Finally Phase 5 Analysing the developed model and Validating it with DES model. The PLC document to Python traceability table 3.3 discusses the conversion of PLC characteristics to python model with a short description.

3.1.1 System Analysis and Documentation

Initially the PLC document of the Drone Factory was analysed. All the different components, systems, workflows and time constraints were identified. These were then compared with the actual model and confirmed. Based on this a System Analysis Document was prepared which included Complete components list, system interactions, timing dependencies and system architecture. The digital model was developed based on this system document.

3.1.2 Python Implementation

This model was implemented as a Python-based step-driven simulation environment derived form the PLC documentation. Following the idea of Laftchiev et al.[25], raw PLC

digital outputs are transformed into discrete event sequences, which are modelled as the Physical Abstraction Layer. These sequences are then aggregated into a global process state, reflected in the Global State Layer, while event ordering constraints and valid transitions are encoded in the Logic Layer. The Configuration Layer captures patterns, loops, and parameter settings inferred from training data. The foundational architecture of the simulation model was realized as an independent, Python-based simulation environment, strictly following these layers:

- **Physical Abstraction Layer:**At the foundation, this system models the physical drone production line by building a complete suite of core hardware abstractions, which was based on the original production line manual (cited manual), including the Main Conveyor, Transfer Elevators, Buffer, Docking Stations, Pneumatic System and RFID Reader modules.

- **Global State Layer:** To ensure the system unity and structural integrity across the different modules, a centralized State container was implemented for management and registration of all components' real-time states, rather than operating it as isolated items. The state manager synchronizes the elapsed time across all modules and generating system-wide snapshots during execution.

- **Logic Layer:** Beyond individual component behaviours, cross-component orchestration and dynamic scheduling are implemented through a collaboration between the logic module and the simulation driver. Controllers and helper routines encapsulate domain rules such as ingress timing, workstation route construction, and conveyor transit-time estimation. The simulator then executes these rules in each simulation tick, coordinates interactions across components, and advances global system state.

- **Configuration Layer:** An independent configuration architecture was employed to facilitate the model with the ability to switch the scenario and repeat the experiments. Mutable simulation parameters like simulation time step, simulation times and division of labour at workstations are isolated from the core logic. A parameter validation mechanism is also applied during initialization.

Finally, the PLC logic of each component was compared with the PLC document to confirm the functioning, the model was debugged and constructed as a Python-based simulation environment with 4 layers, like illustrated in Figure 3.2. During system initialization, the model was loaded and registered the parts of core hardware abstractions into the centralized State container without structural conflicts. Then, tests on the configuration architecture confirmed that the system can apply operational parameters and intercept illegal configurations prior to runtime.

3.1.3 Model Verification

The simulation model was verified through a layered testing framework designed to evaluate both local component behaviours and system-wide execution. This approach ensures

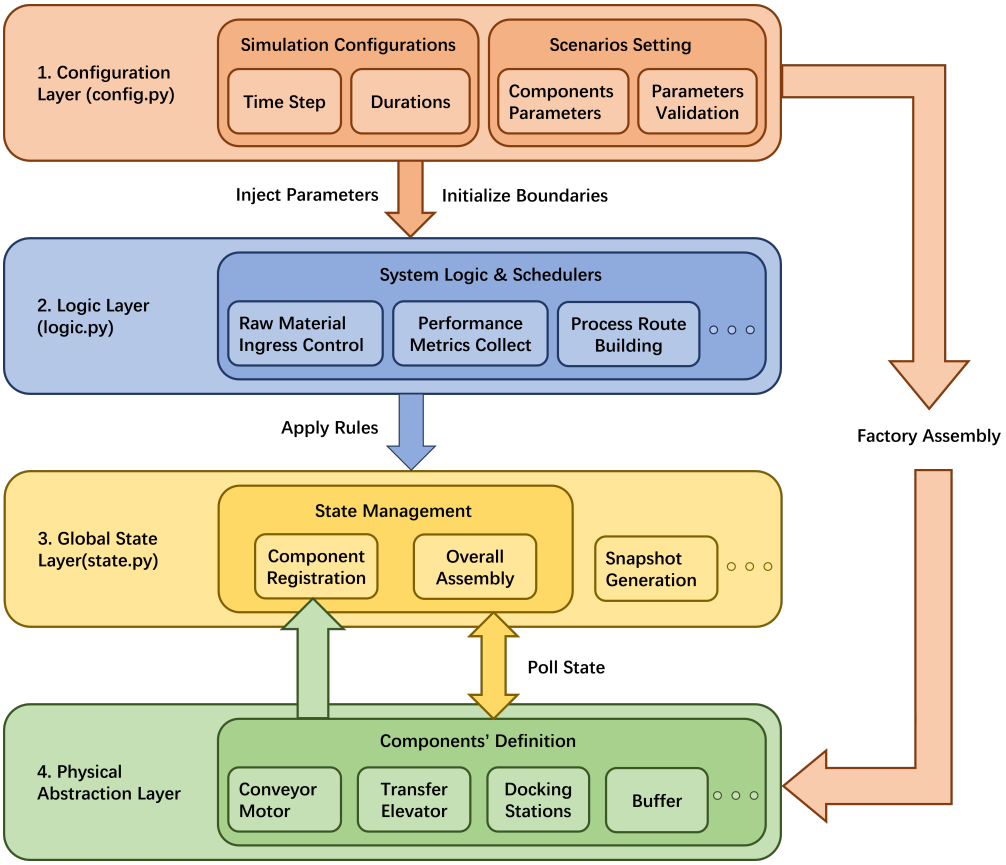


Figure 3.2: Structure of Implementation

the model accurately reflects the physical control logic such as interlocks, timed dependencies, and resource contention while keeping each claim traceable to documented PLC scenarios.[4]

- **Component-Level Verification:** Individual hardware abstractions (e.g., conveyors, elevators, buffers, and RFID modules) were tested in isolation. These state-machine-oriented tests verify legal transitions, prevent incompatible commands, and verify fault recovery paths without involving full-line orchestration.
- **Integration Verification:** This tier evaluates cross-module interactions and safety interlocks. It enforces shared constraints, such as buffer-full shutdowns, shared transfer-path contention, and elevator-docking synchronization. The focus is to verify multi-component operations against strict PLC-style dependency rules.
- **Simulation-Level Verification:**The top tier tests the fully integrated simulator under representative configurations. It evaluates global state validity, complex timing interactions, determinism, and long-term operational stability. This layer catches emergent errors specific to closed-loop scheduling that isolated tests might miss.
- **Trace-Level Consistency Checks:** Complementing the assertion-based tests, a log-centric protocol was implemented to guarantee reproducibility. Repeated simulations under fixed configurations produce identical, verifiable event traces. This proves that the model's externally recorded behaviour is stable, auditable, and diverges only when stochastic inputs are intentionally varied.

3.1.4 Model Simulation and Logging

A simulation engine was created which can plan and perform component interactions based on a virtual clock. A logging system was developed that records component state transitions, performance metrics and errors into a structured log file based on virtual time. Finally, long period simulations (more than 24 hour long) were run to test the stability of simulation engine and logging system.

3.1.5 Model Analysis and Validation

In this phase the results obtained from simulation logs are analysed and converted into a system dynamic report. This report is organised into visual plots of the model's performance metrics for easy decision making.
As a final step of validation, a similar DES model of Drone factory was developed in Siemens Technomatix Plant simulation software for cross comparison [17]. A threshold benchmark of 80% similarity between the results of both models was agreed as a validation criteria for the experiment.

3.1.6 Plant Simulation Model

The following boundary conditions and experiment scenarios were fixed for validating the model with Plant Simulation model see table 3.1. Boundary conditions included varying the conveyor speeds, Raw material intervals and station processing times. Sim talk methods are used for implementing buffer logic, docking station and station assignment. Each pallet was assigned station number and mission number based on the available stations to prevent interlock and for easy navigation. Docking stations scan the pallets and according to pallet details transfer pallet to the respective assembly station or forward to the next station.[3]

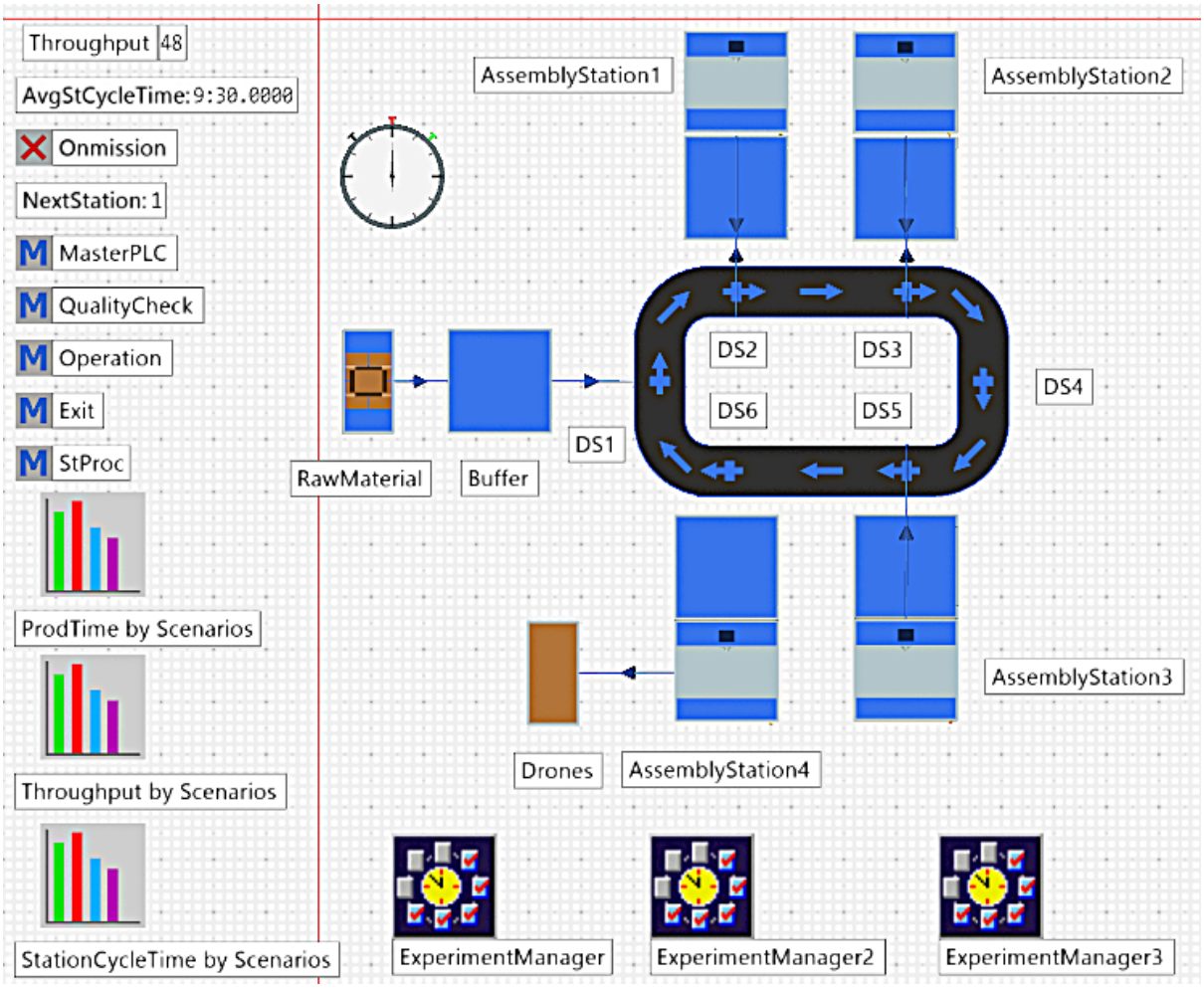


Figure 3.3: Drone factory model in Siemens Plant Simulation.

Table 3.1: Boundary Conditions

Variables	Conditions
Conveyor Speed	0.18 m/s
Conveyor Length	16.25 meters
Buffer Capacity	4 pallets
Raw Material Input Interval	10 minutes
Assembly Stations	4 stations
Docking Stations	6 stations
Assembling Procedure	Parallel / Single station assembly
Station selection (DS2, DS3, DS5, DS6)	Round Robin mechanism
Raw material delivery	To Buffer
Final product	Obtained at assembly Station 4
Pallet Dimension	0.25 X 0.25 m
Pallet details	Station number, My mission
Simulation Time	8 hours

3.1.7 Validation Experiment

Worker efficiency was varied keeping the boundary conditions fixed. Two efficiency levels were assigned for each worker. Highest efficiency where the worker completes whole procedure in 2 minutes and lowest efficiency where the worker completes the procedure in 12 minutes. These efficiency times were implemented as station processing time in both models. The Experiment table 3.2 shows each scenario tested. Three separate experiment managers were created in Plant simulation software to record Production Time, Station cycle time and Throughput. In total 40 independent simulation runs were conducted for each test variable, 10 for each scenario. The average of these observations were plotted against each scenarios. The outcomes of these three scenario KPIs (Production time, Station cycle time and Throughput) are used to validate the model.[26]

Table 3.2: Experiment Table

Scenario	Station 1 (DS2)	Station 2 (DS3)	Station 3 (DS5)	Station 4 (DS6)
Scenario 1	12 minutes	2 minutes	2 minutes	2 minutes
Scenario 2	2 minutes	2 minutes	2 minutes	2 minutes
Scenario 3	12 minutes	12 minutes	2 minutes	2 minutes
Scenario 4	12 minutes	12 minutes	12 minutes	12 minutes

Table 3.3: This table compares and discusses the traceability of PLC documentation to Python model

PLC Documentation Element	**Python Model Artifact**	**Description of Transformation**
I/O list (digital inputs/outputs)	Configuration file (JSON / Python dict)	PLC sensor and actuator data was mapped to simulation variables and component interfaces.
Component logic	Component state machines	The PLC logic was converted into state transitions with parameters and arguments.
Mission workflows	Mission scheduler	The workflow steps were converted as modular mission functions to control system flow.
Timing constraints	Timing parameters in configuration	Cycle times, delays and interlocks were extracted and parametrized for simulation timing behaviour.
Interlocks and safety rules	Guard conditions in state machines	Safety and interlock conditions were implemented as transition constraints within each component.
PLC test scenarios	Python unit tests	Expected PLC sequences were recreated as automated tests to validate model behaviour.
Event logs	Simulation event logs	Execution traces were logged and compared to PLC behaviour for verification and debugging.
Performance metrics (station cycle time, throughput, production time)	KPIs computed from logs	KPIs were derived from simulation logs to support validation and performance analysis.

4

Results

4.1 System Analysis Result

The analysis of the Drone Factory PLC documentation yielded a structured System Analysis Document encompassing four top-level systems: the Transport & Buffer System, the Transfer & Elevator Unit, the Operator Workstation, and the Control & Sensing Layer. For each system, component-level interfaces were extracted, including input/output signal definitions, operational states, and timing constraints.The complete component interface specification, including all I/O signals, states, and timing constraints, is provided in Appendix A.1.

4.2 Implementation Verification Against PLC Specifications

This section estimates the structural equivalence of the Python-based simulation model and the fidelity of the implementation procedure of the Digital Model. Throughout this section, the term *fidelity* is used to refer to the degree of fidelity with which the simulation reproduces the behaviour of its real-world counterpart.

4.2.1 Structural Fidelity

To validate the model's ability to simulate the complete functional scope of the real production line, operational Mission 1-6 from the PLC manual which defines most of the resources turn over in assembly procedure, were used as core benchmarking scenarios. These missions define the routing logic for material transfer within the automated conveyor system. They dictate six distinct operational pathways , enabling the seamless movement of items between the Delivery Side, Work Space, and Buffer areas to ensure efficient coordination and proper sequence execution throughout the entire PLC operation. Figure 4.1 and Table 4.1 establishes the traceability between the physical PLC intents and the Python simulation model's executable artifacts, confirming the structural fidelity of the model.

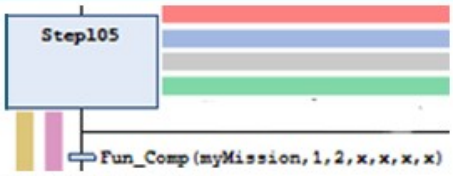
4.2.2 Behavioural Fidelity

While Section 4.3.1 confirms the structural completeness of the model, estimating the fidelity of the implementation procedure also requires validating dynamic execution. Figure 4.2 visualizes a critical material transfer sequence extracted from the simulation logs to benchmark the dynamic system state against the PLC documentation. The Gantt

From Docking Side to Elevator (Part of Mission 1)

Step 105

Condition: Fun_Comp (myMission,1,2,x,x,x,x)
Model: Make a route to WS
Code:




```
def update(self, elapsed_time: float):
        if self.state == DockingStationState.AWAITING_MISSION and
        self.transition_start_time:
                if elapsed_time >= self.time_wait_at_queue and self.allow_auto_pass:
                        print(f"Queue timeout, watchdog forces release.")
                        self.accept_route("1")
```

Step 110

Action: Conveyor_Dir_to_Wstation
Model: Calculating distance between target WS and ingress WS
Code:


```
def conveyor_travel_time_ingress_to_station(config: SimulationConfig, target_station: int) -> float:
        ingress = config.ws_ingress_station
        speed = max(config.conveyor_max_speed_ms, 1e-9)
        line_s = conveyor_travel_time_seconds(config, ingress, target_station)
        approach_s = float(config.conveyor_to_ws_loading_position_m) / speed
        return line_s + approach_s
```

Model: Sending and cleaning buffer
Code:

```
if getattr(target, "state", None).Value == "init":
        rid = buffer.pallet_rfid_at_queue
        travel_s = conveyor_travel_time_ingress_to_station(cfg, target_id)
        buffer.pallet_leaves()
        rfid_queue.clear()
        self. pending line arrivals.append((self._sim_time_s + travel_s, target_id, rid))
```

Step 120

Action: Conveyor_Dir_to_Wstation, Conveyor_on
Model: Turning conveyor and ramping up to target speed
Code:


```
if self.sim_time_s >= 3.0 and not self._did_conveyor_started:
        conveyor.start(speed_percent=50.0)
        self ._ did_conveyor_started = True
…
def start(self, speed_percent: float = 100.0):
        """Begin acceleration to target speed."""
        if self.is_malfunctioning or self.state not in [ConveyorMotorState.IDLE,
        ConveyorMotorState.RAMP_DOWN]:
                return False

        # Limit speed to valid range (20-100%)
        self.target_speed_percent = max(20.0, min(speed_percent, 100.0))

        if self.current_speed_percent == 0.0:
                self.state = ConveyorMotorState.RAMP_UP
                self.transition_start_time = datetime.now()
        elif abs(self.current_speed_percent - self.target_speed_percent) > 5.0:
                self.state = ConveyorMotorState.RAMP_UP
                self.transition_start_time = datetime.now()
        else:
                self.state = ConveyorMotorState.RUNNING
        return True
```

Figure 4.1: SFC Mission Sequence and Paring Model (with code) for Part of The Mission 1: DSide to Wspace

Table 4.1: Mission Definition and Execution Logic in Simulation Model

Mission Definition	**Operational Intent & Sequence**	**Simulation Model Executable Artifacts**	**Execution Behavior in Simulation Model**
Mission 1: DSide → WSpace	Summary: direction/speed config → main convey → elevator up → lateral transfer toward WS.	ConveyorMotor; TransferElevator; TransferDirection.TO_WS; Simulator (_ElevatorTrack, elevator_ds{sid}).	Conveyor ramp/start may occur globally early. Per active station: elevator receive ping while DS RECEIVING and elevator DOWN → DS enters PROCESSING → request_up → gate ElevatorPosition.UP → start_transfer(TO_WS) → stop_transfer.
Mission 2: DSide → Buffer	Arrive → evaluate occupancy → enter buffer.	RawMaterialIngressScheduler; DockingStation; Buffer.pallet_enters; RFIDReader; Simulator RFID/buffer steps.	Scheduler + occupancy checks → DS1 ingress handshake → RECEIVING completes → Buffer.pallet_enters() → queue/storage updated.
Mission 3: WSpace → DSide	Elevator / lateral → pull from WS → mainline egress.	TransferDirection.FROM_WS; TransferElevator; Simulator (DS SENDING branch, _pending_egress_arrivals).	While DS SENDING and elevator DOWN: start_transfer(FROM_WS) → stop_transfer. DS runs SENDING/CLEANUP → returns init → Simulator appends _pending_egress_arrivals → product_output handling.
M4: WSpace → Buffer	Pull from WS → elevator down → enter buffer.	TransferDirection.FROM_WS; request_down; ElevatorPosition.DOWN; Buffer APIs.	Primitives exist, but baseline ‘Simulator‘ does NOT wire FROM_WS → Buffer.pallet_enters() as default. Treat as component-capable / scenario extension.
Mission 5: Buffer → WSpace	Queue release → mainline travel → enter WS.	Buffer.pallet_leaves; transit_scheduled; rfid_queue; DockingStation.pallet_arrived; TransferElevator choreography.	pallet_leaves() → pending line arrival (transit_scheduled) → pallet_arrived on target DS → assembly ingress handshake → elevator receive/up/TO_WS sequence.
Mission 6: Buffer → DSide	Queue release → mainline travel → dock-side propagation.	Buffer.pallet_leaves; ConveyorMotor; Simulator (_pending_line_arrivals; optional _pending_egress_arrivals).	After buffer release: same dispatch as M5 initially. "Outfeed" is _pending_egress_arrivals / product_output after assembly completes—not the immediate next hop after pallet_leaves.

chart shows that the Python simulation model strictly adheres to the PLC's temporal and safety constraints. For example, upon receiving the PLC trigger for the Z-axis elevator motion, the system not only executes the physical lift but simultaneously forces the buffer queue into an interlocked state to prevent premature pallet release. Furthermore, the X-axis lateral transfer strictly waits for the Z-axis motion to complete before executing its 500.0 ms operation. This precise replication of sequential dependencies and safety interlocks at the micro-level proves that the devised implementation procedure restores the real production line's complex control logic into the digital model.

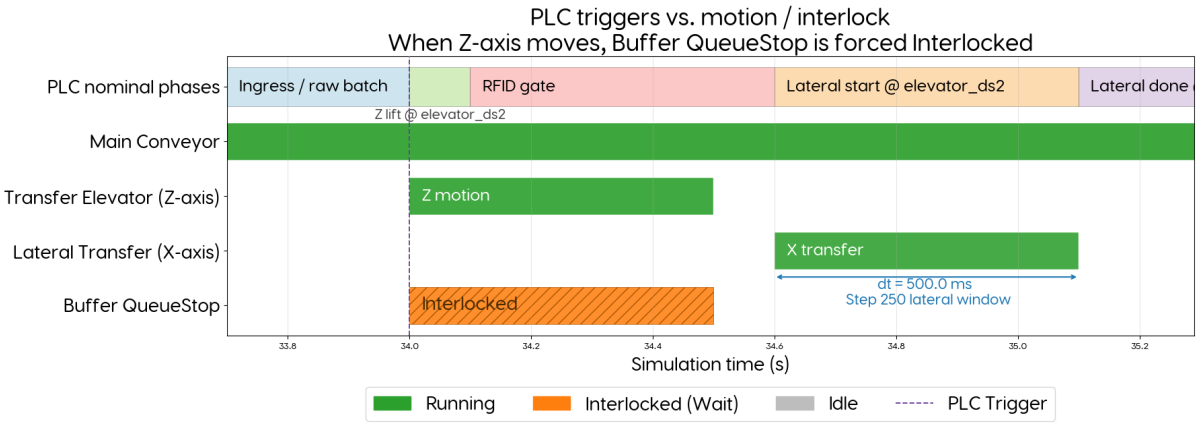


Figure 4.2: Realization of Interlock when Moving Resources from Main Conveyor to DS2

4.2.3 Model Verification Result

Under the tests list, all 57 tests completed successfully, vital information of the tests are presented in the Table 4.2. These runs cover (i) illegal local transitions and fault recovery boundaries at the component level, (ii) representative interlocks and synchronization dependencies between conveyor, elevator, buffer, docking stations, and RFID and cascade consequences of pneumatic loss and conveyor fault on downstream motion and docking/buffer acceptance, and (iii) global simulation validity and deterministic repeatability properties.

Table 4.2: Tests in Model Verification

Layer	Count	Role	Representative tests
Unit	38	Per-component state spaces and mockup transition walks	`test_motor_error_blocks_start_and_reset_returns_idle`; `test_start_transfer_is_blocked_while_elevator_moving`; `test_ingress_no_new_batch_when_line_full_and_ds_blocked`
Integration	12	Inter-module interlocks, resource contention, gating-equivalence to dependencies	`test_buffer_full_resource_interlock_stops_conveyor_and_elevator`; `test_cascade_when_pneumatic_pressure_off_blocks_all_motion_and_ds_progress`; `test_synchronization_transfer_requires_elevator_up_and_direction_match`
Simulation	7	End-to-model timing, determinism, repeatability, metrics export	`test_simulation_no_invalid_states`; `test_simulation_determinism_fixed_mode`; `test_simulation_repeatability_across_multiple_runs`
Total	**57**	—	`pytest tests/`

4.3 Flexibility and Analytical Capabilities

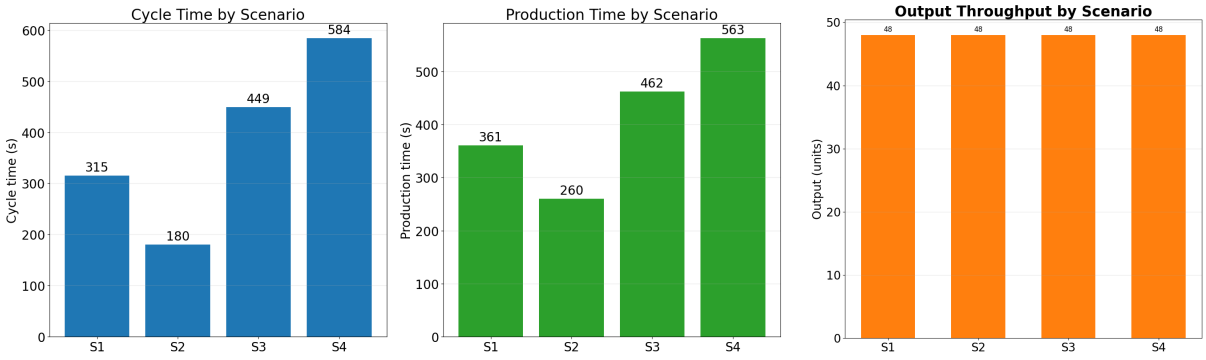
This section shows that simulation model has the ability in configuration and extensibility. It demonstrates the model's flexibility in evaluating diverse production scenarios through parameter adjustments without core code modification, alongside its capacity to generate structured logs that directly drive advanced system performance evaluation, such as identifying operational constraints, conducting bottleneck analysis and its capacity to generate structured logs which can support advanced data visualization and system performance analysis.

4.3.1 Operational Insights and Bottleneck Analysis

To demonstrate the model's analytical capability, a comparative experiment was conducted focusing on the impact of worker proficiency on assembly line performance, a well-documented source of high variability in manufacturing systems [27]. The experiment conditions are as discussed in section 3.1.7.

Figure 4.3 presents the simulation results, revealing a counter-intuitive operational pattern. As expected, the Cycle Time (product lead time) increases significantly as worker proficiency decreases, ranging from 180 seconds in the all-expert scenario (S2) to a maximum of 584 seconds in the all-beginner scenario (S4). However, despite this severe degradation in local cycle time, the system's global throughput remains strictly constant at 48 units across all four scenarios. This suggests that manual assembly speed, while impacting individual product flow, does not dictate the overarching capacity of the line.

To locate the true system constraint, a state-level utilization analysis was conducted, drawing on active-period bottleneck identification principles which link high dwell/idle



(a) Cycle Time and Production Time (b) Output

Figure 4.3: Simulation results for the four scenarios.

times to non-bottleneck processes[28]. An examination of the state distribution logs for the most critical all-beginner scenario (S4) reveals that the manual assembly stations (DS2, DS3, and DS5) spend approximately 73% to 74% of their time in the 'init' (idle or starvation) state. Their actual 'processing' (active) time only accounts for roughly 26% to 27% of the total duration, while the downstream QC station (DS6) exhibits an idle time of 94.5%(see Table 4.3).

Table 4.3: State distribution proportion of components in Scenario 4 (all-beginner)

Component ID	Init	Processing	Receiving	Sending
ds_2	0.739	0.259	0.001	0.001
ds_3	0.734	0.264	0.001	0.001
ds_5	0.727	0.271	0.001	0.001
ds_6	0.945	0.054	0.001	0.000

These results provide a definitive bottleneck analysis: even when the assembly line is populated all by the slowest operational workers, the assembly stations suffer from severe starvation. Consequently, the manual workstations do not dominate the system's throughput limit. The true operational bottleneck resides further upstream—likely constrained by the initial material feed rate or global conveyor pacing. This demonstrates the Python model's value in isolating hidden operational constraints beyond superficial cycle time metrics.

4.3.2 Structured Output and Advanced Analytical Support

The simulation model's analytical value is presented on its ability to generate structured logs that serve as a direct data source for advanced industrial analytics. The model records every state transition and event at the second level into a standardized JSONL format.
Listing 4.1 shows a part of simulation output. Each entry includes a timestamp, the event type, component identity and metadata such as state transitions or calculated metrics. This structured approach allows for automated, programmatic parsing using standard data science libraries such as Pandas and Matplotlib.

Listing 4.1: Part of the Simulation Log, including State Changes and Event Records

```
[SIM 07:45:38] [state_change] [ds_5] processing -> sending
[SIM 07:45:40] [state_change] [ds_5] sending -> cleanup
[SIM 07:45:40] [state_change] [ds_5] cleanup -> init
[SIM 07:45:40] [event] [ds_5] station_cycle_done
[SIM 07:45:40] [event] [align_ds_5] iface_window
[SIM 07:45:40] [event] [conveyor_line] egress_transit_scheduled
[SIM 07:45:40] [state_change] [rfid_elevator] identified -> idle
[SIM 07:45:52] [event] [ds_6_egress] product_output
[SIM 07:46:00] [performance_metric] [system_global] metrics
   throughput=47 lead_avg_s=383.97446808328584 wip=0
```

The visualizations generated in this study are based on this output style. The Gantt charts Figure 4.2 and the comparing analysis in Figure 4.3 were produced by scripts processing these raw logs.

4.4 Model Validation Result

This section discusses about the model validation experiment results that was followed according to research question 3 proposal. Discussion to research question 3 is discussed in detail in section 5.2.

4.4.1 Validation Experiment Result

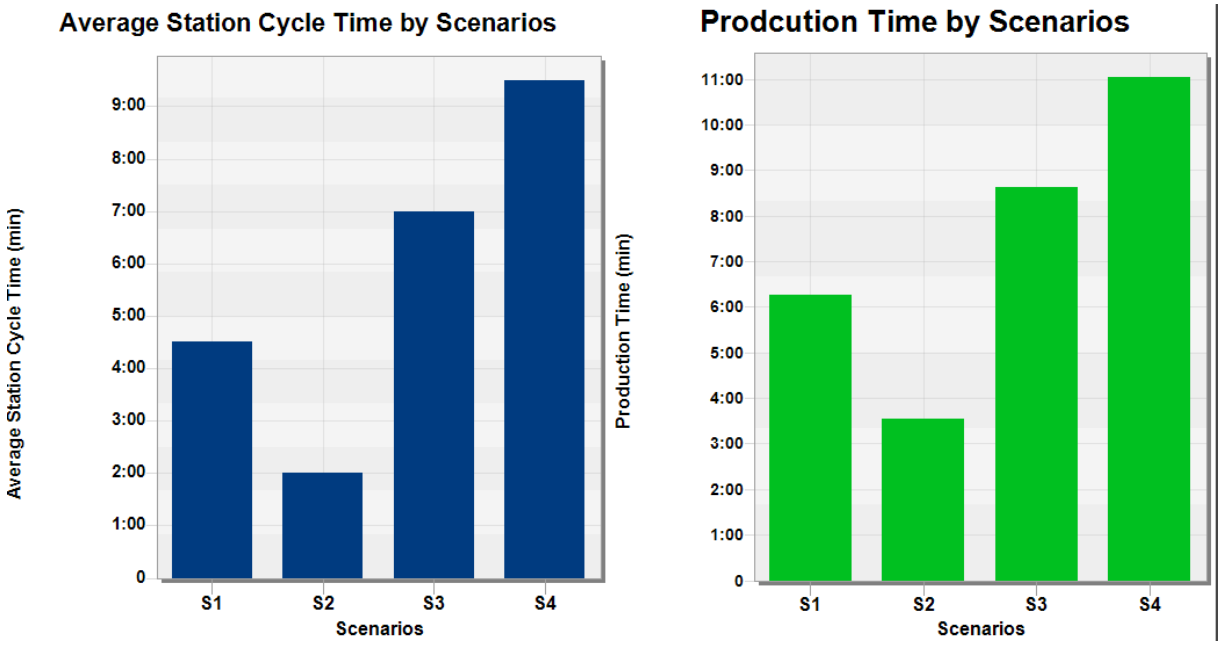


Figure 4.4: Plant Simulation Result: The Cycle Time and overall Production Time plotted against the four scenarios.

Test Result Variables: Production time, Average Station cycle time and Throughput were considered for analysing. From the Figure 4.3 and Figure 4.4 the graphs obtained from

both the digital models representing Production time and Average Station Cycle Time against 4 Scenarios show very similar pattern. Same amount of product throughput 48 products were produced in both the models in 8 hour long simulation.

For comparing the test variable data the percentage similarity formula was used:

$$\text{Similarity } (\%) = 100 - \left(\frac{|A-B|}{\frac{A+B}{2}} \times 100 \right)$$

Analysing the test comparison data for Production time (as recorded in table 4.4) we can see that all the scenarios showed a similarity percentage of 80 and above. Scenario 2 had the lowest similarity percentage while, Scenario 1 shows 96% similarity. The average similarity in the production time between python simulation model and plant simulation model is 87%. Upon comparing the Station Cycle Time data for each scenario (as recorded in table 4.5) Scenario 2 shows the lowest similarity of only 60% while scenario 4 shows 98% similarity. This wide fluctuation in result has appeared due to the irregularities in the boundary conditions of station processing time between both models. Due to limitations of replicating all the model characteristics to plant simulation model certain station specific features like processing time distribution and station recovery time were implemented differently. The average similarity in the station cycle time is obtained as 84% between python simulation model and plant simulation model.

Table 4.4: Production Time Comparison

Scenario	Python model	Plant simulation	Similarity (%)
Scenario 1	361 seconds	376 seconds	96%
Scenario 2	260 seconds	212 seconds	80%
Scenario 3	462 seconds	518 seconds	88%
Scenario 4	563 seconds	662 seconds	84%

Table 4.5: Station Cycle Time Comparison

Scenario	Python model	Plant simulation	Similarity (%)
Scenario 1	315 seconds	270 seconds	85%
Scenario 2	180 seconds	120 seconds	60%
Scenario 3	450 seconds	420 seconds	93%
Scenario 4	584 seconds	570 seconds	98%

5
Discussion

This chapter discusses the simulation model project in relation to the three research questions. It explains implementation fidelity with the PLC documentation, the validation of the Python simulation model, and the benefits of the chosen approach. The discussion also assesses the project's limitations and the scope within which its conclusions hold. In the end, future possibilities for further development of the simulation model are discussed.

5.1 Discussion on the Fidelity of the Implementation Procedure

Research Question 2: **How accurately can the component logic, timing dependencies and mission workflows be translated from a PLC document into a modular Python simulation architecture?**

What needs to be emphasised is that the original PLC specification is an operational engineering manual for a moveable workstation conveyor system, covering physical topology, I/O mappings, HMI interactions, and sequential control logic[3]. Verification therefore does not compare the model directly against live PLC traces; instead, it compares the implementation against a reference baseline obtained by systematically extracting and conceptually reconstructing these fragmented specifications into missions, state machines, and interlock rules, an approach commonly utilized in conceptual model validation for digital twins[4].

Comparing with this baseline, the Python model achieves strong structural and behavioural fidelity within the validated mission set. Structurally, the architecture supports traceable mapping from Missions 1-6 to executable components (Table 4.1), with Mission 4 implemented at component level but not fully orchestrated in the baseline Simulator. Behaviourally, log-based timelines in representative transfer scenarios (Figure 4.2) show that micro-level timing dependencies and resource interlocks for example, lateral transfer gated on elevator position are preserved during dynamic execution,demonstrating white-box validity at the component level[29].

Other differences, such as global conveyor ramp timing relative to per-station Mission 1 episodes or phased egress semantics in Mission 6, reflect differences in abstraction rather than arbitrary coding errors. As established in simulation fidelity literature, exact replication is often unnecessary if the abstraction meets the intended operational purpose[30]. Comparing with the manual, the twin combines component state machines with a mission-oriented scheduling layer to support long-term, configurable simulation and structured trace analysis. This trade-off accepts bounded semantic shifts at the orchestration layer

while preserving necessary logics.

5.2 Discussion on Validation Methodology for the Python Simulation Model

Research Question 3: **How can the Python simulation model be verified and validated using PLC documentation variables and comparison with a DES model?**

The verification and validation approach used in the project proves that the python simulation model was reliable and behaved in the intended manner. The multistage approach confirmed accurate component interactions and ensured that the model's logic matched the PLC documentation. Model stability was ensured with 24-hour long simulation runs and revealed no deadlocks, timing drifts or memory related issues.

Operational validity was confirmed through an offline cross model comparison with Siemens Tecnomatix Plant Simulation model. Both the models showed similar behaviour in KPI graphs. In depth comparison of KPI's like total production time and throughput showed similar values, while values of station cycle time showed large variations for different scenarios. The validation experiment and results are discussed in detail in Section 3.1 and Section 4.4.

Due to technical limitations in adopting the exact logic of python model to plant simulation model, many boundary conditions had to be assumed. The python simulation model used an exponential variation in worker efficiency but in the plant simulation model it was constant. Ramping times of conveyor were also exclude from validation experiment. Even though 40 independent runs were done for each scenario, due to these limitations the station cycle time data could not be accurately validated.

Considering the limitations in the validation model the average similarities between both models were above the 80% threshold that was agreed for operational validation. Hence, the python model is assumed to be validated.

The takeaway here is that the effectiveness of validation techniques used was limited due to these conditions and could be improved if the python model was accurately traced to the validation model.

5.3 Benefits of the Python Simulation Model

Research Question 1: **Identify the benefits and trade-offs of using a Python based Digital Simulation model over commercial DES tools for this drone factory.**

The discussion of benefits in this section is predicated on the structural and behavioural fidelity established in Sections 5.1 and 5.2. Given the approximate three-month scope of this project, the following comparative claims against traditional plant experimentation and commercial discrete-event simulation (DES) tools are illustrative and project-informed.

The first advantage is the lowered licensing barrier for academic testbed environments. Commercial DES packages often rely on proprietary solvers and graphical interfaces bound by commercial licenses. In contrast, using an open-source Python stack provides an accessible foundation for academic research and iterative development [31]Relative to traditional approaches that rely predominantly on physical trials or on-site PLC commissioning, the twin offers repeatable offline experiments, it reduces the need for repeating field validation.[32, 33]

The second benefit lies in architectural transparency and modularity. While Python can host standard DES libraries such as SimPy, as discussed in the literature on mixed discrete–continuous simulation frameworks[21], this project specifically uses a step-driven simulation engine. Where a traditional DES model might encode material flow using generic modules, this twin is based on same terms with PLC(e.g., DockingStation, TransferElevator) that remain transparent and unit-testable. Furthermore, through the SimulationConfig module, parameters such as cycle times, workstation routing, assembly-duration distributions, and worker efficiency settings can be adjusted without altering core control logic, enabling configuration-driven experimentation. [34]

In terms of simulation model integration, published work often emphasises combining shop-floor data with other platforms such as Plant Simulation[17, 35]. Conversely, this project demonstrates seamless integration with data pipelines at the programmatic level. By generating structured JSONL events documented in the project logging specification, the model acts as a configurable data generator that fits Python-based analysis pipelines[36], including Jupyter notebooks for Gantt alignment with PLC narratives and interval validation to supporting repeatable KPI extraction and bottleneck analysis.

While commercial DES tools excel with interactive, graphical user interfaces that facilitate visual exploration[37], the Python twin has a advantage in experimentation efficiency. Pytest support long-horizon batch runs without three-dimensional rendering, reducing the complex reliance in modelling the scenarios.

The Python simulation model has distinct strengths in logical transparency, configuration flexibility, and log-driven, repeatable analysis. Its boundaries must be acknowledged: it trades clear 3D visualisation and possible extreme-scale entity throughput of commercial DES for more programmatic control and analytical reproducibility [38] within the scope established in Sections before.

5.4 Limitations

Several limitations are concluded in this chapter. First, constrained by the three-month scope and parameter limitations—such as representing manual operations with idealised normal distributions—the comparative benefits discussed in Section 5.3 remain illustrative engineering estimates rather than formal economic proofs or long-horizon operational data. Second, with respect to Research Question 2, fidelity was assessed exclusively through offline cross-model validation against a reconstructed baseline derived from the PLC technical manual and from simulation traces[4]. The study did not calibrate the

twin against all live controller trends or physical line measurements, meaning the system currently constitutes a "Digital Model" rather than a fully integrated "Digital Twin". Furthermore, as some of the performance parameters provided by experimenter maybe under an ideal circumstances, logical fidelity within documented scenarios must therefore not be interpreted as validated online shop-floor performance. Third, the model scope is intentionally control- and material-path-centric: dynamic AGV routing, mechanical degradation, realistic sensor noise, and bi-direction real-time operational technology integration were not represented[39][40], although Section 5.3 described soft integration via structured logs, that coupling remains at the offline data-interface level, limiting conclusions strictly to high-level process logic. Fourth, implementation coverage is incomplete relative to every manual narrative such as: Mission 4 is not fully simulated in the baseline simulator, and bounded semantic differences remain for Mission 1 global conveyor energisation and Mission 6 phased egress. The claims beyond documented PLC equivalence are strictly bounded to this validated scenario set, and extrapolating them would require additional work[41]..

5.5 Future Scope

Although the model performs well within the scope of this study, there are several areas that could need more research. The simulation model can be expanded with adaptive scheduling strategies and visualisation tools to make it easier for non-technical users to understand system behaviour. There is also scope to implement AI enabled features like bottleneck prediction and scenario based production management supports[12]. The validation approach in the current study is only an offline approach. Once the drone factory is operational, there is scope for validating the model online directly with the factory through real time PLC connections.[41] Since, the model developed is still in the early stages of digital twin progression, the next research should focus on how to evolve the digital model to a digital shadow and finally to a complete digital twin[8]. Another important area is cybersecurity. As simulation models move closer to real factory systems and begin exchanging data with PLCs, HMIs, or cloud-based analytics, the potential for data breach also increases. Future research in the field should also be consider on secure data channels, authentication layers, and protection against manipulation of configuration files or mission parameters of such simulation models. Apart from this, possibility is to integrate large language models into the digital-twin workflow, not as a replacement for the simulation logic but as an assistive layer for interpreting logs, generating scenario configurations, or supporting operator-level decision explanations. This type of integration could help bridge the gap between raw simulation output and human-readable insights. Large language models can also be developed to generate digital simulation models by accepting simple prompts from the users. Together, these developments would strengthen the model's realism, resilience, and long-term usefulness for both research and industrial applications.

6
Conclusion

This final chapter concludes the project work followed by discussion on the main takeaways and a summary of future works.

6.1 Conclusion

This thesis set out to build a Python-based simulation model from PLC documentation and investigate whether it could accurately represent the behaviour of a semi-automated drone assembly line to serve as a foundation for a future Digital Twin. By reconstructing the factory's physical logic, timing dependencies, and PLC-driven workflows inside a modular Python architecture, the study demonstrated that a lightweight, open-source simulation environment can replicate the functioning of a real production system. The model's performance was evaluated through structural verification, behavioural accuracy checks, and cross-validation against Siemens Plant Simulation, confirming that the proposed approach is both technically feasible and operationally reliable. These are the Main Takeaways from the project:

- The Python model successfully reproduced all major PLC-documented behaviours, including conveyor logic, docking-station routing, workstation timing, and interlock conditions. The simulation engine handled mixed discrete continuous behaviour without requiring a fixed time-step loop, improving computational efficiency.

- Structural verification confirmed that all components behaved according to the PLC specification, including timing constraints, sensor interactions and mission based pallet path. Behavioural accuracy tests showed that the Python model's cycle times and production times aligned closely with expected PLC logic. Cross validation with Siemens Plant Simulation produced average accuracy of 87% similarity in production time performance and 84% in average station cycle time. Though station cycle time performance varied between 60% to 98% depending on complexity of scenario.

- Scenario switching in the validation experiment (like the worker proficiency variations) required only parameter updates, not structural changes, proving the model's modularity. The simulation produced structured logs that supported advanced analysis, including cycle-time variations, bottleneck identification, and scenario based performance comparison. The architecture supports integration with external systems such as IoT devices, real-time dashboards, and Digital Twin communication layers.

- Python's open-source ecosystem provides a cost-effective alternative to commercial DES tools while still supporting complex industrial logic. The model can be extended into a real Digital Twin by connecting it to PLC signals, sensor data streams, or cloud-based monitoring systems. The approach is scalable and can be adapted to other manufacturing systems with similar characteristics.

The results of this thesis show that Python can serve as a powerful and flexible tool for developing Digital Twin simulation models. Future work can build on this foundation by integrating real time data exchange, enabling bi directional communication with PLCs and incorporating cybersecurity layers to protect industrial communication channels. The architecture can also be enhanced with machine learning, automated scenario generation using large language models and hybrid simulation with physics based digital twins. These advancements would move the system toward a fully autonomous, intelligent Digital Twin capable of monitoring, predicting, and optimizing production processes in real time.

Bibliography

[1] S. Mihai, M. Yaqoob, D. V. Hung, W. Davis, P. Towakel, M. Raza, M. Karamanoglu, B. Barn, D. Shetve, R. V. Prasad *et al.*, "Digital twins: A survey on enabling technologies, challenges, trends and future prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291, 2022.

[2] C. Bae, E. Choi, and S. Lee, "Technologies, applications, and challenges of digital twin across industries: A systematic review of the state-of-the-art literature," *IEEE Access*, 2025.

[3] Sven Ekered, *Program code, WS Conveyor SII-Lab*, Chalmers University of Technology, SII Lab, Lindholmen, Gothenburg, Sweden, 2022, unpublished internal technical document.

[4] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the 2010 Winter Simulation Conference*, 2010, pp. 166–183.

[5] T. R. Wanasinghe, L. Wroblewski, B. K. Petersen, R. G. Gosine, L. A. James, O. De Silva, G. K. I. Mann, and P. J. Warrian, "Digital twin for the oil and gas industry: Overview, research trends, opportunities, and challenges," *IEEE Access*, vol. 8, pp. 104 175–104 197, 2020.

[6] L. Li, B. Lei, and C. Mao, "Digital twin in smart manufacturing," *Journal of Industrial Information Integration*, vol. 26, p. 100289, 2022.

[7] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *Ifac-PapersOnline*, vol. 51, no. 11, pp. 1016–1022, 2018.

[8] E. M. Martinez, P. Ponce, I. Macias, and A. Molina, "Automation pyramid as constructor for a complete digital twin, case study: A didactic manufacturing system," *Sensors*, vol. 21, no. 14, p. 4656, 2021.

[9] M. W. Grieves, *Digital Twins: Past, Present, and Future*. Cham: Springer International Publishing, 2023, pp. 97–121. [Online]. Available: https://doi.org/10.1007/978-3-031-21343-4_4

[10] D. Zhong, Z. Xia, Y. Zhu, and J. Duan, "Overview of predictive maintenance based on digital twin technology," *Heliyon*, vol. 9, 03 2023.

[11] M. Kumbhar, A. , and S. Bandaru, "A digital twin based framework for detection, diagnosis, and improvement of throughput bottlenecks," vol. 66, pp. 92–106, 02 2023.

[12] M. Subramaniyan, A. Skoogh, J. Bokrantz, M. A. Sheikh, M. Thürer, and Q. Chang, "Artificial intelligence for throughput bottleneck analysis–state-of-the-art and future directions," *Journal of Manufacturing Systems*, vol. 60, pp. 734–751, 2021.

[13] S. H. Jacobson and E. Yucesan, "Common issues in discrete optimization and discrete-event simulation," *IEEE transactions on automatic control*, vol. 47, no. 2, pp. 341–345, 2002.

[14] L. Junmin, Z. Shouqi, Z. Lianjiong, and D. Fuguang, "Corba-based discrete event simulation system," *Journal of Systems Engineering and Electronics*, vol. 12, no. 3, pp. 13–19, 2001.

[15] E. Negri, L. Fumagalli, and M. Macchi, "A review of the roles of digital twin in cps-based production systems," *Procedia manufacturing*, vol. 11, pp. 939–948, 2017.

[16] T. Wu, B. Sui, D. Du, Z. Han, and F. Zhai, "A model combining discrete event system simulation and genetic algorithm for buffer allocation in unreliable large production lines," *Tsinghua Science and Technology*, vol. 9, no. 3, pp. 363–368, 2004.

[17] A. S. Bangsow and S. O. Service, *Manufacturing Simulation with Plant Simulation and Simtalk : Usage and Programming with Examples and Solutions.* Springer Berlin Heidelberg, 2010.

[18] T. J. Sego, "Simservice: a lightweight library for building simulation services in python," *Bioinformatics*, vol. 40, 01 2024.

[19] T. E. Oliphant, "Python for scientific computing," *Computing in Science Engineering*, vol. 9, pp. 10–20, 2007. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4160250

[20] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Computing in Science Engineering*, vol. 13, p. 9–12, 03 2011. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5725235

[21] N. Matloff, "Introduction to discrete-event simulation and the simpy language," 2008.

[22] "Simpy dev documentation," simpy.readthedocs.io. [Online]. Available: https://simpy.readthedocs.io/en/latest/

[23] N. Karanjkar and S. M. Joshi, "A python-based mixed discrete-continuous simulation framework for digital twins," *arXiv preprint arXiv:2208.01408*, 2022.

[24] T. Naumovic, M. Despotovic-Zrakic, B. Radenkovic, L. Zivojinovic, and I. Jezdovic, "Development of a continuous system simulation engine in python programing language," in *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH).* IEEE, 2020, pp. 1–5.

[25] E. Laftchiev, X. Sun, H.-A. Dau, and D. Nikovski, "Anomaly detection in discrete manufacturing systems using event relationship tables."

[26] U. Dahmen, T. Osterloh, and J. Roßmann, "Structured validation of ai-based systems by virtual testing in simulated test scenarios," *Applied Intelligence*, vol. 53, no. 15, pp. 18 910–18 924, 2023.

[27] T. Baines, H. Lightfoot, O. Benedettini, and J. Kay, "The servitization of manufacturing: A review of literature and reflection on future challenges," *Journal of Manufacturing Technology Management*, vol. 20, pp. 547–567, 06 2009.

[28] M. Subramaniyan, A. Skoogh, H. Salomonsson, P. Bangalore, and J. Bokrantz, "A data-driven algorithm to predict throughput bottlenecks in a production system based on active periods of the machines," *Computers Industrial Engineering*, vol. 125, pp. 533–544, 11 2018.

[29] S. Robinson, "Simulation model verification and validation," *Proceedings of the 29th conference on Winter simulation - WSC '97*, 1997.

[30] C. Kober, V. Adomat, and M. Ahanpanjeh, "Digital twin fidelity requirements model for manufacturing," *Proceedings of the Conference on Production Systems and Logistics: CPSL 2022*, 2022.

[31] G. Dagkakis and C. Heavey, "A review of open source discrete event simulation software for operations research," *Journal of Simulation*, vol. 10, pp. 193–206, 08 2016.

[32] *Virtual Commissioning of Manufacturing Systems a Review*, 06 2010.

[33] C. G. Lee and S. C. Park, "Survey on the virtual commissioning of manufacturing systems," *Journal of Computational Design and Engineering*, vol. 1, pp. 213–222, 07 2014.

[34] A. Law, *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 01 2014.

[35] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia CIRP*, vol. 61, pp. 335–340, 2017.

[36] F. Tao, J. Cheng, Q. Qi, M. Zhang, H. Zhang, and F. Sui, "Digital twin-driven product design, manufacturing and service with big data," *The International Journal of Advanced Manufacturing Technology*, vol. 94, pp. 3563–3576, 03 2018.

[37] J. Göbel, P. Joschko, A. Koors, and B. Page, "The discrete event simulation framework desmo-j: Review, comparison to other frameworks and latest development," 05 2013.

[38] F. Perez and B. E. Granger, "Ipython: A system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, pp. 21–29, 2007.

[39] J. Wang, J. Moreira, Y. Cao, and R. B. Gopaluni, "Simultaneous digital twin identification and signal-noise decomposition through modified generalized sparse identification of nonlinear dynamics," *Computers Chemical Engineering*, vol. 177, p. 108294, 09 2023.

[40] Y. Tang, D. M. Rahmani, and W. G. Gary, "Capturing lifecycle system degradation in digital twin model updating," *arXiv (Cornell University)*, 03 2025.

[41] G. Lugaresi, S. Gangemi, G. Gazzoni, and A. Matta, "Online validation of digital twins for manufacturing systems," *Computers in Industry*, vol. 150, pp. 103 942–103 942, 09 2023.

A
Appendix

A.1 Appendix: System Architecture

Table A.1: System Components, Interfaces, and States

Top-level System	Component	Inputs	Outputs	States
Transport & Buffer System	Main Conveyor Motor	conveyor_on, conveyor_speed, conveyor_ramp	conveyor_running, motors_warning, output_current	STOPPED, RAMPING, RUNNING, ERROR
	Buffer Queue Stop	buffer_queue _stop_down	buffer_queue _stop_sensor, QueueStop-Down_Status	BLOCKED, WAITING, PASSING
	Buffer Pallet Sensors	Physical Pallet (Presence)	pallet_sensor_1, pallet_sensor_2, PalletsInBuffer	IDLE, ACTIVE
Transfer & Elevator Unit	Transfer Elevator	elevator_up, elevator_down	elevator_sensor _up, eleva-tor_sensor_down, Grp_ElevPos	DOWN, UP, MOVING, ERROR
	Cross-Transfer Belt	transfer_on, transfer_dir_from _WSpace	Transfer_Status _Icon, se-quence_Blocked	IDLE, INBOUND, OUTBOUND
	Docking Interface Logic	Mission, RFID_String	myMission, Allow_ Timer_For _Passing	READY, BUSY, PASSTHROUGH
Operator Workstation	Adjustable Worktable Motor	table_up, table_down	Physical Movement	STATIONARY, MOV-ING_UP, MOV-ING_DOWN

Continued on next page

Table A.1 – *Continued from previous page*

Top-level System	**Component**	**Inputs**	**Outputs**	**States**
	Height Sensor Logic	table_height_sensor, Offset, Scale_Factor	Actual_ Height_mm	NORMAL, CALIBRATING
	HMI Interface	Select Mission, Toggle Manual Mode, Set Parameters	System Status Icons, RFID Data Display, Alarm/Warning Indicators	OVERVIEW, MISSION_SELECT, MANUAL_IO, SETTINGS
Control & Sensing Layer	Soft PLC Core	Distributed I/O, HMI Commands, RFID Raw Data	Actuator Commands, Processed Data, State Variables	INIT, RUNNING, STOPPED
	RFID Data Processor	Raw_Bytes, Trigger	Out_String, Read_OK	Processing (Logic-based)
	Mission Dispatcher	Mission_Select, PalAtQueue, AllowTimerForPassing	myMission, Sequence_Blocked	IDLE, ASSIGNED, COMPLETED

A.2 Appendix: Hmi Panel Functionalities

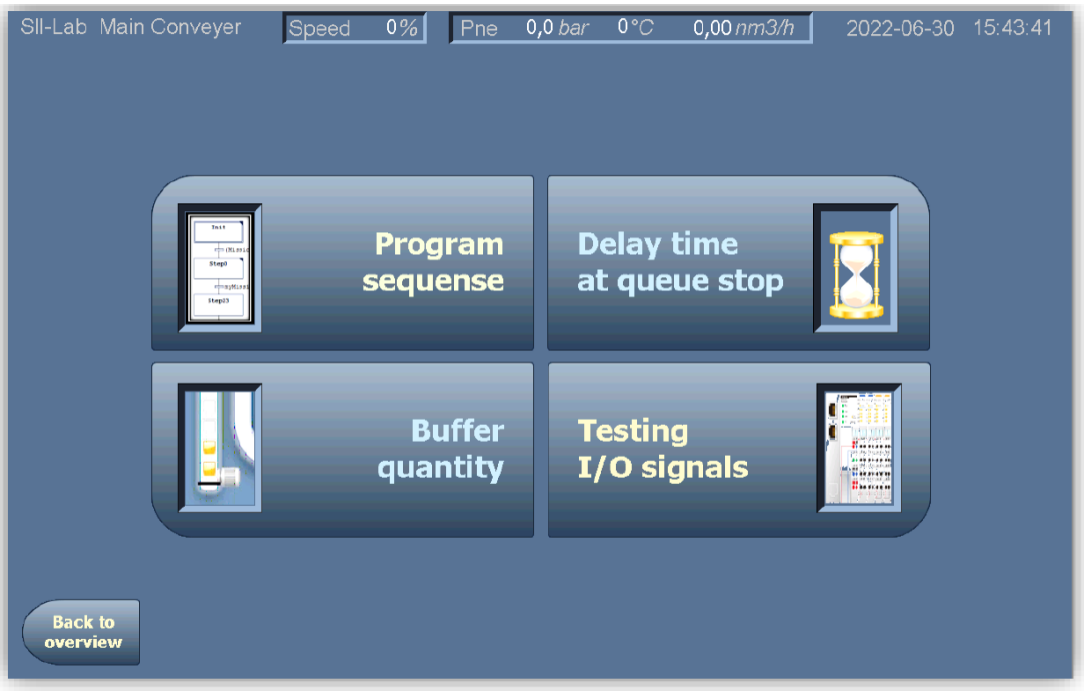


Figure A.1: Functions available in the HMI Panel adapted from internal document "Program Code"[3].

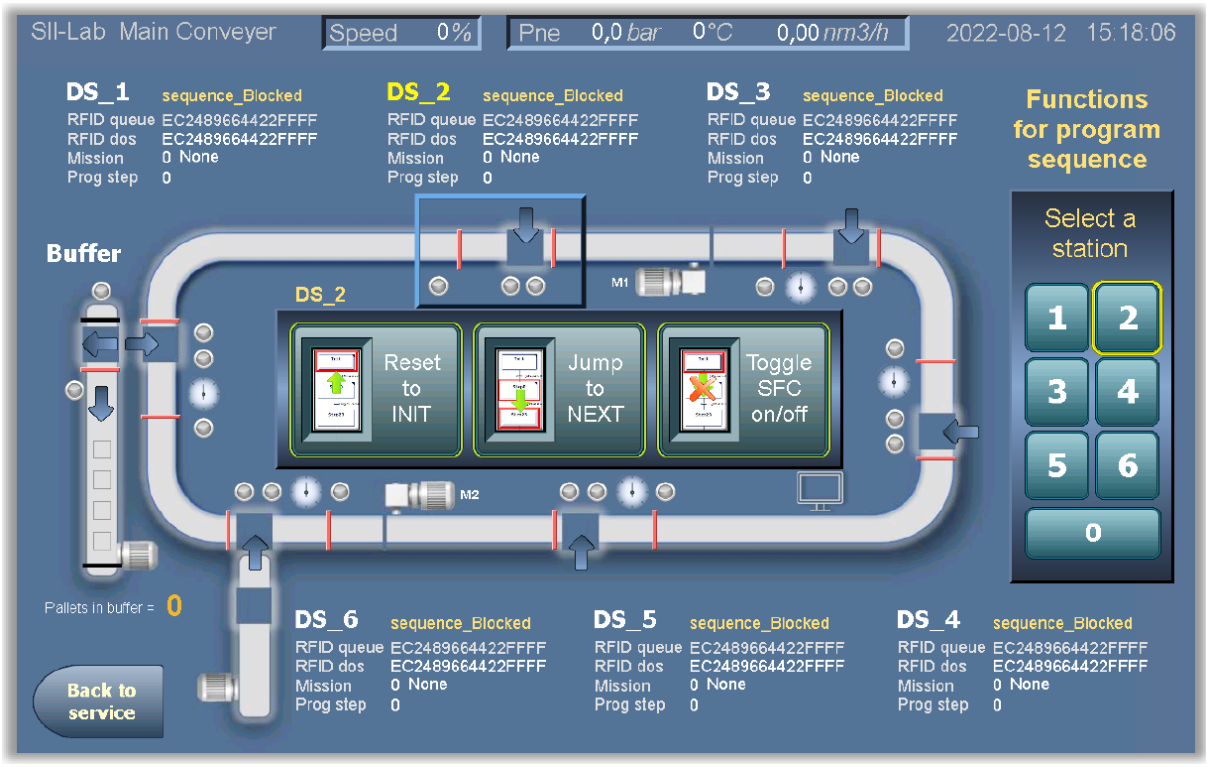


Figure A.2: Program Sequence Functions available in the HMI Panel[3].

Figure A.3: Workstation Table Functions available in the HMI Panel[3].

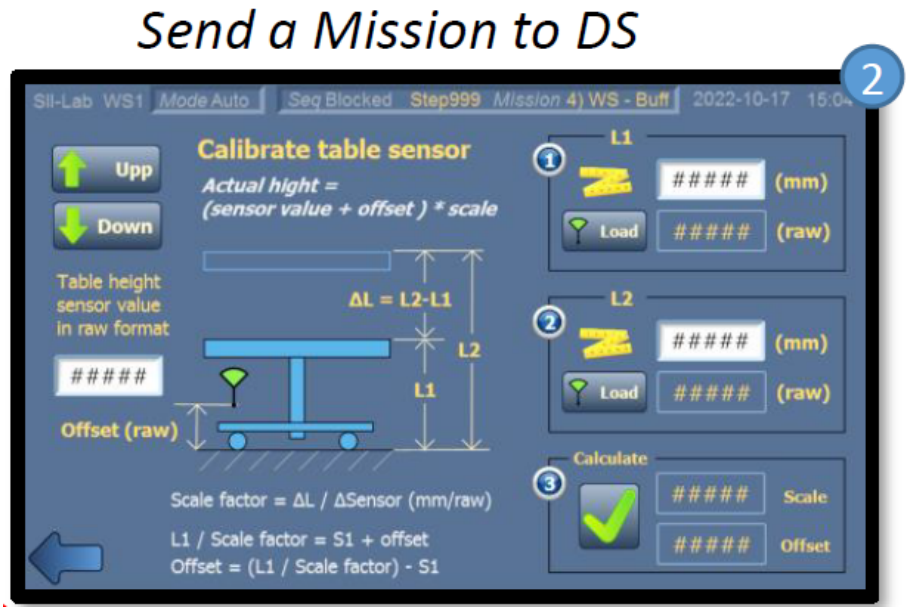


Figure A.4: Mission Dispatcher Functions available in the HMI Panel [3].

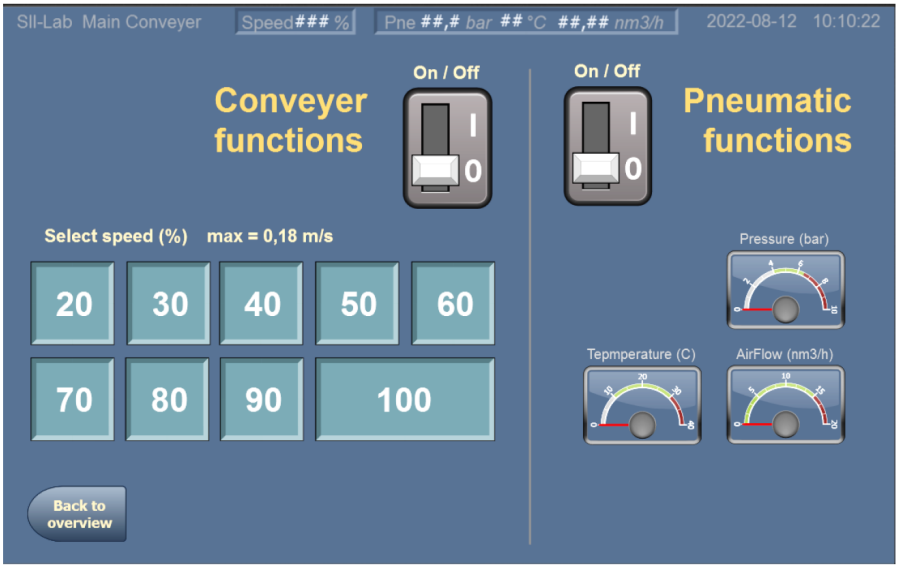


Figure A.5: Pneumatic Functions available in the HMI Panel[3].

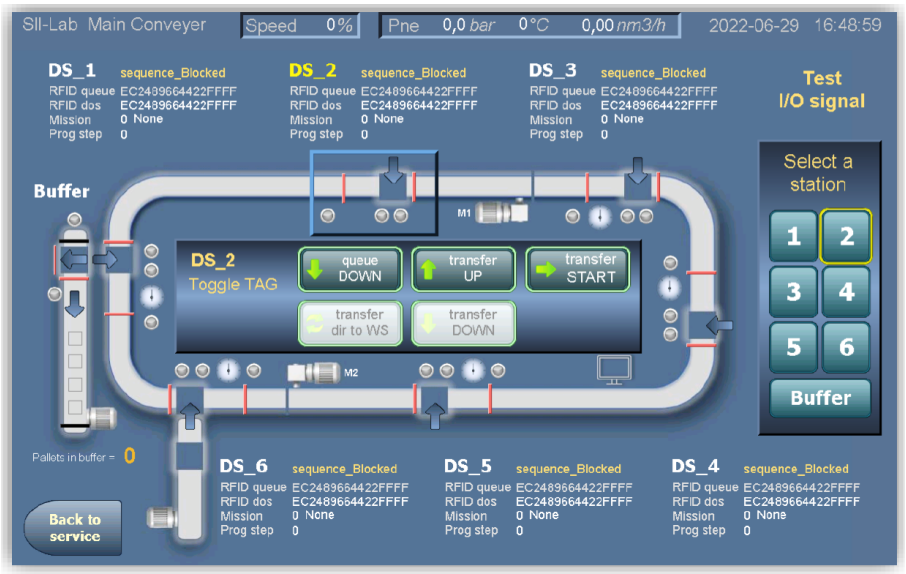


Figure A.6: I/O Signale Test Functions available in the HMI Panel[3].

DEPARTMENT OF INDUSTRIAL AND MATERIAL SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY