Conceptual design of a pipe routing system on a pipelaying vessel

A discrete event simulation study at Allseas

ME54035: MSc Thesis

Luuk Sijm

TU Delft

Allseas

Conceptual design of a pipe routing system on a pipelaying vessel

A discrete event simulation study at Allseas

by

Luuk Sijm

to obtain the degree of

Master of Science (MSc)

at the department Maritime and Transport Technology of the faculty Mechanical Engineering of Delft University of Technology, to be defended publicly on the 21st of October 2025.

Student number:	4719786	
Master programme:	Mechanical Engineering - Multi-Machine Engineering	
Thesis committee:	Ir. M.B. Duinkerken (Mark)	TU Delft, chair
	Dr. Ir. X.L. Jiang (Xiaoli)	TU Delft, committee member
	Ir. R. Leite Patrão (Rafael)	TU Delft, committee member
	D. Bujakiewicz-Baars (Dominik)	Allseas Engineering B.V., supervisor
	J. Ramlakhan (Jermaine)	Allseas Engineering B.V., supervisor
Project duration:	March, 2025 – October, 2025	
Report number:	2025.MME.9113	

TU Delft

Summary

Allseas is renovating the double joint factory (DJF) on their pipelaying vessel *Solitaire* and aims to fully automate the pipe handling equipment, which performs the transport of the single lengths of pipe (joints) on deck. To achieve this, the company intends to implement a pipe routing system that replaces the current human-operator controlling routing decisions. Although the equipment layout has been defined, the routing strategy to ensure a continuous supply of double joints to the firing line, while simultaneously minimizing the turnaround time for pipe supply vessel, remains uncertain. This challenge is defined by this research as the pipe routing problem.

This research demonstrates how the development of a conceptual pipe routing system, implemented within a Discrete Event Simulation (DES) model, provides insight into the pipe handling process and the effects of different routing strategies. The study concludes by identifying a most effective routing strategy for the case study.

The study begins by analysing the renovated configuration of the pipe handling system to identify its key components, operations, and the integration of the pipe routing system within the overall control structure.

A literature review then establishes the methodology for modelling and solving the pipe routing problem. A comparative assessment identifies Salabim as the most suitable DES software for this study. The routing problem is formulated as a sequential decision-making problem within a model-free Markov Decision Process (MDP) framework, enabling future application of Reinforcement Learning (RL) for optimization. In the present study, however, the routing problem is solved through rule-based heuristics that define the routing strategy.

The DES model is developed following the methodology for simulation model development proposed by Robinson 2004, with pipeline specific data based on a 36 inch diameter pipeline from the South East Extension project. Due to insufficient availability of accurate process duration data for the DJF, the required distributions are derived from a self-conducted data analysis.

The model is subsequently used to evaluate six base route configurations (a-f) under three different pipe delivery rates (slow, average, fast) from port side. These configurations are further improved through interactive search experimentation, with performance assessed primarily by pipe transfer crane waiting time.

The results show that route configuration (f) performs slightly better under fast delivery rates and is therefore identified as the most effective routing strategy. Nevertheless, all configurations except (d) achieved satisfactory performance, with limited potential for further improvement given the defined key performance indicators.

The heuristic approach thus demonstrates its effectiveness in automating sequential decision-making for the pipe routing problem and can be adapted to alternative routing strategies. However, a key limitation remains the requirement for full scenario coverage, as the programmer must anticipate and encode all possible scenarios during the design phase.

AI statement

For this Master thesis for the course ME54035, I have used Generative AI to:

- obtain inspiration for the overall structure of the report;
- improve the grammar, style, layout, and/or spelling of the text;
- create (part of) the code in Python for solving the problem.

In all cases I have reviewed and corrected the work and remain fully responsible for the content of the report.

Terminology and Abbreviations

Terminology

Terminology	Definition
beadstall	first welding station in the firing line
double joint factory	working plane where two single joints are welded together to obtain a double joint
firing line	working plane where double joints are welded to the main pipeline
hold crane	overhead crane that arranges joints within a hold area
intralogistics	logistics of internal material and/or information flow within a company facility
joint	single length (12 m) of steel pipe
NP-hard	not solvable in polynomial time
pipe flow	collective movement of joints
pipe routing problem	scheduling and routing of the pipe flow

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
DES	Discrete Event Simulation
DJF	Double Joint Factory
GPL	General Public License
GUI	Graphical User Interface
KPI	Key Performance Indicator
MAPF	Multi-Agent Path Finding
MDP	Markov Decision Process
NDT	Non-Destructive Testing
OR	Operations Research
OS	Open Source
PDL	Programming Descriptive Language
PPD	Pipe Production Department
PSV	Pipe Supply Vessel
PTC	Pipe Transfer Crane
RL	Reinforcement Learning
SEE	South East Extension (project)
TC	Transverse Car

Contents

1 Introduction

This chapter provides the background for conducting this study and introduces the problem statement, the research objective with corresponding research questions and the approach taken to address them.

1.1. Background

Allseas is a world-leading contractor in the offshore energy market, operating a versatile, in-house designed fleet of specialized vessels. Their pipelay vessel Solitaire is due for a large renovation during which all almost all pipe handling equipment on the main deck and in the Double Joint Factory (DJF) will be renewed. The DJF is the first welding factory where single lengths of steel pipe (joints) are welded together to form a double joint. The renovation project is currently in the design phase and is planned to be finished at the end of 2026. Mechanically the design is converging, but the development of the software to manage the pipe flow (collective movement of joints) is still to be started. The new configuration of pipe handling equipment will increase the complexity of the pipe flow, which is currently controlled by operator decision-making. This raises the need for a pipe routing system that automates the decision-making process to autonomously regulate the pipe flow.

Although a previous simulation study within Allseas has been conducted to determine the equipment layout, there is no clear vision yet on how the pipe routing system will be formulated and how the joints should move through the system. The tight renovation schedule causes a lack of time and resources for Allseas to properly investigate opportunities for optimization of the pipe routing system.

There are no comparable automation studies available in the literature. The pipelaying industry is known to be relatively conservative (Charlton et al. 2020), and pipe handling operations in general have not followed other batch and process industries such as the automotive industry in terms of adopting new technologies (Angelle 2020). An explanation for this could be that, although pipe handling involves repetitive tasks, the exact operations depend on the specific processes and are carried out in challenging environments (Moralez et al. 2020). Nevertheless, recent studies (Angelle 2020, Moralez et al. 2020, Craig et al. 2023) show progress toward more advanced automated pipe handling systems aimed at reducing human intervention. This trend is especially relevant because the safety risks for personnel involved in pipe handling operations remain significant (Monacchi et al. 2023).

1.2. Problem definition

As mentioned, Allseas aims to automate the decision-making process relating to the pipe flow on deck by implementing a pipe routing system. This system would solve the scheduling problem (when to move a joint), routing problem (where to move a joint) and resource allocation problem (how to move a joint) of the pipe flow, which is collectively defined as the pipe routing problem. The main objective of the pipe routing system is to ensure that the welding factory for the main pipeline (firing line) is continuously fed with the required double joints. Another important element of the pipe handling operations is the turnaround time for Pipe Supply Vessels (PSV), which can be minimized by regulating the pipe flow in the pipe yard to allow for continuous delivery of joints by the pipe transfer cranes (PTC).

The optimal solution to the pipe routing problem might be influenced by the main pipeline currently being laid, pipe delivery, the stochastic processing times of individual stations, travel times between stations, potential downtime and potential weld rejects. As mentioned, Allseas does not have the time and resources in the planned renovation project to develop an advanced pipe routing system that adapts to all these system dynamics. The company would still like to have an insight on the effects of different routing strategies and the feasibility of a fully automated pipe routing system.

Therefore, a conceptual pipe routing system should be formulated and evaluated using a model of the pipelaying operations. Both the model and the pipe routing system have to be formulated from scratch. A prior literature study revealed a research gap as their were no studies found from comparable intralogistics systems that operate with similar control structure and system dynamics that can be used as a direct example. The combination of the various pipelaying operations, the movement restrictions of the pipe handling equipment and the control structure with the pipe routing system functioning as a centralized decision-maker regulating all pipe movements make that the suitable OR methods for modelling and solving similar problems have to be customized to align with the pipe routing problem.

1.3. Research objective and approach

This research aims to evaluate the feasibility of a fully automated pipe routing system that regulates joint movements on a pipelaying vessel during pipelaying operations. The research objective is to formulate a conceptual design for this pipe routing system and evaluate its performance for different routing strategies using a model of the pipe handling system on *Solitaire*. This will be done by addressing the following research question:

How can the joint movements on a pipelaying vessel be automatically regulated to obtain a continuous supply of double joints to the firing line, whilst considering the pipe supply vessel turnaround time?

To answer this main research question, the study will be divided into four stages in which the following subquestions will be examined:

1. What are the key components and operations of a pipe handling system on a pipelaying vessel that are relevant to the pipe routing system, and what does the control structure including the pipe routing system look like?
2. How can suitable OR methods be applied and adapted to effectively model and solve the pipe routing problem?
3. What does a minimal viable model of the pipe handling system look like that can effectively evaluate the performance of a pipe routing system during pipelaying operations?
4. How does the conceptual pipe routing system perform for different routing strategies and how could this performance be further improved?

The following chapters of this report are organized as follows. Chapter 2 describes a system analysis of the pipe handling system on *Solitaire* based on The Delft Systems Approach by Veeke et al. 2008 to define the system specifications and boundaries for this study. Chapter 3 provides a brief literature study, expanding on the findings of a the previous literature study, to establish the approach to model the system defined in Chapter 2 and the adapted methods to model and solve the pipe routing problem. Chapter 4 describes the development of the model that emulates the system during pipelaying operations with integration of the pipe routing system. The model will be developed using a systematic approach described by Robinson 2004. Chapter 5 applies the developed model to experiment with different routing strategies to identify the most effective configuration found for the conceptual pipe routing system, and highlight opportunities for further improvement. Chapter 6 summarizes the findings by addressing all research questions, while Chapter 7 presents both scientific and practical recommendations for future work.

2

System analysis

This chapter describes a system analysis of the pipe handling system on *Solitaire* to define the system for this research. First the methodology used to perform the system analysis will be described, followed by a brief overview of all the pipelaying operations performed on *Solitaire*. Then the general pipe handling system on *Solitaire* is defined, followed by a more detailed analysis of the pipe yard subsystem. For the pipe yard subsystem, also the control structure is defined with implementation of the pipe routing system.

2.1. Methodology

The pipe handling system will be defined using The Delft Systems Approach by Veeke et al. 2008.This section will explain the principles of this approach, that are used to structure the remainder of this chapter and define the pipe handling system considered in this project.

In The Delft Systems Approach, a system is described as a collection of elements that interact with each other. Elements are the smallest components of the system that are considered by the researcher to achieve his goal. Each element has its own attributes and the collection of elements is referred to as the content of the system. Relationships between elements define the interactions between them and these interactions can change the attributes of an element. The collection of relationships is referred to as the internal structure of the system. At last the environment is defined as the collection of elements outside of the system that have relationships with elements within the system. These relationships are referred to as the external structure.

Complex systems can be differentiated into subsystems and aspectsystems. A subsystem is a partial collection of the element, whereby all relationships remain. An aspectsystem describes the entire content of a system, but only with a partial collection of the relationships.

2.2. Pipelaying operations on Solitaire

This section provides a brief overview of the pipe laying operations on *Solitaire*. As stated in the introduction, *Solitaire* is a pipelaying vessel with two factories: the DJF and the firing line. In the DJF, single joints are welded together to form double joints, which are then supplied to the firing line. In the firing line, the double joints are welded together to form the main pipeline. At the end of the firing line, the main pipeline is lowered into the water until it reaches the seabed.

All pipelaying operations on *Solitaire* have been detailed in a previous study using an IDEF0 diagram. A copy of this diagram is shown in Figure 2.1. The top level functions A_0 and A0 are shown on top of this figure, with the lower level functions A2, A3 and A5 being further refined below.

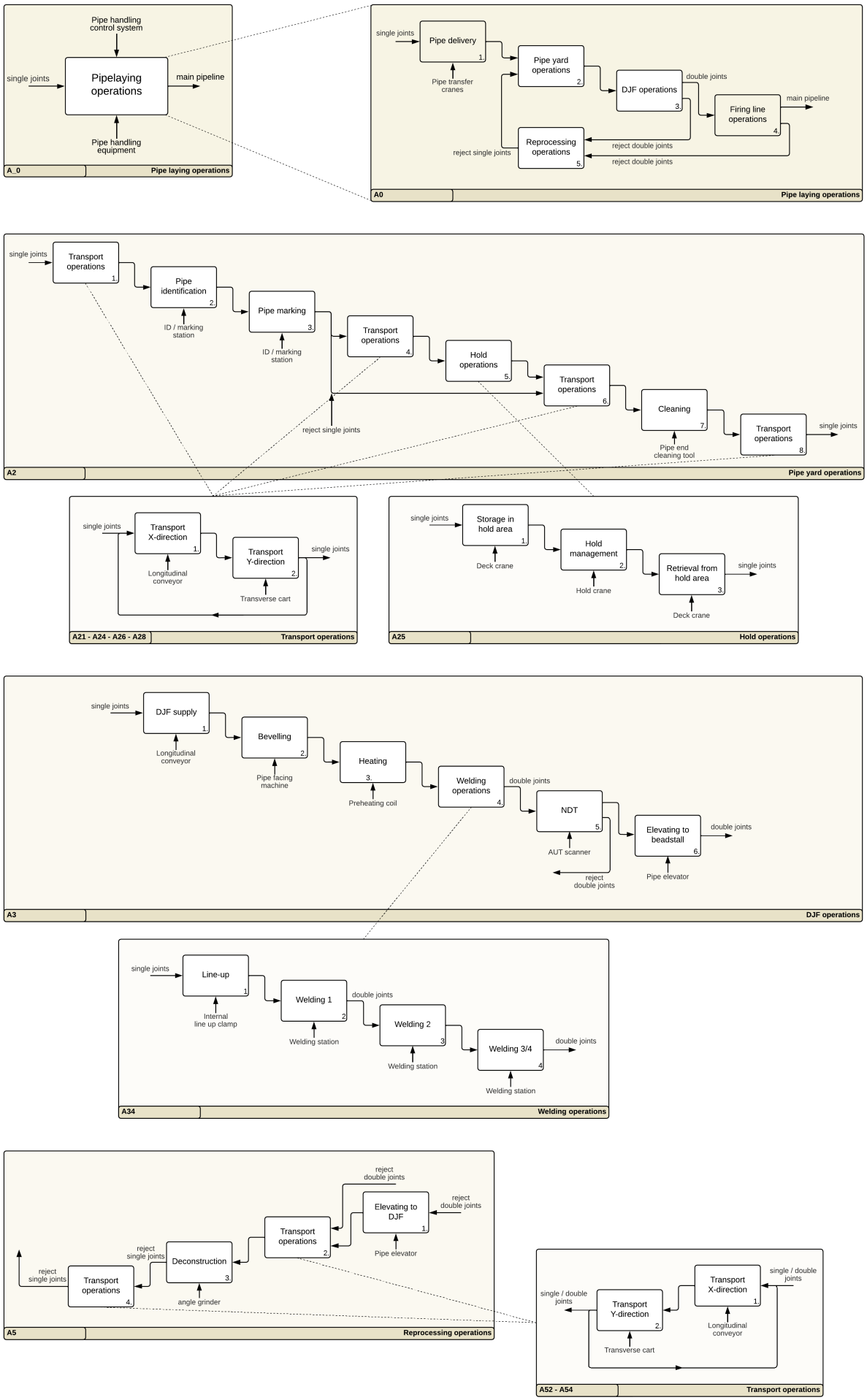


Figure 2.1: IDEFO diagram for pipelaying operations on *Solitaire* (Copy from a prior literature study

Figure 2.1 shows that the pipe laying operations can be divided in five subsystems; A1: pipe delivery, A2: pipe yard operations, A3: DJF operations, A4: firing line operations and A5: reprocessing operations. The single joints are supplied by Pipe Supply Vessels (PSVs) and delivered by the deck-mounted pipe transfer cranes (PTCs) on *Solitaire*. After delivery the joints need to be identified and marked in the pipe yard. From the marking station, the joints can be transported to the holds for temporarily storage or they can directly pass the cleaning station to afterwards be transported to the DJF. In the DJF the joints pass the bevelling and pre-heating station before they can be welded together to form a double joint. The weld of the double joint is checked at a non-destructive testing (NDT) station before it can be supplied to the first welding station of the firing line (beadstall) through the pipe elevator. Here the double joint is welded to the main line and this weld is checked as well. In case of a weld defect, the reject double joints are transported to a return conveyor where it is split into single joints again. From here, the reject single joints can re-enter the process in the pipe yard.

It can be noted that A1: pipe delivery and A4: firing line operations are transparent in Figure 2.1 as these subsystems are not part of the pipe handling system considered in this project, but they do have relationships with elements of the pipe handling system.

It is clear that the objective of the pipelaying operations is the installation of the main pipeline, which is worked on continuously in the firing line. All other operations are subordinate to this objective and should work in coordination to ensure a steady supply of double joints to the firing line, thereby preventing any downtime in the operation due to a lack of resources. Aside from downtime in the firing line operations, the most significant factor affecting operational costs is pipe delivery. This is due to the high operating costs of PSVs, which can be limited by minimizing the turnaround time of a supply operation. However, pipe delivery is also dependent on the pipe yard operations, which must facilitate continuous PTC operations by clearing the delivery position.

2.3. Pipe handling system

This section describes the content and structure of the pipe handling system considered in the DJF renovation project. As noted in the previous section the pipe handling system considered in the DJF renovation project covers functions A2, A3 and A5 of the IDEF0 diagram in Figure 2.1. This means it covers all the pipe handling operations between the pipe delivery on deck and the double joints being lowered to the firing line, with potential reprocessing operations. The PSVs, the PTCS, and the firing line can be seen as elements of the environment.

Figure 2.2 shows a black box model of the pipe yard system. Single joints are delivered by the PTC, then handled by the pipe handling system, and at the end of this system they are delivered as double joints to the firing line through the pipe elevator. To regulate the operations within the pipe handling system, the system requires a pipe routing system. The primary key performance indicator (KPI) of the system is the firing line waiting time, as the system is required to ensure a continuous supply of double joints to the firing line. Any waiting time therefore indicates a failure to deliver the required double joints on time. A secondary KPI is the PTC waiting time, which arises when the delivery position is not cleared in time to enable the PTC to continuously deliver joints. This delay is directly associated with an increase in the PSV turnaround time.

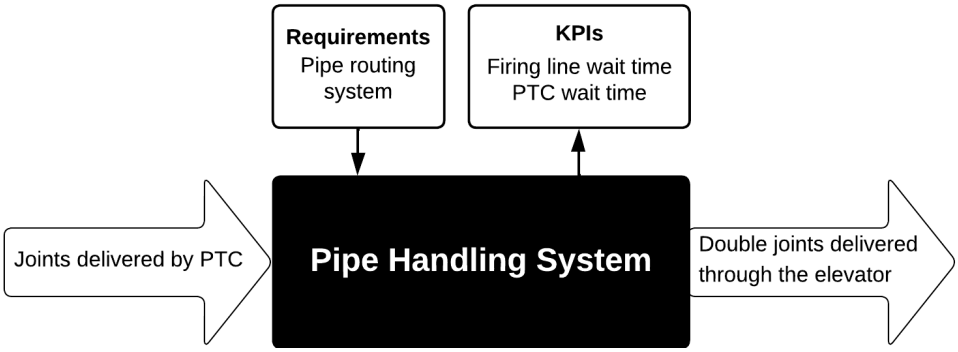


Figure 2.2: Black box model of the pipe yard system

2.3.1. Content of the pipe handling system

The content of the pipe handling system is summarized in Table 2.1. Figure 2.3 shows a photo of the current configuration of the pipe yard on Solitaire featuring equipment comparable to the elements of the new pipe handling system. The remainder of this section provides a description for all elements.

Table 2.1: Content of the pipe handling system

Element	Type	Function	Task
Joint	Resource		
Anode Joint	Resource		
Double Joint	Product		
Anode Double Joint	Product		
Longitudinal Conveyor (LCO)	Transport equipment	Pipe movement	Park and move resources in longitudinal direction
Fixed Pipe Support (FPS)	Transport equipment	Pipe movement	Park resources
Transverse Cars (TC)	Transport equipment	Pipe movement	Transport resources in transverse direction
Deck crane	Transport equipment	Pipe movement	Lift resources to and from holds
Hold	Subsystem	Pipe movement	Store resources temporarily
Marking Station	Process station	Resource processing	Mark resources with identification number
Cleaning Station	Process station	Resource processing	Clean pipe ends of resources
Bevelling Station	Process station	Resource processing	Bevel pipe ends of resources
Heating Station	Process station	Resource processing	Pre-heat pipe ends of resources
Welding 1 Station	Process station	Production	Line-up two resources and connect with external weld
Welding 2 Station	Process station	Production	Lay second external weld
Welding 3/4 Station	Process station	Production	Lay final external and internal weld
NDT Station	Process station	Product testing	Check weld quality of product
Overhead crane	Transport equipment	Pipe movement	Lift products to and from Buffer
Buffer	Storage location	Product storage	Store products temporarily
Pipe elevator	Transport equipment	Product delivery	Lift products to firing line

Figure 2.3: Current configuration of the pipe yard on *Solitaire* ©Allseas

Resources and Products

Figure 2.4: Joints

The resources moving through the pipe handling system are single lengths of steel pipe with a length of approximate 12m and a diameter ranging between 0,2 and 1,5 m. This project considers two types of resources: normal joints and anode joints. Anode joints are fitted with a sacrificial anode that prevents corrosion of the pipeline by cathodic protection. In the DJF resources are welded together to form the products: double joints. If one the resources used is an anode joint, the product becomes an anode double joint. The anode double joints are incorporated in the main pipeline at regular intervals.

Longitudinal Conveyor

Figure 2.5: LCO

The LCOs can park resources and move them in longitudinal direction to adjacent LCOs. In practice this element consist of individual rollers, but here the LCO is referred to as the set of rollers that form a parking spot for the resources. The LCOs also allow for TCs to travel between support points to pick up a resource for transverse transport.

Fixed Pipe Support

Figure 2.6: FPS

The FPSs are their to park joints in positions where there is no longitudinal movement required. Again here the FPS is referred to as the set of supports that form a parking spot for resources and TCs are able to travel between the supports for transverse transport.

Transverse Car

Figure 2.7: TC

The TCs are rail mounted machinery that can transport resources in transverse direction by lifting the joints and travelling between the LCO and FPS supports. Both the transport and lifting speed of the TC can be considered constant. All TCs are configured as dual units with two mounted on a single rail. This provides redundancy, but also requires the TCs to coordinate their movements during operation.

Deck crane

Figure 2.8: Deck crane

The deck cranes are rail mounted overhead cranes that lift joints from an LCO onto an FPS in the hold. There are two deck cranes, meaning two holds can be used simultaneously for either storage or retrieval of joints.

Hold

Figure 2.9: Hold

The hold is defined as a subsystem as in practice it consists of multiple elements that are not explicitly considered in this project. The hold system supports the resource flow by temporarily storing resources when the supply is higher than the demand of the DJF, and supplying stored resources when the supply is lower than the demand. The hold system consist of six different holds.

Marking Station

Figure 2.10: Marking

The marking station automatically reads the supplier identification number of a resource and then paints the identification on the outside of the resource. The pipe handling system is configured with two marking stations located next to eachother. Figure 2.10 shows the marking tool in a test setup. As the marking station has not been used in practice there is no accurate data available on the duration time, but it should have a small variation as the marking station will be a fully automated procedure.

Cleaning Station

Figure 2.11: PECT

The cleaning station uses the pipe end cleaning tool (PECT) shown in Figure 2.11 to scrub the ends of a resource with a wire brush while the resource is rotated. The pipe handling system is configured with two cleaning stations located next to eachother. Again the PECT shown in the figure has not been used in practice so there is no accurate data available on the duration time.

Bevelling station

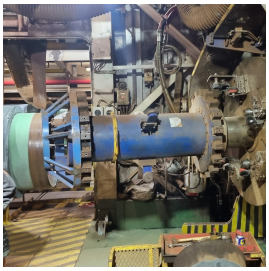


Figure 2.12: Bevelling

At a bevelling station the pipe ends are bevelled by a pipe facing machine. There are four bevelling stations with a total of eight pipe facing machines. Currently the pipe facing machines are handled by an operator and for now it is unclear if this operation will be fully automated.

Heating station

Figure 2.13: Heating

After bevelling the pipe ends are pre-heated for welding by a heating coil at the heating station. This operation is very fast compared to the other operations and can be handled by one of the welding operators from welding station 1. Also for this operation it is unclear if this will be fully automated.

Welding stations

Figure 2.14: Welding

A welding line consists of three welding stations with multiple human operated and (semi-) automated operations. Welding station 1 is supplied with two resources which are aligned and welded together. At welding station 2, the weld is being expanded on the outside. At welding station 3/4 the weld is finished on both the outside and inside of the pipe to form a finished product. The DJF is configured with two welding lines that are both supplied from the same LCO positions in the pipe yard.

NDT Station

Figure 2.15: AUT

The NDT station checks the weld quality of the products using an Automated Ultrasonic Testing (AUT) scanner as shown in Figure 2.15. If the product does not pass the test it will be deconstructed and transport back to the pipe yard from the end of the DJF. The DJF is configured with two NDT stations located next to eachother in line with the welding lines. Although the scanner operates automatic, it is mounted and controlled by a human operator.

Overhead Crane

Figure 2.16: Overhead crane

In the waiting area another overhead crane is present that transfers joints within the waiting area. Currently this crane is equipped with lifting hooks, but these will be replaced by stabbing pins for more flexibility.

Buffer

Figure 2.17: Buffer

When products have passed the welding test, but they can not directly be supplied to the firing line, they are placed in the double joint buffer by means of an overhead crane. As shown in Figure 2.17 the products lay on top of eachother in the buffer location, so not all products are directly accessible to the overhead crane.

Elevator

Figure 2.18: Firing line

The pipe elevator transfers joints from the waiting area to the beadstall with a one-dimensional motion. The pipe elevator is currently supplied by the overhead crane from the waiting area, but after the renovation it can also be supplied by a TCs with extended arms.

2.3.2. Content of the pipe handling system environment

Table 2.2 summarizes the content of the environment and the remainder of this section provides a description of all elements.

Table 2.2: Content of the pipe handling system environment

Element	Type	Function	Task
Pipe Supply Vessel (PSV)	Resource supply	Supply	Supply resources
Pipe transfer crane (PTC)	Transport equipment	Transport	Deliver resources from a PSV to the pipe handling system
Firing line	Subsystem	Production	Produce main pipe line out of products

Pipe Supply Vessel

Figure 2.19: PSV

A PSV supplies joints from shore to ship. The PSV is emptied by a PTC from Solitaire.

Pipe Transfer Crane

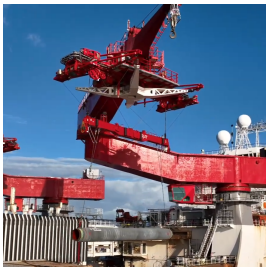


Figure 2.20: PTC

Solitaire is equipped with two PTCs that deliver the resources from a PSV onto a fixed position in the pipe yard. The PTCs are human operated and also requires assistance from personnel on deck to secure and release the resource.

Firing line

Figure 2.21: Firing line

The firing line is another welding line defined as a subsystem where products are welded together to form the main pipeline. The firing also includes multiple welding stations and an NDT station. As products are welded together, a weld rejection necessitates the deconstruction of the six products that have already been welded after the rejected weld. The firing line is supplied from the pipe handling system by the pipe elevator. Reject products may be retrieved directly by the overhead crane.

2.3.3. Structure of the pipe handling system

Figure 2.22 shows the structure of the pipe handling system, including all elements described in the system content. As shown, the system can be divided in four different zones: the pipe yard, the DJF, the waiting area and the return line. These zones correspond to the operations defined in Figure 2.1 with the waiting area belonging to the DJF operations.

Internal structure

The figure highlights relative positions of all the system elements and their internal connection to each other. The layout of the system can be seen as an irregular grid where the coloured cells represent the transport equipment, either an LCO or an FPS, that can hold resources or products. The joints are longitudinal in X-direction and the connection between cells shows how they can move along the grid.

The joints can directly move in X-direction on the LCOs. For movement in Y-direction the joint requires the TCs that are positioned under the cells to lift both pipe ends, travel to another vertical aligned cell and lower both pipe ends again. During this transport in Y-direction it cannot pass any other joints on its path.

Figure 2.22 also highlights the positions of the the process stations, which are placed on top of the transport equipment that park the joints at the stations so they can be processed. After the process station is finished it releases the joint so it can be moved on by the transport equipment. All process stations are featured twice in the system and all joints should pass each type of process station in the order as described.

The holds and the buffer are special cells on the grid that can store multiple resources or products. Resources can reach the holds from the neighbouring LCOs by one of the deck cranes that can move itself along the return line in X-direction. Products can reach the buffer from the end of the DJF by the overhead crane, that can also lift products from here onto the return line.

External structure

The external structure of the pipe handling system is not explicitly shown in Figure 2.22. However, the FPSs marked as delivery position indicate the position where the pipe handling system is connected to the PSVs by means of the PTCs. Furthermore, the pipe elevator connects the pipe handling system to the firing line and when rejected double joints are sent back by the firing line they can be retrieved by the overhead crane through the indicated hatch to the firing line.

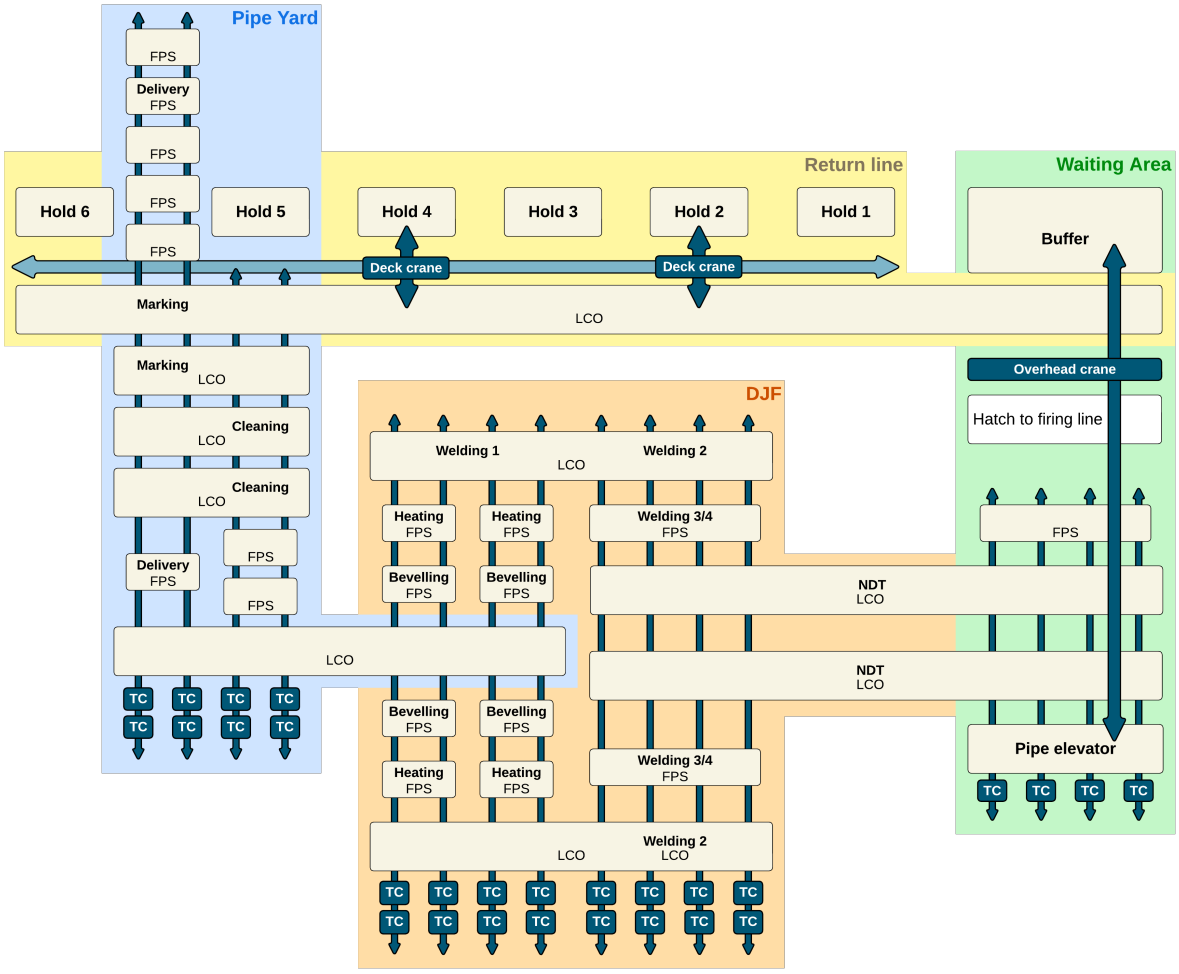


Figure 2.22: General structure of the pipe handling system

From Figure 2.22 it can be concluded that only the grid structure in the pipe yard and the waiting area actually allows for different routing strategies. The adjacent stations of the welding lines in the DJF have a one-directional pipe flow. As this research focusses on routing strategies within the pipe handling system, the scope of this research will henceforth be limited to subsystem A2: pipe yard operations, while accounting for its interaction with the other elements of the pipe handling system. This will from now be referred to as the pipe yard system.

2.4. Control structure of the pipe yard system

Figure 2.23 outlines the control structure of the pipe handling system. The pipe routing system is seen as an intermediate system that can be implemented into the existing control structure. Currently, all transport equipment is controlled by the pipe handling plc and the stations and subsystems each have their own control system. All control systems are connected to a database that receives pipe and station information from all pipelaying operations. In the existing structure, human operators assign the joint movements to be performed by the pipe handling PLC. As human operators only assign destinations to joints and do not control the actual movements, this human decision making can be directly substituted for the pipe routing system that automates the decision-making process. The pipe routing system receives pipe and station information from the database and translates this information into pipe movement requests for the pipe handling PLC.

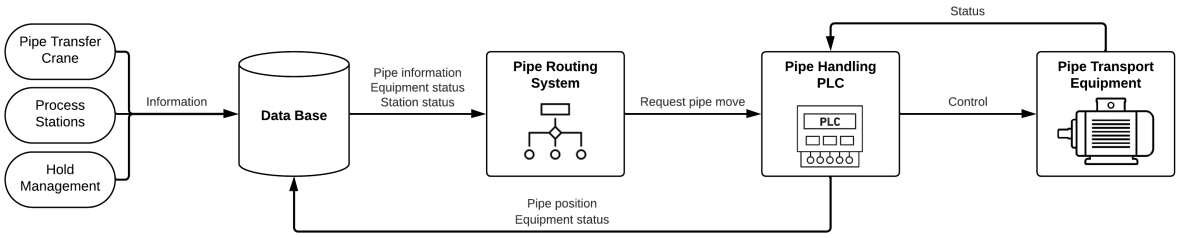


Figure 2.23: Pipe yard system control structure

2.5. Conclusion

In this chapter, all pipelaying operations on Solitaire have been described and divided into subsystems following the IDEF0 function modelling methodology as shown in Figure 2.1. Although the DJF renovation project focusses on the pipe handling system consisting of functions A2: pipe yard operations, A3: DJF operations and A5: reprocessing operations, it is concluded that the scope of this study will be limited to function A2: pipe yard operations, which from now on will be referred to as the pipe yard system. Furthermore, it is concluded that the pipe routing system is implemented in the control structure of the pipe yard system as an intermediate system connected to the database containing all pipe and station information. The pipe routing system translates the pipe and station information into pipe movement requests for the pipe handling PLC and hold management operators.

3

Methodology for modelling and solving the pipe routing problem

This chapter establishes the methodology for modelling and solving the pipe routing problem. The chapter first describes the findings of a prior literature study on OR methods in comparable intralogistics systems, which are then further developed for application in this study.

3.1. Prior literature study

This section provides an overview of the prior literature study and its conclusions, followed by the application of the identified methods in this study.

3.1.1. Overview of the literature study

The prior study defines the pipe routing problem as a combinatorial problem of determining the path for each joint (routing) and scheduling the joint movements together with the other pipe handling operations. Since research on automated pipe handling systems is limited, a literature review was conducted on OR methods applied in comparable intralogistics systems. The pipe handling system is compared to an automated container terminal with integrated scheduling as all operations within the system cooperate to achieve a shared main objective. More specifically, the operations of the TCs in the pipe yard are compared to twin operating yard cranes in the container terminal as they are both part of a larger system with similar objectives and movement restrictions. Additionally the routing challenge of multiple moving joints in the pipe yard is compared to the routing of Automated Guided Vehicles (AGVs) in intralogistics systems. A literature review was conducted on all these comparable intralogistics systems and revealed an abundance of methods to model and solve both routing and scheduling problems. The review concludes the study with an overview of all identified methods and their applicability to the pipe routing problem.

3.1.2. Conclusions of the literature review

The literature review concluded that the pipe handling operations should be modelled using simulation as this is typically used to model a system when it involves stochastic behaviour, which makes it too complex to be analysed using only mathematical models (Hillier et al. 2015). This is applicable to the pipe handling system because of the dynamic infeed of joints and the stochastic behaviour of the pipelaying operations.

The identified routing and scheduling methods are evaluated based on their underlying principles to assess their suitability for the pipe routing problem. The literature review identified Reinforcement Learning (RL) as the most promising approach, as it employs the model-free Markov Decision Process (MDP) framework to represent the problem as a sequential decision-making process, closely aligning with the key principles of the pipe routing problem. Moreover, RL can adapt to unforeseen scenarios (Filom et al. 2022) and may reveal previously unrecognized system relationships (Wuest et al. 2016).

Also Moralez et al. 2020 concludes Artificial Intelligence (AI) will play an important role in the continuous improvement of automated pipe handling systems.

However, RL is also inherently complex. (Filom et al. 2022) claims its industrial application remains limited due to its sensitivity to training data and the need for extensive offline training. The same study promotes for more research exploring how and when RL can replace conventional OR, or how the methods could complement each other. Since there is currently no benchmark performance for the pipe routing system, the immediate priority is to achieve automation through sufficiently effective solutions. Therefore, the literature review proposes less complex methods to model and solve the pipe routing problem at the current stage, and considers optimization of the pipe routing system as a later stage.

The literature review also concludes the multi-agent path finding (MAPF) problem to have a lot of similarities with the pipe routing problem. MAPF finds paths for multiple agents in a shared environment such that the agents do not collide and all agents reach their goal (Stern 2019). This seems to align with the pipe routing problem where the joints would be considered the agents that move within a grid environment (the pipe handling system) and share similar destinations (processing stations). The most straightforward method to solve the MAPF would be rule-based heuristics that are based on operator-experience, but also routing specific heuristics like D* algorithm and priority based path planning were mentioned as suitable solvers.

Additionally, the literature review proposes to incorporate a dynamic programming approach to simply the pipe routing problem by dividing it in sequential subproblems that can each be solved with their own policy.

3.1.3. Application of the identified methods in this study

After the more detailed analysis of the pipe handling system described in Chapter 2, it is evident that there are no example systems in literature that can be directly compared to the pipe routing problem. Nevertheless, this subsection outlines how some of the identified methods are applied to model and solve the problem.

This study follows the proposal to develop a simulation model of the pipe handling system. The applied simulation method and software is further detailed in Section 3.2.

After looking further into modelling the pipe routing problem as an MAPF, it is concluded that some key features make this problem a more complex variation on classic MAPF. Classic MAPF algorithms assume time to be discretized in equal time steps with agents moving at the same time while in the pipe routing problem joints mostly move successively due to moving restrictions like usage of the TCs. Agents are considered to be elements that can move on their own while joints are passive elements that need to be moved around, making it more complex to translate the outcome of the MAPF algorithm into discrete decisions. Furthermore the joints in the pipe routing problem have multiple intermediate goals: visiting processing stations and potentially one of the holds before entering the DJF.

To address the complexity of intermediate goals, the problem can divided into sequential subproblems that can each be solved with their own policy. Instead of calculating the complete route of every joint including intermediate goals, the movement of the joint can be split into shorter routes with separate goals. Addressing all processes of the pipe yard system (see function A2 in Figure 2.1) as subproblems actually simplifies the problem to an extend where it can hardly be seen as a path finding problem as the navigable space decreases and the paths become more predictable. This will be further elaborated in Section 3.3.2.

What remains looks more like a sequential decision-making problem for individual movement steps, with consideration of movement restrictions like usage of the TCs and including decision-making on usage of the holds. This reintroduces MDP as a solution for modelling sequential-decision making problems. Otterlo et al. 2012 distinguishes three classes of algorithms for solving sequential decision problems: programming, search and planning, and learning algorithms.

Search and planning corresponds to traditional OR methods that solve MDPs mathematically to optimization. However, this approach is unsuitable for the pipe routing problem, as it would require modelling the pipe handling operations directly within the MDP, whereas the system's complexity necessitates modelling through an external simulation model.

Programming, similar to learning, can be applied to model-free MDPs with an external model. In

this approach, decisions rely entirely on rule-based heuristics, meaning it requires the programmer to anticipate and encode all possible situations during the design phase. As a result, the application of programmed solutions is limited for complex systems and dynamic environments (Otterlo et al. 2012), which likely explains the absence of comparable studies using heuristic solvers for sequential decision-making. However, for the pipe routing problem, the solution space can be effectively constrained, making it suitable for a well-founded programmed solution.

Therefore, this study models the pipe routing problem within a model-free MDP framework and applies rule-based heuristics for its solution, as further detailed in Section 3.3. Since the problem is formulated in a manner consistent with RL, the approach also enables future extension of this research through the implementation of an RL solver. In this way, it may serve as an example of how to address the research gap identified by Filom et al. 2022 regarding the industrial application of RL.

3.2. Simulation model of the pipe handling system

This section outlines the most suitable method to simulate the operations of the pipe yard system, followed by a literature review on simulation software to establish the program that will be used in this study.

3.2.1. Simulation modelling method

There are three main methods for simulation modelling: System Dynamics (SD), Discrete Event Simulation (DES) and Agent-based Modelling (ABM) (Maidstone 2012). SD is a method for simulating non linear behaviour of complex systems over time that are characterized by feedback and accumulation effects (TU Delft 2020b). It is more focussed on the flow of elements within a system, rather than the individual behaviour of elements (Maidstone 2012). A relevant example would be simulating the accumulation of joints in the pipe yard based on a fluctuating rate of pipe supply and pipe demand by the DJF with a feedback system that regulates the rate of flow of joints to the hold. To analyse the movement of individual joints within the system it is more straightforward to use one of the other simulation methods. DES is probably the most widely used method in OR (Maidstone 2012). This method models a system as a chronological sequence of events that change the state of elements within the system, which subsequently triggers new events (TU Delft 2020a). DES uses a top down modelling approach (Siebers et al. 2010), meaning the pipe yard system can be modelled as a series of sequential process with stochastic behaviour, where the joints are moving through. With this structure the paths of element movement are predetermined by decisions made at user-defined decision points instead of autonomous decisions resulting from interactions between elements (Wang et al. 2013). ABM on the other hand uses a bottom up modelling approach, meaning the joints and equipment can be modelled as individual agents with their own behaviour and decision-making and the interactions between all elements together form the system. Although this approach might be an interesting different view on the structure of the pipe yard system, it goes against the control principles defined in Section 2.4 as the system operations should be controlled by a centralized control agent (the pipe routing system) rather than multiple agents making their own decisions. DES is concluded to be the most suitable method for this study as it best resembles the pipe routing problem and allows for centralized control by the pipe routing system.

3.2.2. Discrete Event Simulation software

Appendix B describes the selection of the simulation software following a selection process described by Robinson 2004. This section will summarize the findings and conclusion of this process.

A shortlist including the following DES software is evaluated: SimPy, Salabim, JaamSim, AnyLogic, Tecnomatix Plant Simulation and MATLAB SimEvents. During the testing and evaluation of the shortlisted software options it became evident that the implementation of the desired control structure within a DES environment is unconventional. In specialized GUI packages DES models are typically thought of as networks of queues and servers with entities moving through (Maidstone 2012), resulting in limited solutions to control the entity flow. As such, a language-based modelling approach offers greater flexibility than a GUI with pre-built mechanisms.

Salabim scored the highest in the evaluation based on the following criteria: ease of model development, integrated animation, optimization solutions for decision-making, modelling support and

potential for future use of the model within Allseas. As a language-based simulation library in Python, this software provides all the essential features, including a basic integrated animation function. Since it is purely code-driven, modelling the system's content may require more effort. However, building the model from scratch allows for greater flexibility and will simplify the implementation of the control structure which is expected to be the biggest challenge.

3.3. Modelling and solving the pipe routing problem

This section explains how the pipe routing problem is modelled within a model-free MDP framework and solved using rule-based heuristics. While the solver algorithm is formulated during the modelling and experimentation phase, this section introduces an approach to constrain the solution space for identifying near-optimal solutions.

3.3.1. Model-free Markov Decision Process framework

This subsection first outlines a classical MDP framework, then the model-free RL framework, and finally the adaptations made to apply this framework to the pipe routing problem.

Classic MDP

MDP is a framework for describing stochastic sequential decision making problems (Miller 2023). An important property of MDP is that decision-making is only based on the information given by the current state of the model and not on any information from the past Hillier et al. 2015. A classic MDP can be formulated as a tuple (S, A, P, R) containing the following elements:

- state space S is the set of all possible states the system can be in with each state capturing the information required to make a decision,
- actions A allow the decision-maker to transition the current state into another state,
- transition probabilities P define the effect of action a by formulating the probability the system transitions to state s' given the current state s,
- rewards R specify the benefit or cost of executing action a given the current state s.

This classic form of MDP can be mathematically solved to optimality with the goal to maximize the cumulative reward. The transition probability and reward function can collectively be referred to as the model of the MDP as they describe what actually happens within the system.

Model-free Reinforcement Learning

In model-free RL, the problem is formulated as an MDP, but decisions are made directly from the policy of the RL model. This policy selects actions based on learning data rather than calculating the most promising option. The transition probability and reward elements are removed from the MDP and replaced by an external model, such as a DES model, which provides the system state and performs the action prescribed by the RL model. This action transitions the external model into a new state, which, together with a new reward function, is fed back to the RL model. This reward function evaluates the benefit or cost of the resulting state rather than the action itself, thereby assessing the actual effect of an action. The RL model then uses this feedback to update its policy.

Framework for the pipe routing problem

As the pipe handling system can better be modelled by a simulation model, the pipe routing problem will be defined as a model-free MDP derived from model-free RL. Instead of the RL model, the decision-maker in this framework is going to be the pipe routing system. The policy of the pipe routing system consists of a preprogrammed algorithm that will not be updated based on the reward of individual actions, therefore the the reward function can be removed from the framework.

Figure 3.1 shows the decision framework for the pipe routing problem. The DES model provides the system state s, comprising the status of joints, the occupancy of positions and the status of processing stations and transport equipment. The pipe routing system is the decision-maker that selects action a from its policy, given state s. Action a represents the next joint movement to be executed by the pipe handling system in the DES model. Once the action is performed, the DES model transitions to a new state s', which is then returned to the pipe routing system. The policy will from now on be referred to as the routing strategy of the pipe routing system.

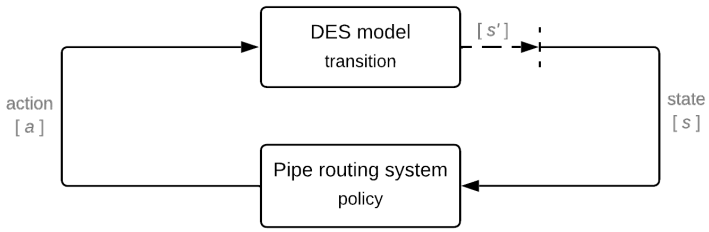


Figure 3.1: Decision framework for the pipe routing problem

3.3.2. Limiting the solution space

The routing strategy is formulated using rule-based heuristics, requiring the programmer to anticipate all system states in order to select the best action at each instance. Chapter 5 will evaluate different routing strategies to identify a best performing solution for a case study. However, the problem cannot be solved to optimality in this approach due to the size of the solution space. This section introduces an approach to constrain the action space of the decision framework, enabling the identification of near-optimal solutions with a limited number of experiments.

In Section 3.1.3 it was mentioned how the concept of dynamic programming inspired a structured approach to divide the pipe routing problem into subproblems relating to the processes within the pipe yard system as defined in Figure 2.1. This approach will also be used to limit the solution space for the pipe routing problem. Figure 3.2 shows how the pipe yard system can be divided into three subprocesses: marking, cleaning and DJF.

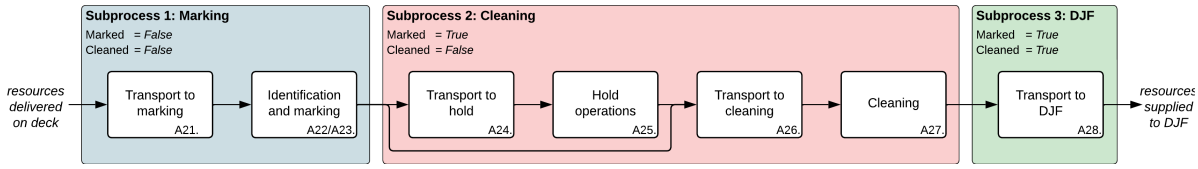
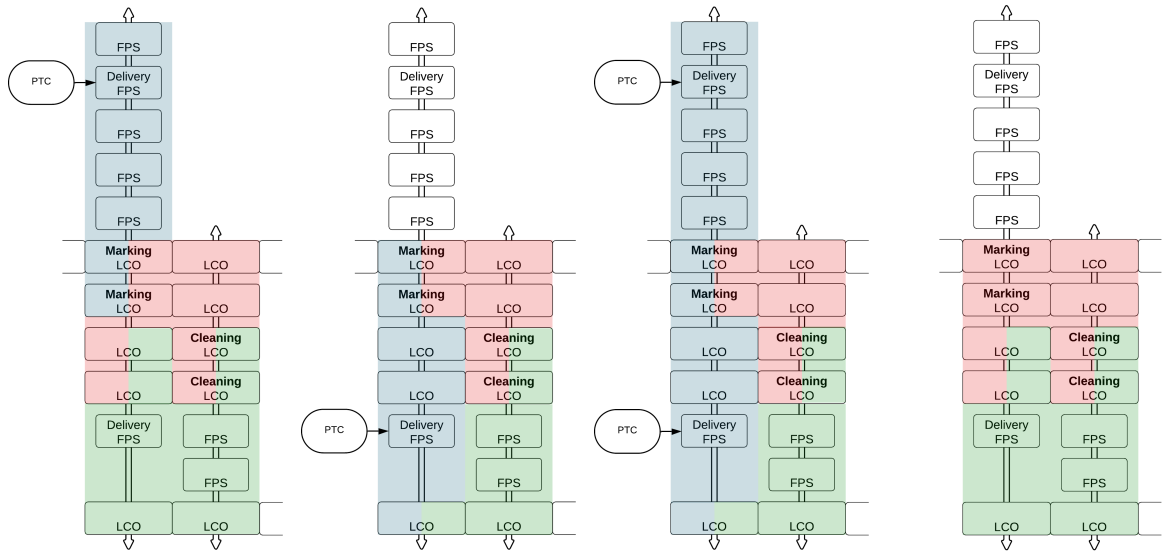


Figure 3.2: Pipe routing problem for the pipe yard system divided in subproblems

The navigable space in the pipe yard can be constrained based on these subprocesses. As each subprocess has clear starting points and destinations it is illogical to assign the navigable space beyond these positions. Figure 3.3 shows the navigable space corresponding to each subprocess, with the colours relating to the colours for the subprocess given in Figure 3.2. Figure 3.3 shows that the navigable space is also dependent on the pipe delivery scenario.



(a) PSV at port side **(b)** PSV at starboard **(c)** PSV at both **(d)** No PSV

Figure 3.3: Navigable space based on the joint's subprocess and the pipe delivery scenario

With the limited navigable space visualized it becomes clear the action space can significantly be reduced when only considering actions that contribute to the goal of the subprocesses. In RL this called action masking, which can be seen as a guided learning strategy that allows for integration of additional human knowledge (Eber et al. 2023). Opposed to reward shaping where an agent is penalized with a large negative reward for performing impossible or undesirable actions, action masking prevents the agent from making these actions by removing them from the action space (Wilson et al. 2024). This action space shaping induces an increased manual set up time and is susceptible to human bias (Kanervisto et al. 2020), however it will also result in increased data efficiency and it allows for user-defined constraints (Wilson et al. 2024). Previous studies have shown that action masking can be beneficial for simplifying the learning process in RL as well as improving safety.

The concept will be used in this study to limit the action space by masking joint movements that clearly do not do not contribute to the subprocesses of the pipe yard system. The limited action space requires less routing strategy variations during the experimentation phase to explore the entire solution space. For a future optimization phase based on RL it is also recommended to limit the action space to reduce learning time and ensure that joints do not collide.

3.4. Conclusion

This chapter summarized the relevant findings of a prior literature study on OR methods applied in comparable intralogistics systems and expanded the research on the proposed methods to make them applicable to the pipe routing problem.

The pipe routing system will be implemented in a DES model of the pipe handling system. Based on a brief literature review and exploratory testing of several programs, the Python library Salabim is concluded to be most suitable software for the development of this model.

The pipe routing problem will be modelled within a model-free MDP framework that is derived from model-free RL. The problem will be solved trough an algorithm build from rule-based heuristics, which is referred to as the routing strategy. In this approach it is not possible to identify an optimal routing strategy, however, the action space of the problem will be constrained to enable the identification of near-optimal solutions for the routing strategy during the experimentation in Chapter 5.

4

Modelling of the pipe handling system

This chapter describes the development of a simulation model for the pipe handling system controlled by a pipe routing system. The first section describes the methodology for the modelling procedure and the resulting sections that follow from this procedure.

4.1. Methodology

The modelling procedure of this project will be performed following the systematic approach described by Robinson 2004. This book divides the modelling procedure in four key phases: conceptual modelling, model coding, experimentation and implementation. This project only focusses on the first three phases, just like most studies consider the implementation phase as a separate project. The conceptual modelling phase is described in Section 4.2. Robinson 2004 defines the conceptual model as 'a non-software specific description of the simulation model that is to be developed, describing the objectives, in puts, outputs, content, assumptions and simplifications of the model. The model coding phase is described in Section 4.3. During model coding, the conceptual model is converted into a computer model. After development of the model, experiments are performed to obtain a better of understanding of the real system, or find solutions for the problem of the real system. This phase is described in Chapter 5. Although Robinson 2004 does not mention a specific model testing phase as it considers verification and validation to be continuous processes that should be performed throughout the modelling procedure, Section 4.4 describes how the verification and validation was performed for this study. Furthermore it should be noted that the modelling procedure is not linear with repetition and iteration of the modelling phases as well as sub-processes.

4.2. Conceptual model

This section describes the conceptual model of the pipe yard system controlled by a pipe routing system. The conceptual modelling phase consist of four sub-processes: understanding the problem situation, determining modelling objectives, designing inputs, outputs and content of the model, and collection and analysis of model data. The processes will be described in the following subsections.

4.2.1. Understanding the problem situation

The understanding of the problem situation has for the most part been described in Chapter 2. A prior simulation study within Allseas has given some additional insights on the pipe flow within the new pipe yard layout. From this prior study it was concluded that for pipe delivery with one PSV there was still a small percentage of the joints that was sent to the hold, meaning the supply of one PSV was sufficient to keep up with the DJF demand. It remains unclear whether anode joints require a priority policy over normal joints as they are not supplied as frequently.

4.2.2. Modelling objective

The general project objective is to evaluate different routing strategies in the pipe yard for the conceptual pipe routing system with the aim to get better understanding of the system and identify a most effective routing strategy. Therefore a simulation model is required that replicates the operations in the pipe handling system and the control structure of the pipe routing system, as described in Chapter 2. The pipe routing system should be implemented as a control agent making decisions based on the system state and an adaptable predefined algorithm, which is referred to as the routing strategy.

As the model should provide Allseas with a better understanding of the influences of routing strategies on the systems performance, a visual representation of the pipe flow during simulation is desired.

The simulation model is required to produce a realistic output for both the production rate of double joints by the DJF and the layrate of the firing line. In addition, the model will be employed to minimize the waiting time KPIs defined in Section 2.3, although no specific performance targets have been predetermined for these measures.

Since this study focusses on routing strategies in the pipe yard, some operations in the waiting area and beadstall will be simplified. To exclude the impact of this simplification on the system performance, the continuous supply of double joints to the firing line is reinterpreted as a continuous supply to the waiting area. Accordingly, the firing line waiting time as KPI is substituted by the waiting area waiting time, which reflects instances where a double joint required at the firing line is not supplied to the waiting area in time.

4.2.3. Model input

The model inputs can be seen as the experimental factors that can be used to achieve the modelling objectives. For this project the only experimental factor is the routing strategy of the pipe routing system, as all other parameters are assumed to be given.

An initial strategy for the pipe routing system will be formulated to achieve a continuous supply of double joints to the waiting area. During the experimentation phase in Chapter 5 this routing strategy will be experimented with to identify a most effective configuration.

While the simulation model will be prepared to adapt to different pipe delivery scenarios, during this phase the pipe routing system will only be designed for pipe delivery with one PSV at port side as this should be enough to verify and validate the simulation model. Chapter 5 will describe the different pipe delivery scenarios that will be evaluated in this study.

4.2.4. Model output

The model output can be seen as the responses of the model that evaluate whether the modelling objectives are achieved. For this project the most relevant model output are defined in Table 4.1. The DJF production rate and the main pipeline length indicate whether the model generates a realistic production output. The PTC waiting time and the waiting area waiting time are defined as the KPIs targeted for minimization by the routing strategy. Additionally, the DJF waiting time is considered as a supplementary output parameter, as it reflects inefficiencies within supply chain, although it is not directly related to the overall performance of the pipe handling system.

Furthermore a lot of output parameters are added to provide a better understanding of the system, like hold ratios and station utilization, as well as visual output through animation snapshots.

Table 4.1: Component process characteristics in the pipe yard system

Output parameter	Unit	Definition
DJF production	joints / day	The amount of non-rejected double joints produced by the DJF
Main pipeline length	m / day	The produced length of the main pipeline length that has successfully been welded in the firing line
PTC wait time (KPI)	min / day	The sum of the wait time where the PTC was ready to deliver a single joint, but the delivery position is still occupied by the earlier delivered joint
DJF wait time	min / day	The sum of the wait time where two bevelling stations are free, but there is no pair of single joints on the DJF supply positions
Waiting area wait time (KPI)	min / day	The sum of the wait time where the firing line is free to receive a double joint, but the correct double joint is not present in the waiting area

4.2.5. Model content

This subsection will describe the elements of the pipe handling system that will be implemented as components and the assumptions and simplifications that are applied on the system as it is described in Section 2.3 to formulate the model. Additionally, this subsection will describe the initial pipe routing system that will be implemented in the model.

Components

Figure 4.1 shows the content of the simulation model including all components. Some small adaptations can be noted compared to the content of the pipe handling system in Figure 2.22:

- The LCOs and FPSs are not actual components in the model, but they are featured in the model as positions for joints to move. The LCOs are split into fixed positions and joints cannot stay between these positions.
- The TCs have been simplified to a single TC carrying a joint.
- The TC couples are identified by a port side (ps) and a starboard side (sb) TC that cannot overtake eachother.
- The TC in the waiting area has been removed to simplify the routing strategy in this zone. All movements will be performed by the overhead crane.
- The deck cranes are removed to simplify the hold procedure. The deck crane will be implemented in the hold component.
- A deconstructor component is added at the end of the return line.

The pipe routing system regulates the movement of joints within the pipe yard, DJF and the return line zone. For the the waiting area zone an additional component is added that regulates the joint movements based on its own predefined algorithm that will not be adjusted during experimentation. Also an additional component for the DJF is added to regulate the joints being sent to either the port side or starboard welding line and store the waiting time for the DJF.

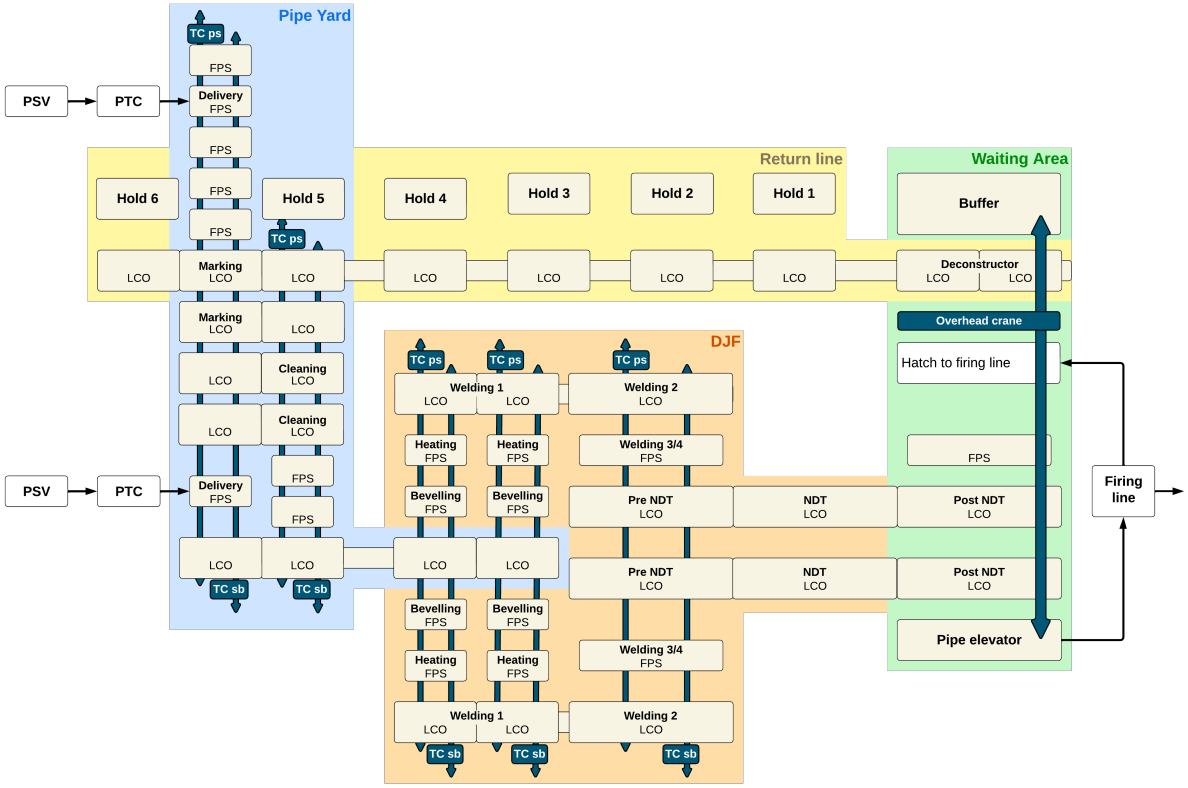


Figure 4.1: Model content

Assumptions

The following assumptions are made to further limit the complexity and scope of the simulation model:

- The PTC will have a fairly continuous pipe supply rate with randomized but reasonable delivery durations.
- The anode joints are delivered at fairly consistent intervals, with at most a one-position deviation from the desired spacing.
- While at least one PSV is present, there is only pipe flow to the hold and not from the hold.
- The elements in the simulation model will have no downtime.
- Deconstruction of rejected double joints is done on the return line and rejected double joints can be retrieved directly from the firing line through a hatch.

Simplifications

The following simplifications are made to further limit the complexity and scope of the simulation model:

- Acceleration and deceleration are simplified to movement with half the constant speed.
- The welding lines in the DJF are considered unidirectional all the way to the waiting area, joint crossing to the opposite line is not considered.
- The firing line is simplified to be directly supplied by the pipe elevator without a buffer location in the beadstall.

Pipe routing system

The pipe routing system applies a routing strategy consisting of rule-based heuristics to solve the pipe routing problem. The routing strategy will be further developed during the experimentation phase in Chapter 5, but for now it solves the pipe routing problem in the following sequential manner:

1. Which joint to move?
   This will be determined according to progress of the joint within the pipe yard system. From all idle joints with movements available, the joint that is the furthest into the process (in this simplified model simply the joint that has the lowest identification number) is moved first.
2. Where to move joint?
   This will be determined based on a predefined route that is implemented in the model through move maps. The joint will be moved to the furthest position available in the direction of the destination given by the move map.
3. When to move to hold?
   Part of the question where to move the joint is when to move a joint to the hold. This will be determined based on a push system where a joint will move to the hold when it is next to the marking station at port side, it is not able to move to one of the cleaning positions and it is blocking the movement of a marked joint at the marking station.
4. Which resource to use?
   The TC dispatching will be determined based on the destination that is assigned to the joint. This will be implemented in the model through a resource allocation list. When a TC is in use, all movements corresponding to that TC become unavailable.

4.2.6. Model formulation

The model will be formulated using a language based modelling approach where components are generated with a set of attributes and a process cycle. The components can interact with the environment and each other's process cycle through functions like hold, wait, passivate and activate. In general all process cycles loop infinitely, which is controlled by passivating the component. This section will explain the structure of the model environment and the general formulation of the process cycles of all components in textual form. The section is divided in several paragraphs describing similar or relating components. Additionally, Appendix C provides an explicit formulation of all the components for the conceptual model using a Programming Descriptive Language (PDL). Both this section and Appendix C focus on the operational concept of the components and exclude additional functions for animation and programming purposes.

Model environment

This paragraph describes the model environment which mostly stores input data, like component parameters, that can be referenced by all components. The environmental position map, occupancy map and move maps form the functional basis of how joint movements are regulated in the model.

The structure of Figure 4.1 is recreated in the model environment by giving all positional elements a positional name or grid coordinate. These names and grid coordinates are translated into map coordinates by means of an environmental position map. The map coordinates correspond to the actual distances between positions and allow for calculation of movement durations within the model. Part of the positions in the position map are listed in the routing positions, this list refers to all positions that are included in the scope of the pipe routing system. The environment also includes user defined move maps for the pipe routing system that define the destination and required movement resource for a joint based on the current position and status of the joint.

One dynamic feature of the environment is the occupancy map, which is used by the components to write or read the occupancy of a position. All grid coordinates are referenced and can be set to one of the following states: free, reserved or occupied.

Pipe delivery

This paragraph describes how joints are delivered in the model through the PSV and PTC components.

A PSV generates a given amount of regular joints and anode joints and stores these in a joint list with regular anode interval. The PSV is then passivated.

A PTC delivers the first joint from the list of its corresponding PSV to its delivery position. With the delivery a joint is given an identification number and it is added to joint list of the pipe routing system.The delivery time is split into a lifting time and a drop time. If after the lifting time the delivery position is not free, the PTC waits before it drops the joint. The waiting time is stored for evaluation at the end of the simulation. A PTC is passivated when the joint list of the corresponding PSV is empty.

Pipe Routing System

This paragraph describes how the pipe routing system assigns movements to the joints.

The pipe routing system is automatically activated every time a position in the position map is freed or occupied. For each joint that is added to the joint list of the pipe routing system, the destination and required movement resource of the joint are derived based on its current status and position through the move map. The occupancy map provides which position between the current position and destination of the joint are reachable (the move) and which positions are unreachable (the remainder).

If there is a move with reachable positions and with the required resource available, the move, remainder and resource are set to the joint and the joint is activated. All the positions in the move are set to reserved in the occupancy map. If the joint also has a remainder of unreachable positions it is temporarily stored in a separate list. The pipe routing systems updates the move of the joints in this list every time a position is freed.

If there are no more moves to assign the port side TC is sent to a position above the cleaning stations if it ended at one of the cleaning stations. Then the pipe routing system passivates itself.

The pipe routing system also has a separate function to send a joint to one of the holds. This function is activated by the marking station and sends a joint to the hold if it is blocking the joint at the port side marking station after it has been marked.

Joints and Double Joints

This paragraph describes how joints and double joints perform a move that has been assigned by the pipe routing system. Double joints are generated by the first welding station and hold a reference to the two single joints that have been used to form the double joint.

After generation, both joints and double joints are directly passivated, waiting to be activated by the pipe routing system with an assigned move. If the move is in x direction the move is performed directly by the joint itself. If the move is in y direction, the joint first has to request the resource to travel to its position and lift it. Then the joint moves together with the resource. And after the joint is dropped by the resource, the resource is released.

A move consists of three phases: acceleration, constant velocity and deceleration. During the constant velocity the joint may surpass several reserved positions that are freed again when the joint has passed. During the move additional positions from the remainder may be added to the move by the pipe routing system as long as the deceleration phase has not started. The joint position at the end

of the move is set to occupied in the occupancy map.

If the final position is a processing station, the joints passivates itself to be reactivated by the processing station. Then the joint indicates that it is ready for a new move, it activates the pipe routing system and it passivates to be reactivated once a new move is assigned.

Movement resources

This paragraph describes the TC and Crane which are both components and resource elements that facilitate joint movements in y direction.

A TC is mostly assigned to a joint by the pipe routing system and then requested and activated by the joint. When activated, the TC first moves to an assigned new position which can be the joint position, but this can also be just repositioning. If the TC was activated to perform a joint move, the TC then holds for the lift time and then activates the joint to perform the move together with the joint. After the move is performed, the TC holds for the drop time and activates the joint again. The TC is then passivated waiting for a new request.

The crane component refers to the overhead crane that is positioned in the waiting area, Crane 38, and operates a similar process. Instead of the lift and drop time, the cranes operates with a hoist time to lower the crane and lift the joints.

Processing Stations

This paragraph describes the general formulation of all processing station components that the joints have to pass by: Marking, Cleaning, Bevelling, Heating, Welding1, Welding2, Welding34 and NDT.

The processing stations are given a position and wait until this position is occupied by a joint. The stations then hold for a given processing time and optionally adjust the status of the joint. After this process the station activates the joint. Some processing stations perform some additional actions.

Welding station 1 is given a fore and aft position and waits until both are occupied by a joint. The single joints are removed from the simulation after the processing time and a new double joint is generated with references to the single joints. The double joint destination is then set to the next station, and when this position is free the double joint is activated to move there directly.

The NDT stations are given a reject probability and generates a random number between 0 and 1 after the processing time. If the random number is lower than the reject probability, the double joint status is set to be rejected.

Hold

This paragraph describes how the hold component stores joints. The hold components can be set to available or unavailable for storage before the start of the simulation.

A hold is given a position and a storage location. While the hold is available for storage, it waits until its position is occupied by a joint. If the joint status is set to hold, the holds for a short time to lift the joint. The joint position is then set to the storage location and the hold position is freed in the occupancy map. The holds then holds for a longer storage time. After this process the joint is added to the hold storage list and removed from the pipe routing system joint list.

DJF

This paragraph describes an additional component, the DJF, that regulates the transition of joints from the pipe yard to the bevelling stations in the DJF and registers the DJF waiting time.

The DJF waits until both the fore and aft supply position are occupied by a joint. The DJF component first checks the status of the bevelling stations at port side and then of the bevelling stations at starboard side. If both the fore and aft bevelling station at one side are free, the joints are sent to these stations. If both the port side and starboard side are not available the DJF passivates itself, waiting to be reactivated by one of the heating stations indicating that a bevelling stations has come free.

If the DJF is activated by a heating station but the supply positions have not been occupied by new joints yet, the DJF starts a timer that is stored in the wait memory once the positions are occupied.

Waiting Area

The pipe routing system regulates joint movements up until the waiting area zone. This paragraph describes how the components in the waiting area zone are connected through an additional waiting area component to regulate the joint movements within this zone.

The waiting area has three positions where double joints can be available, behind the NDT stations

at port side and starboard side and at an extra reserve spot. Furthermore the waiting area includes a buffer component to store double joints, an elevator component to transfer double joints to the firing line and a deconstructor component to deconstruct rejects. The waiting area component regulates the joint movements in this zone using a sequential if statement. If the firing line has rejects it sends these double joints to the deconstructor until the reject list from the firing line is empty. If the elevator position is free it sends one of the double joints in the waiting area to the elevator position. If there are no correct double joints in the waiting area it request a double joint from the buffer to move to the elevator. Then, if there are rejected double joints in the waiting area it sends these to the deconstructor. If their are double joints in the waiting area that block the movement of a double joint at the NDT they are sent to either the reserve position or the buffer. If none of these actions are possible the component is passivated. After the waiting area assigns one of the actions it waits until the crane ready again.

When the firing line is waiting for a double joint and the elevator is free, but the required double joint is not available in the waiting area, the waiting starts a timer that is stored in its wait memory once the double joint is available.

The buffer component waits until its position is occupied by a double joint or it is activated by the waiting area. If its position is occupied by a double joint it stores the double joint in a normal or anode list and removes it from the simulation. If there is no double joint the component is activated by the waiting area for retrieval. The buffer retrieves the required double joint to the simulation and set its move to the elevator.

The elevator waits until its position is occupied and then holds for a loaded elevator time and unloading time. Then the elevator waits until it is activated by the firing line and then holds for an empty elevator time. The elevator position is then set to free and the waiting area is activated.

The deconstructor waits until its deconstruction position is occupied by a double joint and then holds for the deconstruction time. The double joint is then removed from the simulation and the referenced single joints are brought back to the simulation at the fore and aft position. The joints attributes are adjusted, there destination is set to the hold and then they are added to the pipe routing system and activated. The deconstructor waits until all positions are freed and then activates the waiting area.

Firing line

This paragraph describes the formulation of the firing line that constructs the main pipeline.

After delivery of a double joint the elevator is sent back by the firing line. The pipe length counter is increased by one and if the counter can be divide by the anode interval the firing line indicates to the waiting area component that the next double joint should be an anode. The delivered double joint is added to a pipeline list and if the list is now longer than the reject length the first added double joint is removed from the list and from the simulation. Then the firing holds for the processing time.

If a random number between 0 and 1 is lower than the firing line reject probability all double joints in the pipeline list are added to a reject list, the pipeline length counter is adjusted and all double joints are set to rejects. The waiting area is activated to remove the double joints and the firing line passivates itself until the reject list is emptied.

4.2.7. Model data

This section provides the model data that is stored in the model environment at the start of the simulation. Figure 4.5 shows an overview of the positional layout of all components mentioned in the model formulation. The figure shows the grid coordinates of components, as well as the actual distances between components that are used to formulate the position map.

Table 4.2 shows an overview of all the input data that has been used to form the parameters of the simulation. All durations are based on 36 inch diameter pipeline from the South East Extension (SEE) project. This project is taken as example as it is used before in a prior simulation study and there are video recordings available of the production during this project. Equipment attributes are based on the actual equipment specifications while process attributes are based on estimations provided by the Pipeline Production Department (PPD). The process durations with a specified distribution are defined with an estimated probability density function (PDF) that is based on estimations provided by PPD combined with a self-conducted data study that is detailed in Appendix D. The following paragraphs will explain the how the process duration for the processing stations are defined.

PTC

The PTC lift duration is defined by a triangular distribution to simulate stochastic behaviour of the lift process. A triangular distribution is defined only by the minimum, the most common (mode) and the maximum value. This is often used in simulation when there is relatively little data available to conduct a full statistical analysis (Kissell et al. 2017). The defined triangular PDF, shown in Figure 4.2 is based on an average delivery rate of 24 joints per hour with a maximum speed of 40 joints per hour and a minimum speed of 15 joints per hour.

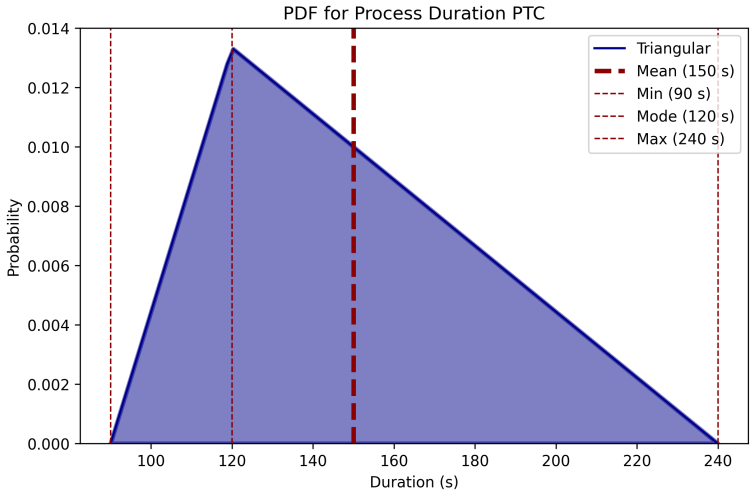


Figure 4.2: Probability density function for station duration PTC

Firing line

The firing line process duration will be randomly sampled from a data set with more than 20.000 data logs collected by a datalogger during the SEE project. The minimum duration and maximum duration of the firing line are respectively set to 300 and 1920 seconds. All data logs outside of this range are removed from the data set as faster durations are unrealistically fast while slower durations indicate some form of downtime and the simulation is meant to simulate continuous production. This resulted in the removal of a negligible number of low durations and a total of 2% of the highest durations from the dataset. Figure 4.3 shows the final PDF for the firing line process duration. The process duration consists of a critical station duration and a pull duration. The pull is assumed to be a constant value of 120 seconds, making the critical station duration equal to the sampled process duration minus 120 seconds. From now on when the firing line process duration is mentioned, this references to the critical station duration which has a mean of 333 seconds.

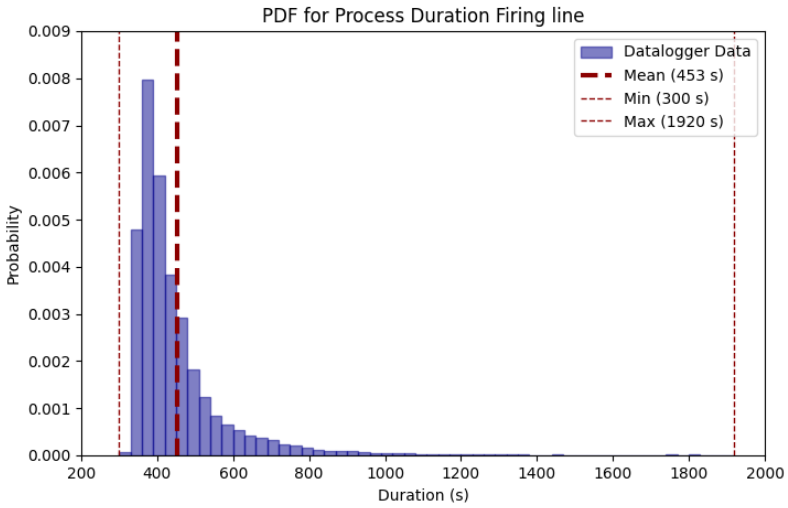


Figure 4.3: Probability density function for process duration Firing line

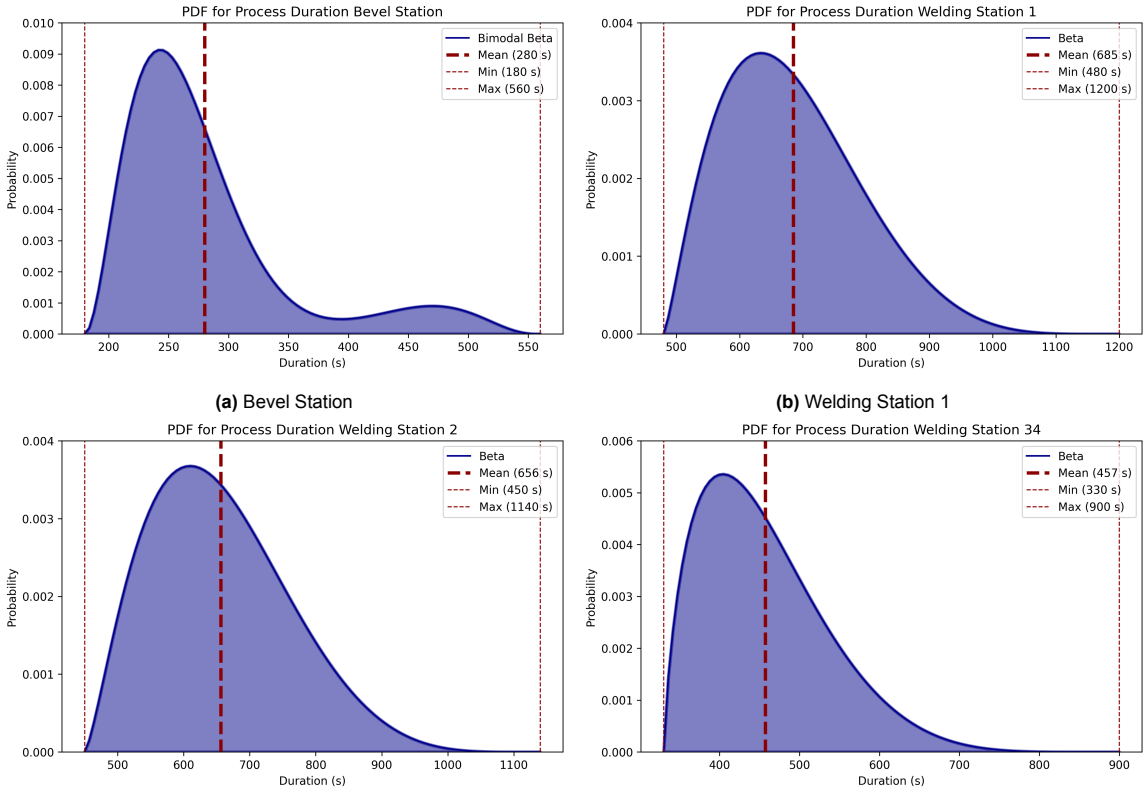
Marking and cleaning

The marking and cleaning process duration are both simplified as a constant value as these processes are fully automated.

DJF processing stations

The process durations for the stations within the DJF should be derived from empirical data to accurately capture the stochastic nature of the operations in the simulation. However, there is a lack of accurate data available on the DJF within Allseas as PPD generally assumes the production of the DJF to be sufficient to supply the firing line and has therefore not been interested in accurately logging data for the DJF during projects. An attempt has been done to extract duration data from a datalogger, but the output shows a flattened distribution with an overestimated mean for the duration of the welding stations.

Therefore Appendix D describes a self-conducted data study that concludes with an estimated PDF for the duration of the bevel station, welding station 1, welding station 2 and welding station 3/4. The PDFs shown in Figure 4.4 are formulated as Beta functions with the estimated mean duration provided by PPD and estimated minimum and maximum values from the data study. The process duration for the Heating station and NDT station is assumed to be constant, based on the estimated duration provided by PPD.



(a) Bevel Station

(b) Welding Station 1

(c) Welding Station 2

(d) Welding Station 34

Figure 4.4: Probability density functions for station durations in the DJF

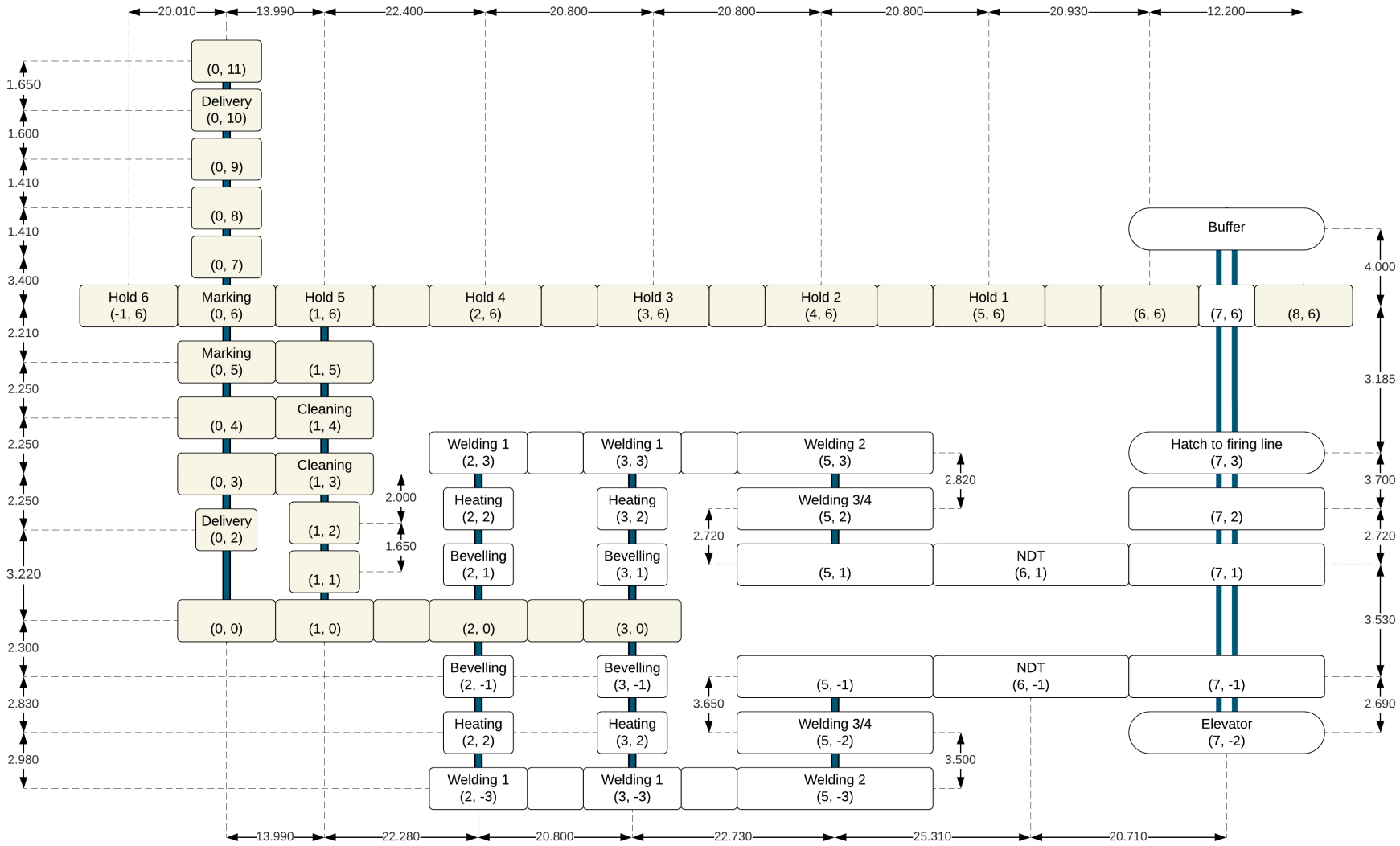


Figure 4.5: Model annotations

Table 4.2: Component process characteristics in the pipe yard system

Component	Attribute	Distribution Type	Value	Unit
PTC	Process duration	Triangular (min, mean, max)	90, 150, 240	s
LCO	Travel velocity	Constant	500	mm/s
	Travel acceleration	Constant	150	mm/s^2
TC	Travel velocity	Constant	250	mm/s
	Travel acceleration	Constant	100	mm/s^2
	Lift duration	Constant	5	s
Crane38	Travel velocity	Constant	20	m/min
	Travel acceleration	Constant	100	mm/s^2
	Hoist velocity	Constant	5	m/min
	Hoist stroke for travel	Constant	2.9	m
	Hoist stroke to firing line	Constant	10	m
Marking	Process duration	Constant	45	s
Cleaning	Process duration	Constant	90	s
Hold	Process duration	Constant	144	s
Bevelling	Process duration	Bimodal Beta (min, mean, max)	180, 280, 560	s
Heating	Process duration	Constant	120	s
Welding 1	Process duration	Beta (min, mean, max)	480, 685, 1200	s
Welding 2	Process duration	Beta (min, mean, max)	450, 656, 1140	s
Welding 3/4	Process duration	Beta (min, mean, max)	330, 457, 900	s
NDT	Process duration	Constant	240	s
	Reject probability	Constant	0.007	
Elevator	Stroke	Constant	10	m
Elevator	Empty velocity	Constant	25	m/min
Elevator	Loaded velocity	Constant	20	m/min
Deconstructor	Processing duration	Constant	120	s
Firing line	Process duration	Empirical (min, mean, max)	180, 333, 1800	s
Firing line	Pull duration	Constant	120	s
	Reject probability	Constant	0.007	

4.3. Computer model

The conceptual model detailed in Appendix C is implemented in Python (version 3.12.9) with the DES package Salabim (version 25.0.10). This section describes the structure of the computer model and its inputs and outputs.

The computer model consists of a main python script, two additional python scripts for the parameters and the map and an Excel file for the input data. The Excel file may be adjusted by the user and includes the model parameters, map specifications and move maps describing the predefined routes for the pipe routing system. The additional python scripts import the input data from the Excel files and copy the information in data files that are accessible by the main python script. The main python script describes the setup, process and generation of the simulation environment and all components. The main script also specifies how long the simulation will run, during which time range the data will be collected and what data will be output at the end of the simulation.

The simulation can optionally run with animation, a feature that was not described in the conceptual model, but is added to the computer model. Salabim's animation engine allows for the animation of a number of shapes, texts and images that can be dynamically updated. This is implemented in the computer model to animate the layout of the position map, the movement of joints and movement resources and the occupancy of positions and stations, as shown in Figure 4.6. The animation feature is used for debugging and verification of the computer model, but can also be used in the model output to get a better understanding of the simulation results.

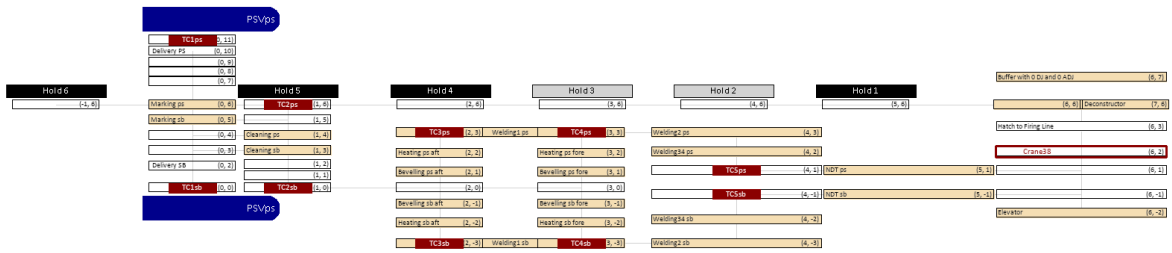


Figure 4.6: Snapshot from animation feature of the simulation model

4.3.1. Model input

As mentioned the model input can be adjusted by the user within an Excel file that includes the simulation parameters, the map specifications and the move maps. The simulation parameters are initially based on the realistic component process characteristics summarized in Table 4.2. Figure 4.7 shows a visualization of the move map for supply from one PSV at port side that is initially implemented in the input data with joints following the red arrows. Notice that in this visualization, the port side and starboard TC that operate in the same column are split in two vertical lines with the port side TC performing all joints for which the red arrow is placed on the left line and the starboard TC performing all joints movements for which the red arrow is placed on the right line.

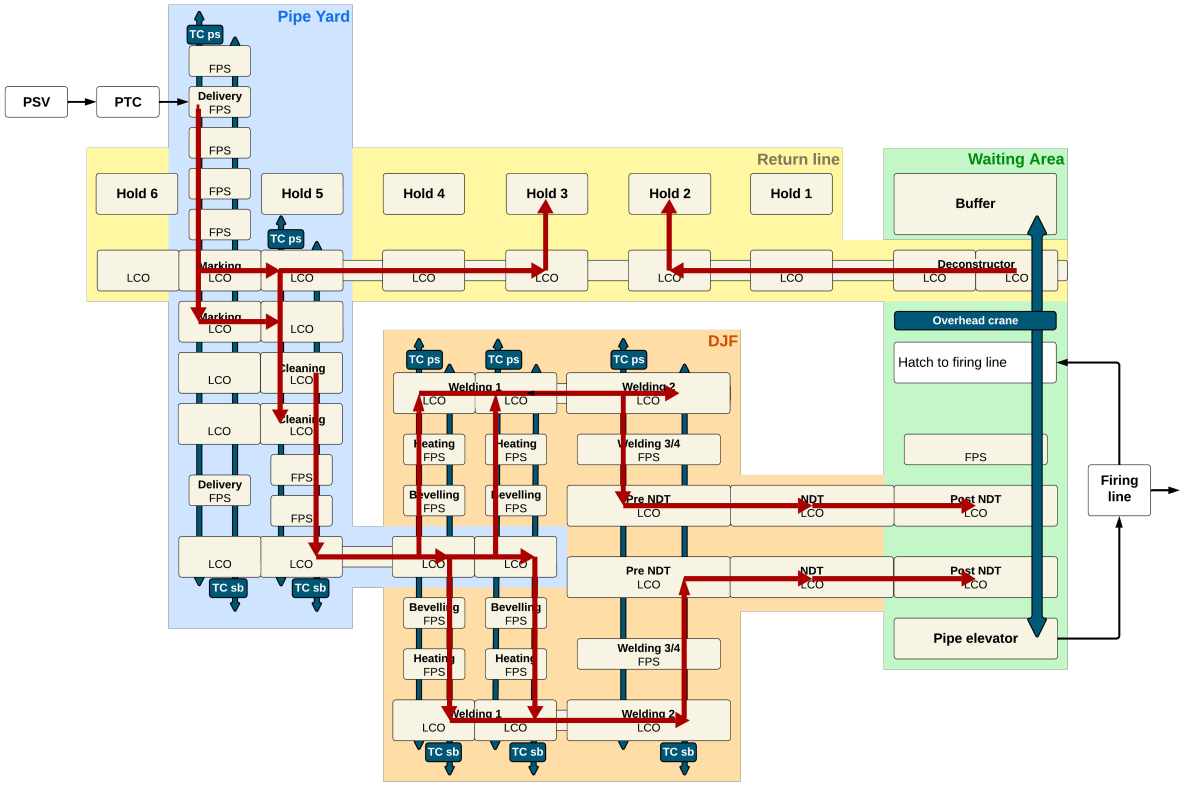


Figure 4.7: Input route for supply from PSV at port side

4.3.2. Model output

The simulation output at the end of the simulation consists of a data report and wait report. The extensive data report provides insights on the pipe flow, station utilization and production quantities. The data report and wait report are saved to a new sheet in another Excel file with their seed value for reproducibility.

The wait report also summarizes the most relevant output parameters, which are the DJF production, the main pipeline length and the waiting times for the PTC, DJF and waiting area. The PTC wait time and the waiting area wait time form the actual KPIs of the system.

The model output utilizes a warm-up period for data collection to address initialization bias. Each simulation starts with an empty pipe yard, DJF and waiting area. When the sum is taken of all maximum process durations and movement durations, it shows that it takes the first double joint maximum 1.5 hours to arrive in the waiting area. As this double joint has spent the maximum duration at all processing stations in this hypothetical simulation run, it is inevitable that this double joint has been blocking the movement of the double joints behind, resulting in a completely filled pipe yard in DJF. It is therefore reasonable for now to assume a warm-up period of 2 hours is long enough to reach a steady state for the simulation output.

4.4. Verification and validation

Verification and validation are an important element of any simulation study to ensure that the simulation results are useful and reliable. The model testing methods from Robinson 2004 will be used in this section to perform a structured and complete verification and validation of the simulation model. Robinson 2004 proposes that model testing should not be seen as a specific phase, but a continuous process that is performed throughout the simulation study. Figure 4.8 shows how for each phase in the modelling procedure at least one verification or validation step is performed. The remainder of this section will the highlight the conceptual model validation, data validation and the computer model verification, white-box validation and black-box validation that are performed in this simulation study.

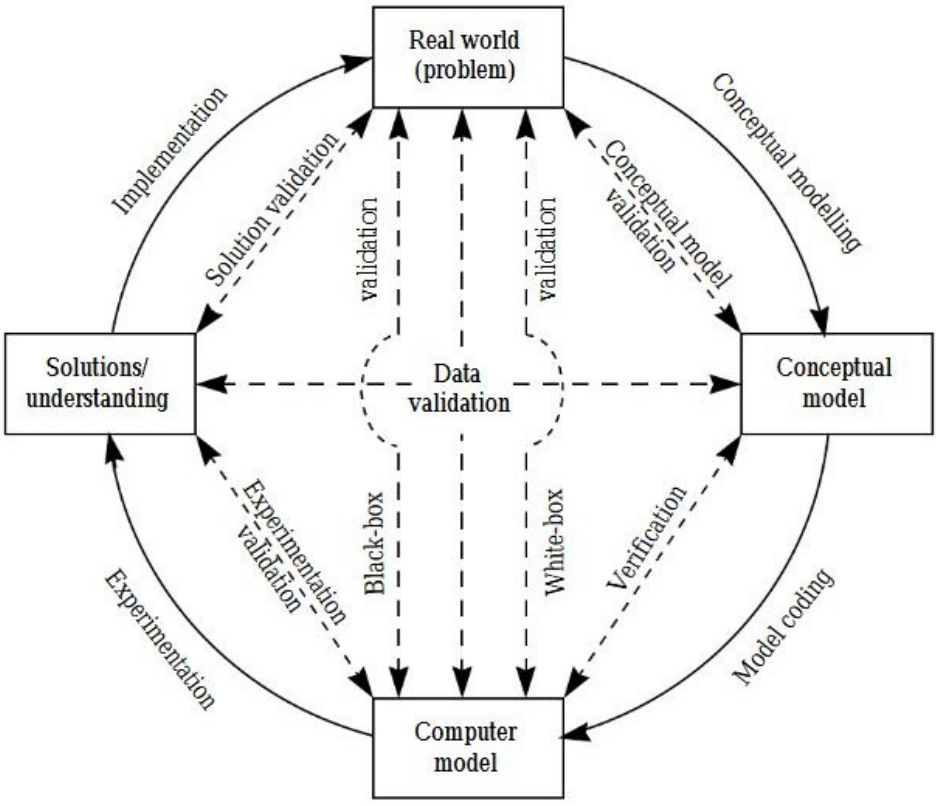


Figure 4.8: Verification and validation in simulation study Landry et al. 1983

4.4.1. Verification

Verification ensures that the computer model is true to the conceptual model and is performed throughout the model coding phase. This subsection describes the verification methods that have been used to ensure that the model performs as expected. The output data from three simulation runs with the same realistic input parameters is provided in Table 4.3. This data will be used for several verification methods.

Table 4.3: Data report for verification with realistic input parameters

		Simulation Run 1			Simulation Run 2			Simulation Run 3			
Data	**Unit**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**	**Average**
PTC delivery	joints	568	0	568	577	0	577	574	0	574	573
Marked	joints	314	254	568	334	243	577	317	257	574	573
Cleaned	joints	210	210	420	208	209	417	215	215	430	422
Sent to hold	joints			148			159			146	151
Sent to hold ratio	-			0.26			0.28			0.25	0.26
DJF production	joints	103	104	207	102	103	205	108	105	213	208
DJF rejects	joints	2	0	2	3	0	3	0	1	1	2
Firing line rejects	joints			6			0			6	4
Main pipeline length (m)	m			4294			4562			4294	4383
PTC wait count	-	0	0	0	0	0	0	0	0	0	0
PTC wait time (KPI)	s	0	0	0	0	0	0	0	0	0	0
DJF wait count	-	10	30	40	18	12	30	8	23	31	34
DJF wait time	s	675	2124	2799	1013	1267	2280	369	1886	2255	2445
Waiting area wait count	-			2			1			0	1
waiting area wait time (KPI)	s			1173			1798			0	400
Firing line wait count	-			9			7			9	8
Firing line wait time	s			1173			1798			517	1163

Run-time checks

The simulation model is build iteratively, adding components and complexity gradually. Run-time checking through event tracing was performed during coding to verify the process cycle of each new component that was added to the model. Especially the combination of the animation feature showing the events within the model and the run-time checks deriving the origin of the events ensured each component was programmed correctly.

Visualization

As mentioned the animation feature of the computer model was used to verify the process of components during coding. Also the finished computer model is verified through visualization of the joint movements and station occupancy. Figure 4.6 shows a snapshot of Simulation 1 in Table 4.3 at the start of data collection.

The animation is used to visualize the layout of the position map, the movement of joints and movement resources and the occupancy of positions and stations. The outline of positions colours green when they become free, black if they are free, orange if they are reserved and red if they are occupied. Holds are coloured black if they are closed and grey if they are available for storage or retrieval. The joints receive an identification number at pipe marking that is coloured orange for anode joints. The identification number of double joints is a composition of the single joint identification numbers and turns red if a double joint is rejected at NDT.

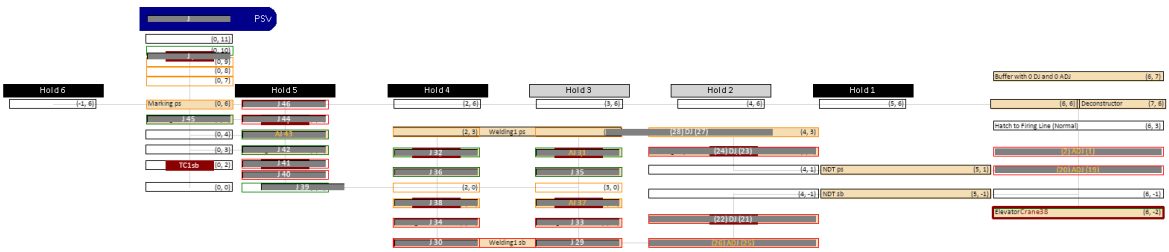


Figure 4.9: Snapshot from Simulation 1 in Table 4.3

Data verification

Table 4.4 shows the average station duration from the simulation runs in Table 4.3 to verify that the used distribution functions provide the expected average process duration. The last column ”Diff” shows that the average process duration over three simulation runs of 24 hours (total of 600 samples) differs less than 2% from the expected average value. The welding 1 wait time shows the average time that welding station 1 is idle and waiting for welding station 2 to become free.

Table 4.4: Average station durations for simulation runs in Table 4.3

		Simulation Run 1			Simulation Run 2			Simulation Run 3			Average	Diff
Data	**Unit**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**		
PTC process duration	s	152	0	152	150	0	150	151	0	151	151	0.7%
Bevel process duration	s	279	276	278	281	277	279	281	283	282	280	-0.1%
Welding 1 process duration	s	671	672	671	669	686	678	666	679	672	674	-1.7%
Welding 1 wait time	s	70	74	71	67	67	70	57	56	57	66	
Welding 2 process duration	s	657	665	661	666	668	667	644	652	648	659	0.4%
Welding 3/4 process duration	s	465	470	467	460	453	456	461	465	463	462	1.1%
Firing line process duration	s			341			328			344	338	1.4%

Balance checks

The continuity of the pipe flow can be verified with some simple balance checks:

- Joints marked = Joints delivered
- Joints cleaned + Joints sent to hold = Joints marked
- DJF production = 0.5 * Joints cleaned
- Main pipeline length $\leq$ DJF production

Filling in the output data from Table 4.3 shows that all balance checks can be confirmed considering an allowable difference of a couple of joints. This variation arises due to joints being between the measuring points and the fact that data collection begins during the simulation run rather than at a stationary starting point.

Experiments

Table 4.3 already shows three simulation runs with the same input, verifying the seed independency as al results are in the same order of size. However, it can be noted that the amount of rejects per seed is highly influential on the final main pipeline length with a simulation duration of 24 hours as the reject probability is so low. On average a 0.7% reject probability for 200 double joints would result in 1.4 rejects (a reject in the firing line results in 6 rejected double joints).

Table 4.5 shows three more experiments to verify that the model behaves as expected.

The first experiment will be run with setting all process distributions to a constant average value and the reject probability set to zero. This experiment will be run twice to verify that the simulation is reproducible with these settings. With constant process durations it is expected that there will be less waiting time between stations and therefore a higher DJF production, which matches with the data in Table 4.5.

The second experiment will be run with the PTC process duration being set to the constant maximum delivery rate. It is expected that the PTC wait time will now be higher than zero and there are more joints sent to hold. Furthermore it should not impact the DJF production as the supply to the DJF was already sufficient.

The third experiment will be run with the PTC process duration being set to the constant minimal delivery rate. It is expected that the process stations in the pipe yard can keep up with the delivery rate and no joints are sent to the holds. The delivery will not be able to keep up with the DJF and therefore increase the DJF wait time and decrease the DJF production. This can be seen clearly in the results in Table 4.5 as the DJF now produces at port side as this side is preferred in the model when all bevelling stations in the DJF are free.

Table 4.5: Data report for verification experiments

		Constant process durations			Max PTC delivery rate			Min PTC delivery rate		
Data	**Unit**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**	**PS**	**SB**	**Total**
PTC delivery	joints	576	0	576	698	0	698	360	0	360
Marked	joints	295	281	576	490	208	698	0	360	360
Cleaned	joints	226	226	452	209	208	417	0	360	360
Sent to hold	joints			123			281			0
Sent to hold ratio	-			0.21			0.4			0
DJF production	joints	114	113	227	105	104	209	104	76	180
DJF rejects	joints	0	0	0	0	0	0	0	0	0
Firing line rejects	joints			0			0			0
Main pipeline length	joints			191			189			179
PTC wait count	-	0	0	0	412	0	412	0	0	0
PTC wait time (KPI)	s	0	0	0	23409	0	23409	0	0	0
DJF wait count	-	21	29	50	28	17	45	102	20	122
DJF wait time	s	1317	2485	3802	1898	1830	3728	22613	12744	35357
Waiting area wait count	-	0	0	0	0	0	0	7	0	7
Waiting area wait time (KPI)	s	0	0	0	0	0	0	1292	0	1292
Firing line wait count	-			0			10			28
Firing line wait time	s			0			408			4446

Analytical results

The last verification step will be comparing the analytical DJF production to the simulated DJF production from the constant process duration data in Table 4.5. The constant process duration data will be used as welding station 2 is always faster than welding station 1 in this simulation run. The analytical DJF production can therefore be calculated with the bottleneck process duration by adding up the TC transfer duration from heating to welding 1 (Equation 4.1), the welding 1 process duration and the LCO transfer duration to free the occupancy of welding station 1 (Equation 4.2). Equation 4.3 shows that the analytical DJF production is 230 double joints, which matches with the 227 double joints measured in the simulation.

$$t_{TC} = t_{acc} + t_{constant} + t_{dec} = 2.50 + \frac{2.98 - 2 * 0.31}{0.25} + 2.50 = 14.42s \tag{4.1}$$

$$t_{LCO} = t_{acc} + t_{constant} = 3.33 + \frac{24.40 - 0.83}{0.50} = 50.74 \tag{4.2}$$

$$\text{DJF production} = 2 * \frac{t_{sim}}{t_{bottleneck}} = 2 * \frac{24 * 3600}{14.42 + 685 + 50.74} = 230 \text{ double joints} \tag{4.3}$$

4.4.2. Conceptual model validation

The conceptual model is validated during formulation by discussing the working principle of the conceptual model content and the assumptions and simplifications with the company supervisors at Allseas.

4.4.3. Data validation

Most of the simulation data is based on equipment specifications and / or has been collected together with co-workers at Allseas to ensure it provides an accurate estimation of the real system parameters. Additionally, Appendix C describes a self-performed data analysis to formulate and validate the process duration data that was not accurately enough available at first. This analysis concludes with four proposed PDFs describing the process duration for bevelling, welding 1, welding 2 and welding 34.

Table 4.6 shows the results of the Kolmogorov-Smirnov test (K-S test) to indicate if the self-measured empirical distribution and the proposed PDF are considerably different. The K-S test statistic is the largest deviation between the empirical data and the PDF. The K-S test p-value is the probability that the statistic may occur assuming the empirical data comes from the PDF. A p-value lower than 0.05 provides strong evidence that the distribution of the empirical data and the PDF are significantly different (Wasserman 2004).

Table 4.6 therefore shows that proposed PDF is not an accurate estimation of the self-measured empirical data for the welding stations. This was already concluded in Appendix D as the difference between the mean provided by PPD and the video analysis data is too large to formulate a PDF fitting both data. The mean value provided by PPD is assumed to be the most reliable and therefore it is concluded that the proposed PDFs based on this mean value are the most accurate data that can be generated with the available resources. But it should be noted that this data is not successfully validated.

Table 4.6: Results from K-S test between empirical data and proposed PDF

Process	K-S statistic	K-S p-value
Bevelling	0.11	0.68
Welding 1	0.40	0.00
Welding 2	0.26	0.01
Welding 3/4	0.33	0.00

4.4.4. White-box validation

White-box validation ensures that the model content is similar to the real system, which makes it indirectly also a form of conceptual model validation (Robinson 2004). Like verification, white-box validation is performed throughout the coding phase. The formulation of the code, and the data reports and animation output have been discussed with supervisors at different stages of the coding phase to ensure every component operates similar to the real system or an agreed simplification of the real system.

4.4.5. Black-box validation

Black-box validation considers the overall behaviour and performance of the model, which is generally validated by comparison to the real system or another model of the system (Robinson 2004). However, in this occasion there is no data from the real system available as the real system does not exist yet and there is also not another model that the simulation model can be directly compared to. However, the general performance of the simulation model can be compared to empirical production data of the old system and a prior simulation study.

Figure 4.10 shows the actual production of the firing line during the SEE project with comparison to the average DJF production and firing line production of the simulation model. This figure shows that the firing line production from the simulation can indeed be seen as an estimate of the average layrate during the SEE project. The DJF production line shows that the average production rate of the DJF can generally keep up with the layrate during the SEE project.

The actual layrate data cannot be directly compared to the simulation output as the logged process durations have been filtered with minimum and maximum thresholds to remove downtime from the actual production data. By interpreting all excluded process durations as downtime, it can be estimated that the total downtime, or more precisely, the total period without continuous production at expected process durations, amounted to 55.3 days over the 198-day project. During the 142.7 days of continuous production, *Solitaire* made 26.843 pulls which relates to a total pipeline length of 655 km and an average layrate of 4.6 km/day.

The average layrate of around 4.4 km/day obtained from the simulation is 4.5% lower than the layrate observed during the SEE project, which can be partially explained by the firing line waiting time in the simulation which is currently 1.3% of the total production time. The remaining 3% is considered small enough to conclude that that the model produces a realistic output for the layrate of the firing line.

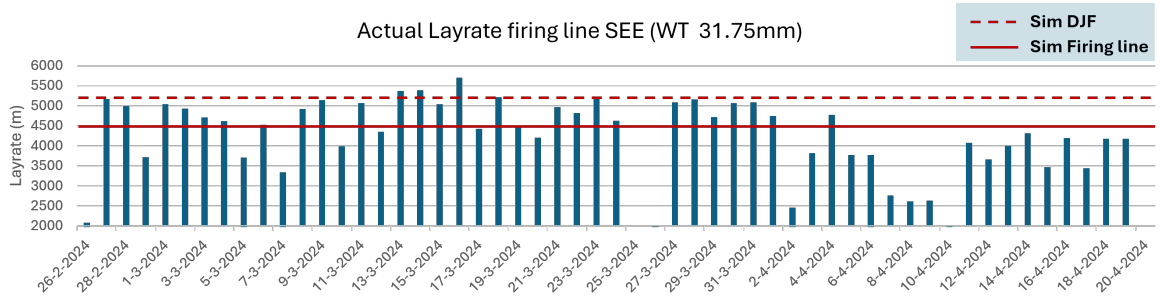


Figure 4.10: Actual firing line production rate during SEE project ©Allseas

The black-box validation can also be supported by a prior simulation study within Allseas where a simplified model was used to estimate the production rate for the new DJF configuration. In this study an average DJF production of 5.7 km per 24 hours was estimated based on constant process duration parameters provided by PPD. Table 4.5 shows a DJF production of 5.5 km with similar input parameters, which is only a difference of 3.5%.

4.5. Conclusion

This chapter has described the development of a simulation model that simulates the operations of the pipe handling system controlled by a pipe routing system. The objective of the model is to evaluate the performance of the pipe routing system and provide a better understanding of the effects of different routing strategies. The model content and assumptions and simplifications have been established to formulate a conceptual model which describes the process cycle of each component in the model. An initial routing strategy for the pipe routing system is formulated and implemented, which may be adjusted during the experimentation phase. The modelling data is based on the SEE project with a pipeline with a diameter of 36 inch, which is an average size diameter. The conceptual model is implemented in a computer model using Salabim, a DES package in Python. The conceptual model, model data and computer model are verified and validated throughout the modelling process. The duration data for the processing stations in the DJF could not be successfully validated, however it is well substantiated and concluded to be the most accurate estimation of the data with the available resources.

5

Experimentation with routing strategies

This chapter presents the simulation-based experimentation conducted to evaluate and further refine the conceptual pipe routing system across various routing strategies. The objective is not to achieve a predefined performance target, instead the aim is to explore potential solutions and gain a deeper understanding of how different routing strategies influence system performance under varying scenarios.

The chapter begins with an introduction to the experimentation methodology, followed by a detailed description of the experimental setup. Subsequently, the baseline performance of the initial pipe routing system is presented. This is followed by an exploratory phase in which various additions to the routing strategy are tested, and a conclusive phase in which combinations of strategies are evaluated. The chapter concludes by identifying the most effective routing strategy for the examined scenarios.

5.1. Methodology

As in Chapter 4, again Robinson 2004 will be followed to establish a well-founded research methodology. Robinson 2004 indicates two issues in simulation experimentation: obtaining accurate results, and efficiently and effectively searching the solution space.

Obtaining accurate results focuses on addressing initialization bias and collecting sufficient output data. To address this the simulation model should be classified as either a terminating or a non-terminating simulation. Initialization bias can be addressed by setting initial conditions or determining a warm-up period. The collection of output data can be addressed by determining an appropriate run-length or number of replications.

Searching the solution space focuses on the selection of scenarios and experimental factors that are evaluated to achieve the desired outcome of the simulation study, which may be an optimal solution, one that meets predefined requirements, or simply a deeper understanding of the real-world system. Robinson 2004 proposes all simulation experiments can be categorized by two pairs of terms: interactive and batch experimentation, and comparing alternatives and search experimentation.

The first pair describes how the simulation runs are performed. Interactive experimentation involves observing the model and apply changes to evaluate the effect. This can be helpful to develop an understanding of the system, but the results should always be tested by more thorough batch experimentation. Batch experimentation is performed by running the model for a predefined run-length and a set number of replications to obtain results that have statistical significance.

The second pair describes how the experimental factors are determined. In comparing alternatives the set of scenarios that will be evaluated is limited and often known in advance. In search experimentation there is not a given set of scenarios, but a set of experimental factors that may be varied until a target or optimum level is reached. A simulation study can involve a combination of both strategies to compare scenarios and explore the solution space.

5.2. Modification of the model

Before formulating and executing the experimental plan, this section addresses two issues in the simulation model output that must be resolved, as the model in its initial form was not suitable for the intended experimental application.

Buffer capacity

Whilst addressing the attainment of accurate results for experimentation it was observed that, for longer simulation durations, the unlimited buffer in the waiting area produced irrelevant waiting times for both the waiting area and the firing line, causing misalignment with the KPIs. Three possible adaptations were identified, each addressing different research objectives:

- Unlimited buffer capacity: Ignore firing line waiting time and sequencing requirements, focusing solely on maximum throughput in the pipe yard and DJF.
- Limited buffer capacity: Simulate a realistic scenario where sequencing requirements may cause deadlocks if the waiting area becomes congested with only one type of double joint. The simulation study will be less informative on maximum throughput in the DJF and waiting area becomes less informative.
- Reduced firing line process duration: Balance DJF and firing line production rates, maintaining focus on throughput in the pipe yard while considering sequencing. However, firing line waiting time becomes strongly influenced by stochastic DJF behaviour.

This simulation study applies a combination of the second and third adaptation to consider both sequencing strategies and throughput in the pipe yard in a realistic scenario. The buffer capacity is limited to 6 double joints based on the pipe line specifications of the SEE project. The firing line process time is reduced by using only the 80th percentile of the empirical data shown in Figure 4.3. This results in the PDF shown in Figure 5.1, which has a decreased mean process duration of 400 s, compared to the former used mean process duration of 453 seconds. The decreased process duration results in a theoretical production of $\frac{3600*24}{400} = 216$ double joints, which is slightly lower than the theoretical production of 230 double joints for the DJF calculated in equation 4.3. The decreased firing line process duration simulates an optimistic, but also realistic, production rate that results in the desired scenario where the DJF is able to continuously provide the firing line when sequenced correctly without being held up by a congested waiting area as the limited buffer is utilized as intended.

The buffer capacity limit is implemented by adapting the routing strategy of the waiting area to not send more double joints to the buffer when it has reached its maximum capacity. The decreased firing line process duration is implemented by filtering the empirical data when loaded into the model. Both implementations have been verified and validated through the animation feature and the output results.

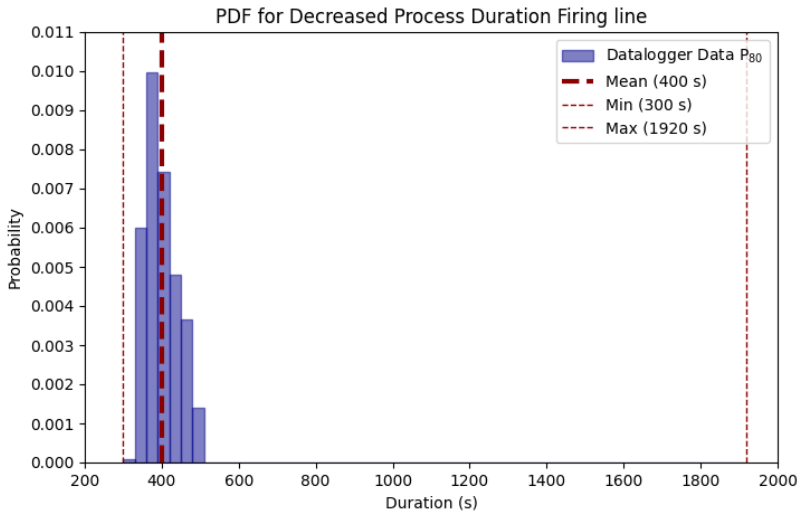


Figure 5.1: Probability density function for decreased process duration Firing line

Hold policy

As mentioned, with the addition of the buffer capacity limit, the pipe routing system now requires a sequencing strategy for the supply of the DJF to ensure the waiting area is alternately supplied with regular double joints and anode double joints. Longer simulation runs show that without such a strategy, deadlocks do in fact occur within the time frame intended for conducting the experiments. Figure 5.2 shows an example of a deadlock in the waiting area as the waiting area positions are all occupied by normal double joints and the buffer is completely filled with normal double joints, while the firing line requires an anode double joint. The hold policy is seen as the main experimental factor to influence sequencing as this policy regulates the amount of regular and anode joints that are not sent to the DJF and therefore regulates the ratio of regular and anode joints in the DJF. Three different hold policies have been evaluated, purely on their performance to prevent deadlocks:

- Regulate the anode ratio in the holds. If the anode ratio in the holds is lower than the pipeline anode ratio, no more regular joints are sent to hold. If the anode ratio in the holds is higher than the pipeline anode ratio, no more anode joints are sent to hold.
- Regulate the buffer level in the waiting area. If the amount of regular double joints in the buffer is higher than a certain level, no more anode joints are sent to hold. If the amount of anode double joints in the buffer is higher than a certain level, no more regular joints are sent to hold.
- Regulate the anode ratio between the marking stations and the firing line. All marked single joints in the pipe yard, DJF, waiting area and firing line are counted, including the single joints that are part of a double joint. The joints that are sent to hold or part of a rejected double joint are excluded. If the anode ratio of all counted single joints currently in the system is higher than the pipeline anode ratio, no more regular joints are sent to hold. If the anode ratio of all counted single joints currently in the system is lower than the pipeline anode ratio, no more anode joints are sent to hold.

The first policy seemed to perform well, but failed to prevent deadlocks over longer simulation runs. Possibly because it takes an indirect approach that does not directly monitor events in the DJF or waiting area. Similarly, the second policy was also ineffective in avoiding deadlocks during extended runs, as the delay between implementing the hold policy and its impact on the buffer level was too great, leading to a bullwhip effect. With the third policy implemented, the model does succeed to deliver continuous simulation results for extended runs. For now this policy is implemented in the pipe routing system as described. The policy may be further developed to improve its effect on other KPIs during the interactive search experimentation performed to establish routing improvements.

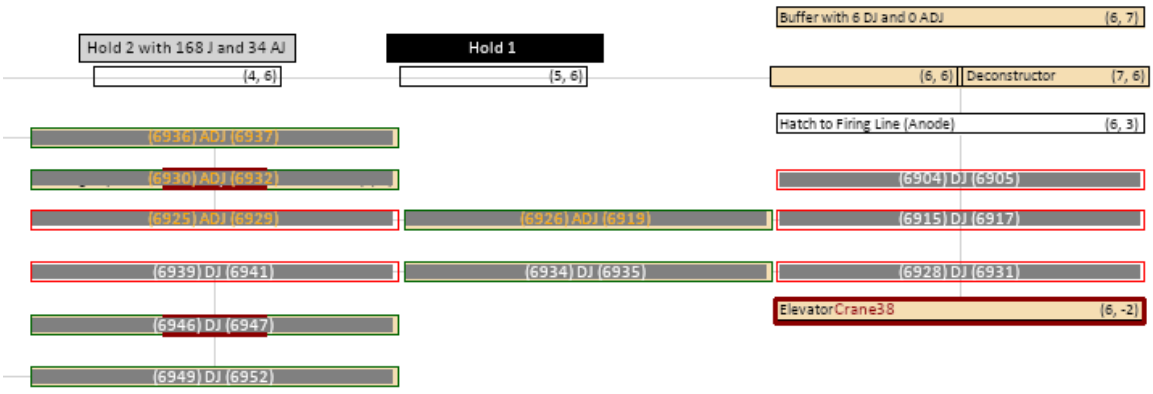


Figure 5.2: Example of a deadlock in the waiting area

5.3. Experimental plan

This section outlines the multi-step experimental plan that is followed to effectively search the solution space of different routing strategies and achieve the research objective. As mentioned the objective of the simulation experimentation is to evaluate different routing strategies under varying scenarios with the aim to provide a deeper understanding of the system rather than achieving a performance target.

The experimental plan includes three delivery scenarios (presented in Subsection 5.3.1) and six route configurations (presented in Subsection 5.3.2), resulting in a total of eighteen experiments. The experiments will be analysed individually through a non-terminating simulation run to conclude the mean output results during continuous pipelaying operations. The actual simulation run setup is detailed in Subsection 5.3.3.

This full set of experiments is conducted twice: first to evaluate the baseline performance, and later to assess the final performance of each configuration. Between these two phases, the route configurations are further refined by incorporating additional routing strategies within the same pathways. Rather than testing all possible combinations, each addition is evaluated individually to determine its specific impact on performance.

Multi-step experimental plan

1. In Subsection 5.3.1, the relevant pipe delivery scenarios are determined that contribute to the research objective of the experimentation phase. Subsection 5.3.2 provides an overview of the route configurations that will be evaluated. The combination of all scenarios and configurations form the total set of experiments that will be evaluated.
2. In Section 5.3.3, the base setup for the simulation runs is determined with an appropriate warm-up period and run-length to obtain accurate results.
3. In Section 5.4, the baseline performance for all scenarios and route configurations is established and the simulation output is visually analysed. Based on the outcome of the baseline results, each route configuration will be further developed in the following experimentation phase.
4. Section 5.5 describes the interactive search experimentation phase. In this phase, each route configuration is further developed with additional routing strategies and adjustments that are individually tested to asses their effectiveness to improve the baseline performance results.
5. To conclude the experimentation phase, Section 5.6, combines the effective routing strategies from the previous phase to form the final routing strategy for each route configuration. All scenarios and route configurations are evaluated once more through a final batch experiment to conclude the most effective routing strategy.

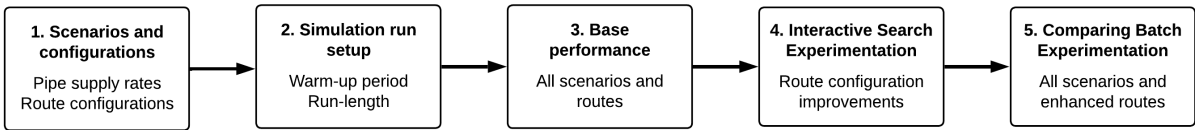


Figure 5.3: Summary of experimental plan

5.3.1. Experimental scenarios

As previously mentioned, there are four possible pipe delivery scenarios: a PSV at the port side, a PSV at the starboard side, a PSV at both sides, or no PSV. Although the initial plan was to investigate pipe delivery from the port side, starboard side, and both sides, the scope of the experiments is restricted to pipe delivery from the port side only. After running the first trial experiments it became evident that there a lot of experimental factors to consider and also small details within the experimental factors can have a significant impact on the results. Therefore the scenario limitation was adopted to enable a more elaborate exploration of a single delivery scenario, rather than a superficial exploration of all scenarios.

So far the verification and validation experiments have mostly been run with a realistic average PTC rate of 24 SJ/h. To validate the performance of a routing strategy it is also important to evaluate the effects of the routing strategy when the average PTC rate is lower or higher. A baseline performance simulation run for 5 different average PTC rates revealed the differences in output results shown in

Table 5.1. These results clearly show that very slow PTC rate of 15 SJ/h would be insufficient to supply the DJF continuously, while a slow PTC rate of 18 SJ/h is almost balanced with the DJF demand with only 3% of the single joints being sent to the hold. As insufficient PTC supply will be supplemented by supply from the holds in practice it is irrelevant to consider the very slow PTC rate. The slow PTC rate of 18 SJ/h will be included in the experimental scenarios to conclude the efficiency of the routing strategies when the supply and demand of single joints is balanced.

For the fast and very fast PTC rate the output shows similar results other than the PTC wait time. In fact both scenarios do not actually reach their PTC rate as the maximum throughput of the marking stations is limited at 26 SJ/h by the pipe flow in the pipe yard. The fast supply rate of 30 SJ/h, which is the actual design rate of the PTC, will be included in the experimental scenarios.

In conclusion, all routing strategies will be evaluated for three experimental scenarios:

- Pipe supply from port side with an average rate of 18 SJ/h
- Pipe supply from port side with an average rate of 24 SJ/h
- Pipe supply from port side with an average rate of 30 SJ/h

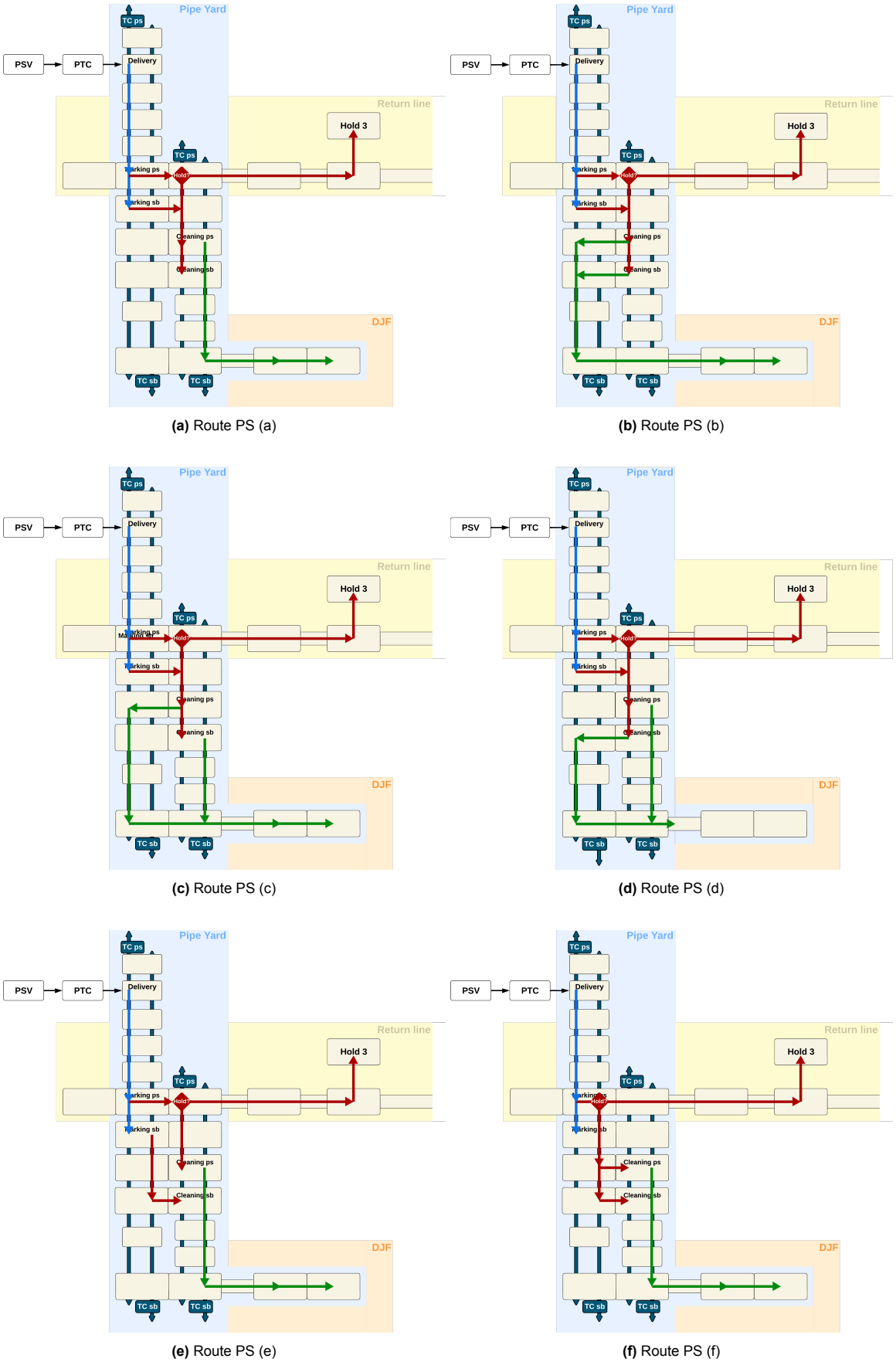
Table 5.1: Experimental scenarios

Data	Unit	Very slow 15 SJ/h	Slow 18 SJ/h	Average 24 SJ/h	Fast 30 SJ/h	Very fast 36 SJ/h
Sent to hold ratio	-	0%	3%	26%	32%	32%
DJF production	joints	165	207	208	207	208
Main pipeline length	m	3895	4917	4943	4924	4939
PTC wait time (KPI)	min	0	0	22	198	410
DJF wait time	min	687	67	56	46	51
Waiting area wait time (KPI)	min	128	14	11	10	9

5.3.2. Experimental configurations of the routing strategy

The experimental configurations are based on different routing options within the pipe yard. The route shown in Figure 4.7, previously applied for model verification and validation, is used to assess the baseline performance. This baseline configuration serves as a reference for evaluating the performance of the alternative route configurations. The baseline configuration is presented again in Figure 5.4a.

Figure 5.4 also present the five alternative route configurations that have been identified using the methodology described in Section 3.3.2. These six route configurations contain all routing strategies that move within the navigable space as defined in Figure 3.3a without moving against the pipe flow. The colours of the route sections represent the subprocesses that were also featured in Figure 3.3a



(a) Route PS (a)

(b) Route PS (b)

(c) Route PS (c)

(d) Route PS (d)

(e) Route PS (e)

(f) Route PS (f)

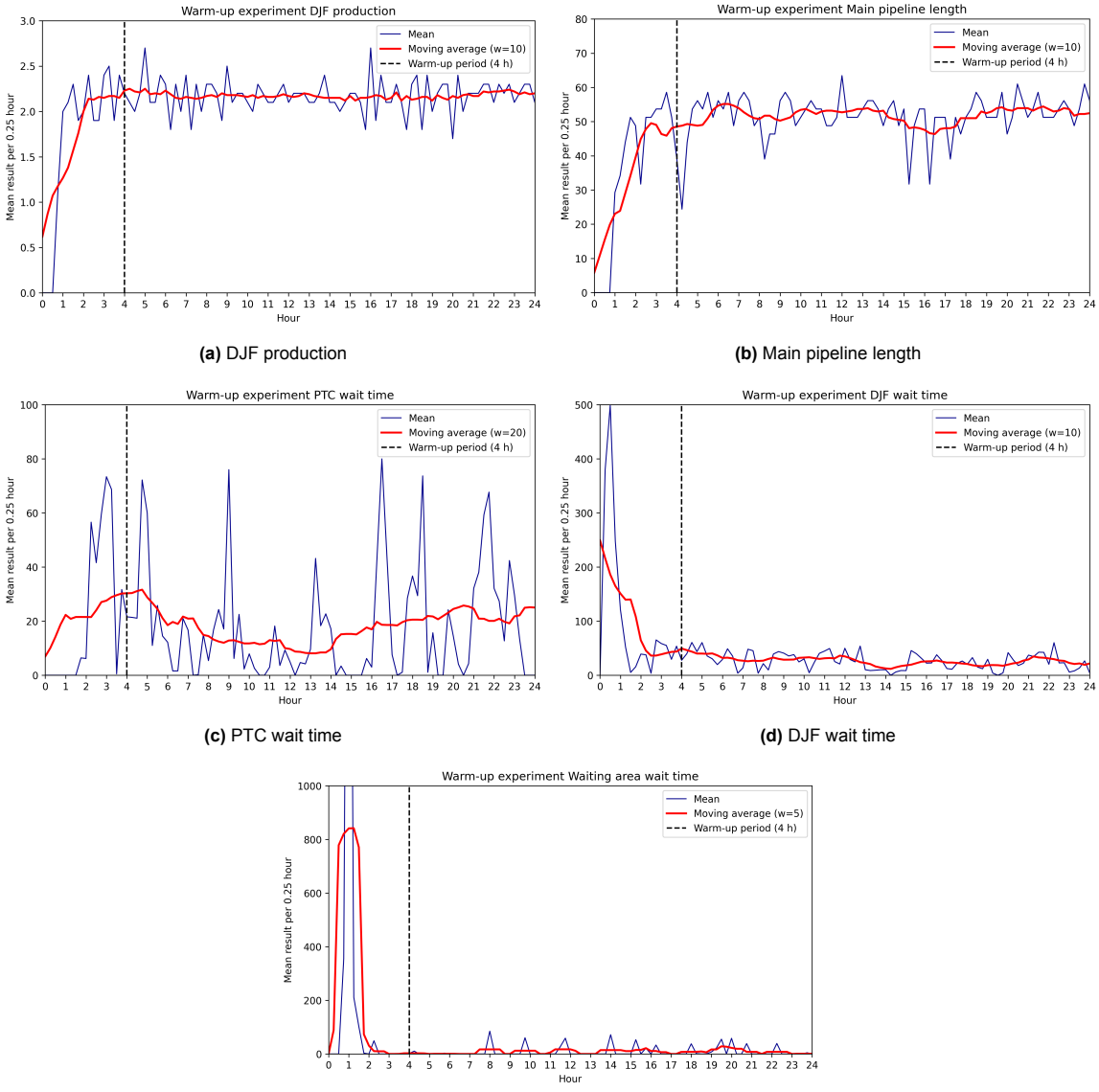
Figure 5.4: Routes for pipe supply from port side

5.3.3. Simulation run setup

This subsection will establish the appropriate warm-up period and run-length for the simulation runs to obtain accurate simulation results, followed by a procedure to process and present the simulation input and output for the experiments.

Warm-up period

As already mentioned in Section 4.3.2 the model output utilizes a warm-up period for data collection to address initialization bias. For the modelling phase the warm-up period was analytically calculated. For the experimentation phase the warm-up period is re-evaluated using Welch's method where a moving average is calculated over the mean output per period of a series of replications. Figure 5.5 shows the mean DJF production rate for each period of 0.25 hour based on 10 replications. The moving average is plotted with different window sizes per output parameter. For the PTC wait time the output per 0.25 hours was too random to draw a flattening moving average, however still a warm-up effect in the first two hours is clearly noticeable. Considering all output results it was concluded that a warm-up period of 4 hours would be appropriate for all output parameters.



(a) DJF production

(b) Main pipeline length

(c) PTC wait time

(d) DJF wait time

(e) Waiting area wait time

Figure 5.5: Warm-up period experiment for output parameters

Run length

As the simulation experiments will be run as a non-terminating simulation it is chosen to use a single long run rather than multiple replications. The appropriate run-length to obtain accurate results is determined by analysing the convergence of the output results for five simulation runs at different time periods. The level of convergence is calculated with equation 5.1, where C_i is the convergence at period i, and $\overline{Y}_{ij}$ is the cumulative mean of the output data at period i for replication j. The convergence is generally considered acceptable at a level of less than 5% (Robinson 2004).

$$C_i = \frac{Max(\overline{Y}_{i1}, \overline{Y}_{i2}, \overline{Y}_{i3}) - Min(\overline{Y}_{i1}, \overline{Y}_{i2}, \overline{Y}_{i3})}{Min(\overline{Y}_{i1}, \overline{Y}_{i2}, \overline{Y}_{i3})} \quad \text{(Robinson 2004)} \tag{5.1}$$

Similar to determining the warm-up period the DJF production, the main pipeline length and the wait time KPIs are considered to be the relevant output parameters. The results of all output parameters for all three replications are detailed in Appendix E and visually presented in Figure 5.6. The plots in Figure 5.6 are equally scaled with the y-axis being set from 0% to 200% of the mean output value to visualize the convergence level. This shows that although all simulation runs reach a constant output value, not all output parameters actually converge within the desired 5%.

Based on this observation it is concluded to adopt a run-length of 21 days (3 weeks) of continuous production as Figure 5.6 shows that all output values become constant around this period. For the evaluation of output results during experimentation it is important to consider the level of convergence for the output parameters at this period, which is summarized in Table 5.2. This table shows that the DJF production and main pipeline length are very accurate, but all the wait times require a higher relative deviation to be considered a significant performance difference.

Table 5.2: Run-length experiment - Level of convergence at day 21

Output Parameter	Unit	Min	Max	Convergence	Min. deviation
DJF production	joints	207	208	0.55%	1%
Main pipeline length	m	4877	4903	0.53%	1%
PTC wait time (KPI)	min	26	34	23.53%	30%
DJF wait time	min	48	55	12.73%	20%
Waiting area wait time (KPI)	min	9	13	30.77%	50%

Simulation input

In the initial trial experiments, it was observed that the number of rejected double joints at the DJF and firing line varied significantly, affecting the output results. To improve comparability, a constant seed was applied to the random reject probability of the NDT stations, ensuring a fixed number of rejects per experiment while keeping the specific rejected joints random. The remaining randomized inputs use a completely random seed for each experiment.

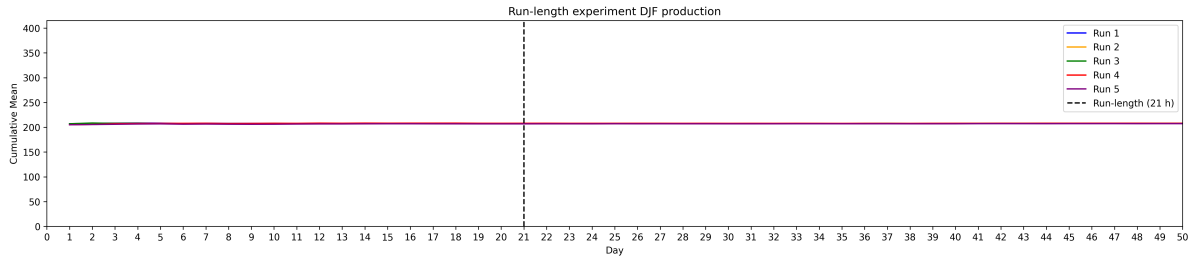
Simulation output

The simulation output for the experiments that is used for evaluation consists of three components:

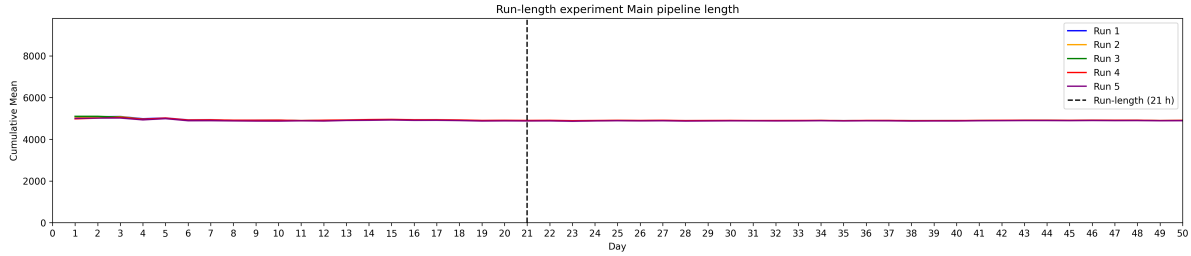
- Extensive data report of average simulation output per operating day.
- Snapshot of the simulation animation at the end of each operating day.
- Video of the simulation animation from the first operating hour.

A simulation run of 21 days with the data report and snapshots as output takes between 20 and 30 minutes to run when executed on a laptop with an Intel Core i7 processor (quad-core) and 8 GB of DDR4 RAM, running Windows 10 (64-bit). The animation video is produced in a separate simulation run. Appendix F provides the extensive data reports, accompanied by the final simulation snapshot. This report includes many output parameters, categorized by PS and SB as well as by normal and anode joints, in order to provide context for the KPI output parameters. Together with the snapshots and the animation video, this information illustrates the progression of the simulation.

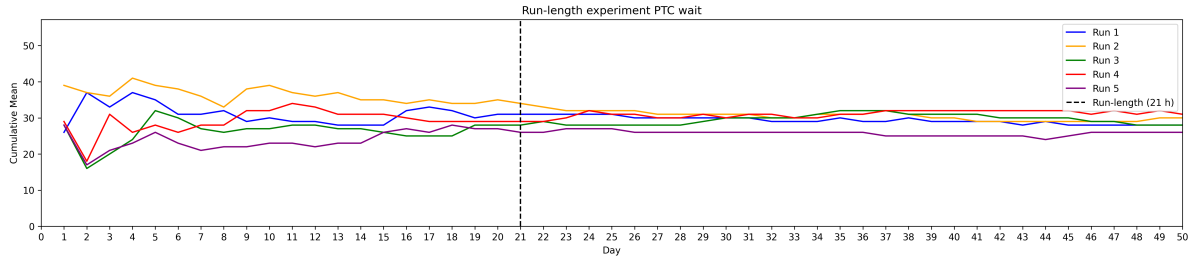
This main report only presents the main output parameters that were specified in Table 4.1, Section 4.2.4. The PTC wait time and the waiting area wait time form the actual KPIs of the system.



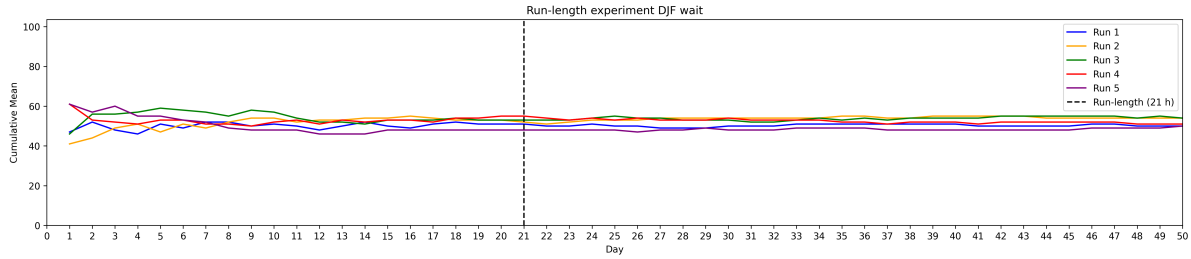
(a) DJF production



(b) Main pipeline length



(c) PTC wait time



(d) DJF wait time

(e) Waiting area wait time

Figure 5.6: Run-length experiment for output parameters

5.4. Baseline performance

This section presents the baseline performance of the pipe routing system with the base route configurations. The simulation output is observed to identify directions for further improvement that can be explored in the interactive search experimentation.

The comprehensive simulation output for the baseline performance are presented in Appendix F Section F.1. The main output results are summarized and evaluated below.

5.4.1. Reference simulation results

The simulation results for the baseline performance with route configuration (a) are presented in Table 5.3. These results serve as a reference for evaluating both the base configurations of the alternative routes and the final outcomes of the conclusive batch experimentation. The results show that no PTC waiting occurs under the slow PTC rate, but this output parameters increases as the PTC rate rises. All other output parameters remain largely unaffected across the different delivery scenarios.

The baseline performance results for the alternative route configurations are presented in Tables 5.4 to 5.8. Some differences are observed between routes, suggesting that the routing strategy indeed influences the output results. However, the differences are not sufficiently high to already draw conclusions on performance differences between routes.

5.4.2. Observations from baseline performance

The following paragraphs describe four routing strategy inefficiencies observed from the baseline performance experiments that can potentially be resolved during the interactive search experimentation.

Throughput at marking stations

As explained in Section 5.2, the hold policy is required to balance the anode ratio in the DJF and thereby prevent deadlocks in the waiting area. However, the baseline performance animations show that by restricting certain joints from moving to the hold, the current policy blocks throughput at the marking stations, resulting in a higher PTC wait time compared to the verification experiments (Table 4.3), where this policy was not applied.

Moreover, the baseline experiments revealed that deadlocks still occurred occasionally across all routes under a slow PTC rate. Since these baseline results serve as a reference for later analysis, the simulations were repeated with different random seeds. During the interactive search experimentation, the hold policy must be refined to fully prevent deadlocks and improve throughput at the marking stations.

Supply sequence for the waiting area

The waiting area waiting time is defined as the primary KPI, but the results show it already remains below 15 minutes per day across all routes. Simulation traces show the DJF generally supplies sufficient double joints to the waiting area, however the hold policy combined with stochastic process durations creates a random sequence of normal and anode joints for the supply. Consequently, delays occur when none of the available double joints in the waiting area are of the required type. A controlled distribution of anode joints between the port-side and starboard welding line may restructure the sequence in a way that reduces these delays

Movement between cleaning stations and the DJF

Another observed inefficiency in some routing strategies is the movement between the cleaning stations and the DJF. Routes (a), (b), (e), and (f) rely on a single supply path to the DJF, where consecutive movements of two single joints cause delays in supplying the DJF. A more efficient approach would be to line up two joints in the pipe yard and move them simultaneously into the DJF.

Routes (c) and (d) may also benefit from such a strategy, as the current right-of-way rules for the parallel supply paths to the DJF are based on joint number, which does not always match the order in which you would expect the joints to move.

Positioning of idle TCs

The final observation is that total transportation time could be reduced by relocating TCs to their next required position during idle periods, a benefit most evident under slow PTC rates. This would require the pipe routing system to reliably predict subsequent tasks to avoid unnecessary movements.

Table 5.3: Reference simulation results with base configuration for Route (a)

Output parameter	Unit	Slow PTC rate (18 SJ/h)	Avg PTC rate (24 SJ/h)	Fast PTC rate (30 SJ/h)
DJF production	joints / day	207	207	207
Main pipeline length	m / day	4886	4889	4880
PTC wait time (KPI)	min / day	0	69	291
DJF wait time	min / day	53	46	56
Waiting area wait time (KPI)	min / day	11	11	12

Table 5.4: Simulation results with base configuration for Route (b)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**	**Result**	**Dev.**	**Result**	**Dev.**
DJF production	joints / day	207	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4892	*0%*	4903	*0%*	4888	*0%*
PTC wait time (KPI)	min / day	0	*0%*	90	*30%*	323	*11%*
DJF wait time	min / day	63	*19%*	51	*11%*	46	*-18%*
Waiting area wait time (KPI)	min / day	11	*0%*	10	*-9%*	9	*-25%*

Table 5.5: Simulation results with base configuration for Route (c)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**	**Result**	**Dev.**	**Result**	**Dev.**
DJF production	joints / day	207	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4880	*0%*	4894	*0%*	4886	*0%*
PTC wait time (KPI)	min / day	0	*0%*	91	*32%*	324	*11%*
DJF wait time	min / day	56	*6%*	48	*4%*	50	*-11%*
Waiting area wait time (KPI)	min / day	12	*9%*	9	*-18%*	11	*-8%*

Table 5.6: Simulation results with base configuration for Route (d)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**	**Result**	**Dev.**	**Result**	**Dev.**
DJF production	joints / day	208	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4888	*0%*	4885	*0%*	4880	*0%*
PTC wait time (KPI)	min / day	1	*0%*	62	*-10%*	240	*-18%*
DJF wait time	min / day	54	*2%*	52	*13%*	54	*-4%*
Waiting area wait time (KPI)	min / day	11	*0%*	13	*18%*	16	*33%*

Table 5.7: Simulation results with base configuration for Route (e)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**	**Result**	**Dev.**	**Result**	**Dev.**
DJF production	joints / day	207	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4889	*0%*	4896	*0%*	4880	*0%*
PTC wait time (KPI)	min / day	1	*0%*	38	*-45%*	221	*-24%*
DJF wait time	min / day	53	*0%*	54	*17%*	48	*-14%*
Waiting area wait time (KPI)	min / day	10	*-9%*	11	*0%*	13	*8%*

Table 5.8: Simulation results with base configuration for Route (f)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**	**Result**	**Dev.**	**Result**	**Dev.**
DJF production	joints / day	207	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4880	*0%*	4893	*0%*	4882	*0%*
PTC wait time (KPI)	min / day	0	*0%*	20	*-71%*	138	*-53%*
DJF wait time	min / day	61	*15%*	50	*9%*	49	*-13%*
Waiting area wait time (KPI)	min / day	14	*27%*	10	*-9%*	15	*25%*

[1] Deviation from the baseline performance result presented in Table 5.3

5.5. Interactive search experimentation to improve configurations

Through an interactive search experimentation phase, several routing strategy adjustments and additions will be tested to gradually improve the route configurations performance results. Based on the observations from the baseline performance experiments in Section 5.4.2, the interactive search experimentation will explore strategy adjustments in four directions:

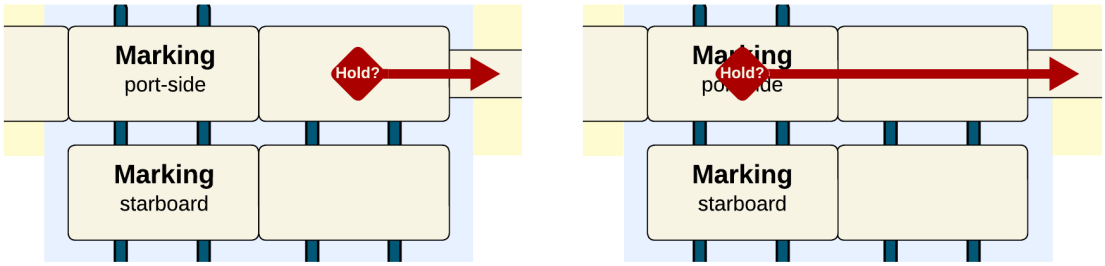
1. Regulate the anode ratio of joints between marking and the firing line through the hold policy.
2. Regulate the anode distribution between the port side and starboard welding line in the DJF.
3. Line up two single joints in the pipe yard for simultaneous supply to the DJF.
4. Add supportive TC moments without a task to transfer a joint.

As mentioned these routing amendments, which will be referred to as routing updates, are developed through interactive experimentation. The following subsections describe the interactive process that lead to the design of route update. The updates are sequentially added to the route configurations and each subsections concludes with a performance comparison between the previous version of each route configuration. The subsections only present essential output results to highlight differences between route configurations within the update.

5.5.1. Hold policy

Section 5.4.2 indicated that the hold policy requires further refinement, as deadlocks in the waiting area still occasionally occur under slow PTC rates, and the current policy blocks throughput at the marking stations. The current hold policy, as described in Section 5.2, is illustrated in Figure 5.7 and as shown it is divided in two policies: one for routes (a) till (e) with the hold decision on the position next to the port-side marking station and one for route (f) with the hold decision on port-side marking as this route configuration does not move to position next to marking.

The following two paragraphs will describe how this hold policy is adjusted to respectively solve the occurrence of deadlocks and reduce the PTC wait time. The final paragraph concludes the performance of the new hold policy for all route configurations.

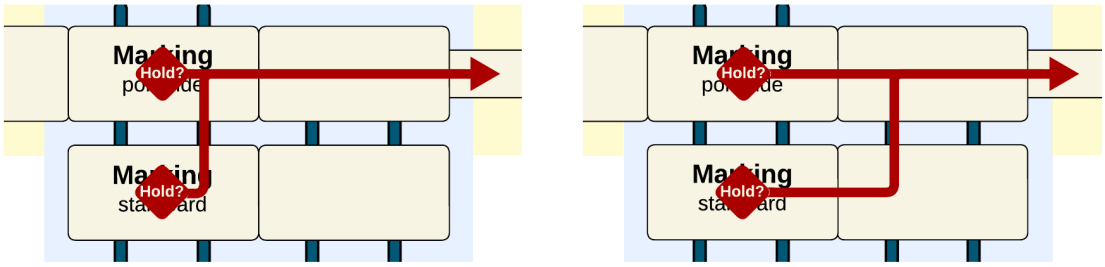


(a) Base hold policy for routes a, b, c, d, e **(b)** Base hold policy for route f

Figure 5.7: Hold policy for base route configurations

Hold policy adjustment to prevent deadlocks

The prevention of deadlocks is an essential feature of the hold policy as this is the only measure to regulate the ratio of normal and anode joints in the DJF. The occasional deadlocks occur only under slow PTC rates as the hold decision is only made on or next to the port-side marking station. Under slow PTC rates, most of the time there is no accumulation of joints in the pipe yard and as one marking station is enough to handle the throughput almost all joints move trough the starboard marking station. This means to be able to control the anode ratio within the system through the hold policy, the policy should be applied to both marking stations. For route configuration f this is easily applicable as the positions next to marking are not included in the route, while for the other route configurations the hold route has to move against the general pipe flow. Interactive experimentation revealed that the most efficient route for this would be to move a joint from the starboard marking station to the port-side marking station as this is the shortest route to follow the general pipe flow. This leads to the new routes to the hold as illustrated in Figure 5.8.



(a) New hold policy for routes a, b, c, d, e **(b)** New hold policy for route f

Figure 5.8: New hold policy for route configurations

Hold policy adjustment to reduce PTC wait time

The increased PTC wait time is caused by the old policy as it prevents a joint from moving to the hold when its type (normal or anode) is under-represented in the system's anode ratio, thereby causing congestion in the pipe yard. Through interactive experimentation, it was concluded that the hold policy should instead direct oversupplied joint types to the hold, rather than restricting undersupplied ones.

The first experimental solution for this was to decrease the anode count by two in the anode ratio calculation, allowing for two extra anodes in the system. Results showed that two extra anodes helped reducing the waiting area wait time as the waiting area is now mostly waiting for the more common normal double joints, where it was mostly waiting for anodes at first. The first experimental solution was to decrease the anode count used in the calculation by one, effectively allowing an additional anode in the system. This adjustment proved beneficial, as the waiting area results showed that the frequency of waiting for an anode became equal to that of waiting for a normal double joint, whereas previously waiting was predominantly caused by anodes.

However, deadlocks were still observed, making it necessary to also limit the number of normal joints sent to the DJF. The automatic movement from the port-side marking station to the hold in case of pipe yard congestion is retained. In addition, a normal joint can be forced to the hold from either marking station if the number of normal joints in the system exceeds a perfect anode ratio by more than six. This solution defines the final configuration of the hold policy as it fully prevents deadlocks.

Simulation output results

Since the hold policy is identical for routes (a)–(e), the deviations in output are also comparable. Therefore, only the simulation results of routes (a) and (f) are presented, respectively in Tables 5.9 and 5.10.

Table 5.9: Simulation results with enhanced hold policy for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4895	*0%*	4883	*0%*	4887	*0%*
PTC wait time (KPI)	min / day	0	*0%*	1	*-99%*	21	*-93%*
DJF wait time	min / day	128	*+142%*	56	*+22%*	45	*-20%*
Waiting area wait time (KPI)	min / day	5	*-55%*	12	*+9%*	9	*-25%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.10: Simulation results with enhanced hold policy for Route (f)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	208	*0%*
Main pipeline length	m / day	4885	*0%*	4890	*0%*	4893	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	0	*-100%*
DJF wait time	min / day	154	*+152%*	46	*-8%*	53	*+8%*
Waiting area wait time (KPI)	min / day	9	*-36%*	9	*-10%*	10	*-33%*

[1] Deviation from the base configuration results for route (f) presented in Table 5.8

Conclusion for hold policy

The results in Table 5.9 and Table 5.10 demonstrate that the enhanced policy substantially reduces the PTC waiting time for route (a), and even fully eliminates it under all PTC rates for route (f). The greater effectiveness of the hold policy in route (f) is due to the fact that joints can always be directed to the hold without moving against the pipe flow. By contrast, in the other routes, a joint sent to the hold from the starboard marking station must move against the pipe flow through the port-side marking station, which can cause some congestion near the PTC.

Another observation is that the DJF waiting time increases considerably under slow PTC rates in both simulation results. This happens because the new policy sends joints to the hold to maintain the anode ratio, even when the pipe yard is empty and the PTC delivery rate is barely sufficient to meet DJF demand. While this may appear counter-intuitive, the policy is necessary to avoid deadlocks in the waiting area. Moreover, the simulations show that the higher DJF waiting time is not large enough to also cause an increase in the waiting area waiting time.

In conclusion, the proposed hold policy is an improvement to all route configurations and will also be implemented for the following experiments.

5.5.2. DJF anode distribution

In Section 5.4, it was proposed that regulating the distribution of anode double joints between the two welding lines in the DJF may affect the waiting area waiting time. As the relation between the distribution and the waiting time is unclear, three completely different distribution strategies are tested:

1. One-sided distribution
2. Alternating distribution
3. Non-adjacent distribution

The following paragraphs present the simulation results for each distribution strategy applied to the enhanced hold policy configuration of route (a).

One sided distribution

In the one-sided distribution strategy all anode double joints are sent to the port-side welding line. The simulation results in Table 5.11 show that the waiting area wait time is increased instead of reduced and the distribution causes delays in the pipe yard.

Table 5.11: Simulation results with one-sided anode distribution for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	206	*0%*	206	*0%*	207	*0%*
Main pipeline length	m / day	4861	*-1%*	4868	*0%*	4868	*0%*
PTC wait time (KPI)	min / day	0	*0%*	3	*+200%*	39	*+86%*
DJF wait time	min / day	449	*+251%*	400	*+614%*	382	*+749%*
Waiting area wait time (KPI)	min / day	14	*+180%*	13	*+8%*	14	*+56%*

[1] Deviation from the enhanced hold policy performance result presented in Table 5.9

Alternating distribution

In the alternating distribution strategy anode double joints are alternately sent to the port-side and starboard welding line. The simulation results in Table 5.12 show that the waiting area wait time is not significantly reduced and the distribution causes delays in the pipe yard.

Table 5.12: Simulation results with alternating anode distribution for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4879	*0%*	4878	*0%*	4889	*0%*
PTC wait time	min / day	0	*0%*	2	*+100%*	37	*+76%*
DJF wait time	min / day	395	*+209%*	317	*+466%*	313	*+596%*
Waiting area wait time (KPI)	min / day	10	*+100%*	9	*-25%*	8	*-11%*

[1] Deviation from the enhanced hold policy performance result presented in Table 5.9

Non-adjacent anodes

In the non-adjacent distribution strategy anode double joints are sent to the opposite welding line when they would be positioned directly behind another anode double joint. The simulation results in Table 5.13 show that the waiting area wait time is increased instead of reduced and the distribution causes delays in the pipe yard.

Table 5.13: Simulation results with non-adjacent anode distribution for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	206	*0%*	206	*0%*
Main pipeline length	m / day	4877	*0%*	4858	*-1%*	4859	*-1%*
PTC wait time	min / day	0	*0%*	3	*+200%*	43	*+105%*
DJF wait time	min / day	439	*+243%*	439	*+684%*	406	*+802%*
Waiting area wait time (KPI)	min / day	11	*+120%*	13	*+8%*	15	*+67%*

[1] Deviation from the enhanced hold policy performance result presented in Table 5.9

Conclusion for DJF anode distribution

The simulation results show that none of the distribution strategies reduce the waiting area wait time significantly. In fact the distributions mostly cause and in crease in the wait time, as well as significant extra delay of pipe movement in the pipe yard. As the proposed strategies were completely different it is concluded that the DJF anode distribution is not an interesting experimental factor to further explore and the proposed strategies will not be implemented for the following experiments.

5.5.3. DJF supply line

The DJF supply line is introduced as an experimental factor to reduce DJF waiting time. Although not defined as a KPI, this waiting time could be decreased by lining up two single joints in the pipe yard while both welding lines of the DJF are occupied. The line up will also prevent that two anode single joints are not welded together. The six base route configurations provide three different pathways to the DJF: route (b) follows the marking stations line, routes (a), (e) and (f) follow the cleaning stations line, and routes (c) and (d) combine both infeed lines. The following paragraphs therefore present three different solutions that will be applied to the route configurations, as well as their corresponding simulation results.

DJF supply line for routes (a), (e) and (f)

As shown in Figure 5.9, the infeed line to the DJF for route (a), (e) and (f) moves along the cleaning stations line $(1, y)$, which means in the base configuration routes only one single joint could be lined up in front of the supply positions $(3, 0)$ and $(4, 0)$.

When either DJF supply position $(3, 0)$ or $(4, 0)$ is unoccupied, the joints proceed directly to the available position as long as this does not generate a pair of anode joints. If both supply positions are occupied, a joint moves to $(0, 0)$ to form a line-up for the next pair of single joints, which will collectively move to the supply positions as soon as they become available. If both single joints in the line-up are anodes, only the joint at $1, 0)$ advances to the supply positions, followed by another joint from $(1, 1)$.

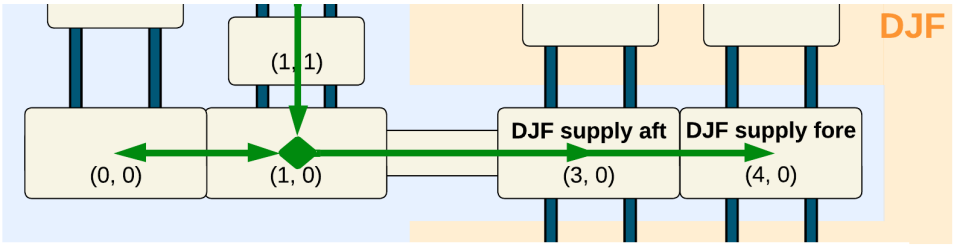


Figure 5.9: DJF supply line for routes (a), (e) and (f)

When this DJF supply line strategy is incorporated into the final hold policy configuration of route (a), the results presented in Table 5.14 are obtained. The result deviations in output are similar for route (e) and (f) as their are no difference in this particular path section for these route configurations.

Table 5.14: Simulation results with DJF supply line for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4879	*0%*	4879	*0%*	4890	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	14	*-33%*
DJF wait time	min / day	50	*-61%*	36	*-36%*	32	*-29%*
Waiting area wait time (KPI)	min / day	9	*+80%*	9	*-25%*	9	*0%*

[1] Deviation from the hold policy configuration results for route (a) presented in Table 5.9

DJF supply line for route (b)

As shown in Figure 5.10, the infeed line to the DJF for route (b) moves along the marking stations line $(0, y)$. In this base route configuration two single joints automatically got lined up, however the single joints did not move collectively as a pair and there was no correction for a pair of anode joints.

When either DJF supply position $(3, 0)$ or $(4, 0)$ is unoccupied, the joints still proceed directly to the available position as long as this does not generate a pair of anode joints. If both supply positions are occupied, a pair of joints is formed in the line-up. When both single joints in the line-up are anodes, the joint at position $(1, 0)$ is moved to $(1, 1)$. This joint is then reintroduced into the line-up as soon as position becomes available.

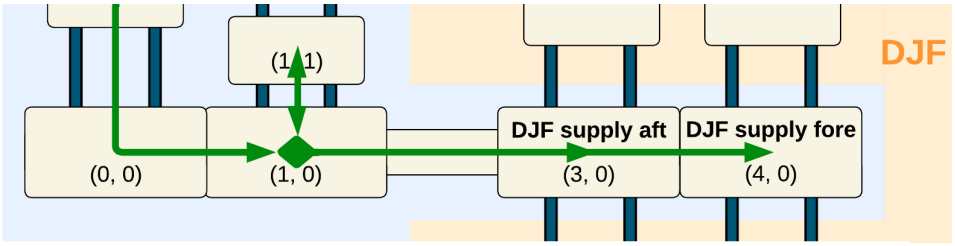


Figure 5.10: DJF supply line for route (b)

When this DJF supply line strategy is incorporated into the final hold policy configuration of route (b), the results presented in Table 5.14 are obtained.

Table 5.15: Simulation results with DJF supply line for Route (b)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	208	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4896	*+1%*	4887	*0%*	4882	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*0%*	19	*+12%*
DJF wait time	min / day	68	*-62%*	36	*-29%*	36	*-31%*
Waiting area wait time (KPI)	min / day	8	*-38%*	9	*0%*	7	*-46%*

[1] Deviation from the hold policy configuration results for route (b)

DJF supply line for routes (c) and (d)

As shown in Figure 5.11, route (c) and (d) use both infeed lines $(0, y)$ and $(1, y)$. Of course this means also these base route configurations already line up two single joints, however also in these configurations there was no collective movement and no correction for pairs of anode joints.

When either DJF supply position $(3, 0)$ or $(4, 0)$ is unoccupied, again the joints proceed directly to the available position as long as this does not generate a pair of anode joints. If both supply positions are occupied, a pair of joints is formed in the line-up from both infeed lines. When both single joints in the line-up are anodes, only the joint at $1, 0)$ advances to the supply positions, followed by another joint from infeed line $(1, y)$.

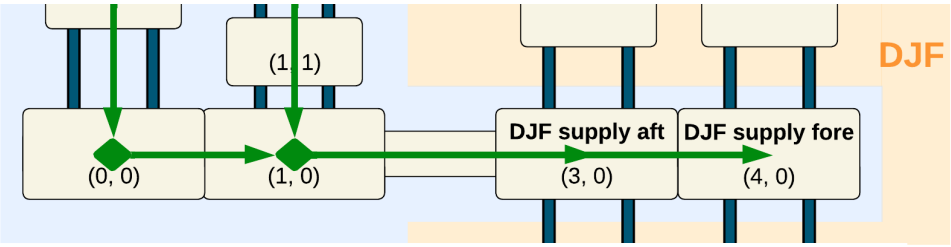


Figure 5.11: DJF supply line for route (cd)

Since the infeed lines of route configurations (c) and (d) are not fully identical, their output results also differ. When the DJF supply line strategy is applied to the final hold policy configuration of route (c), the results shown in Table 5.16 are obtained. For configuration (d), however, the DJF supply line strategy leads to deadlocks in the pipe system as it prevents the combination of two anode single joints. Figure 5.12 illustrates a recurring situation, where the anode joint from infeed line $(0, y)$ cannot move due to another anode joint already occupying the DJF supply position. The DJF supply line then waits for a joint from infeed line $(1, y)$, but this infeed line is blocked by the crossing pathways of this route.

Table 5.16: Simulation results with DJF supply line for Route (c)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	208	*0%*
Main pipeline length	m / day	4871	*0%*	4879	*0%*	4892	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	23	*+21%*
DJF wait time	min / day	110	*+38%*	32	*-40%*	35	*-24%*
Waiting area wait time (KPI)	min / day	12	*+33%*	11	*+38%*	8	*-11%*

[1] Deviation from the hold policy configuration results for route (c)

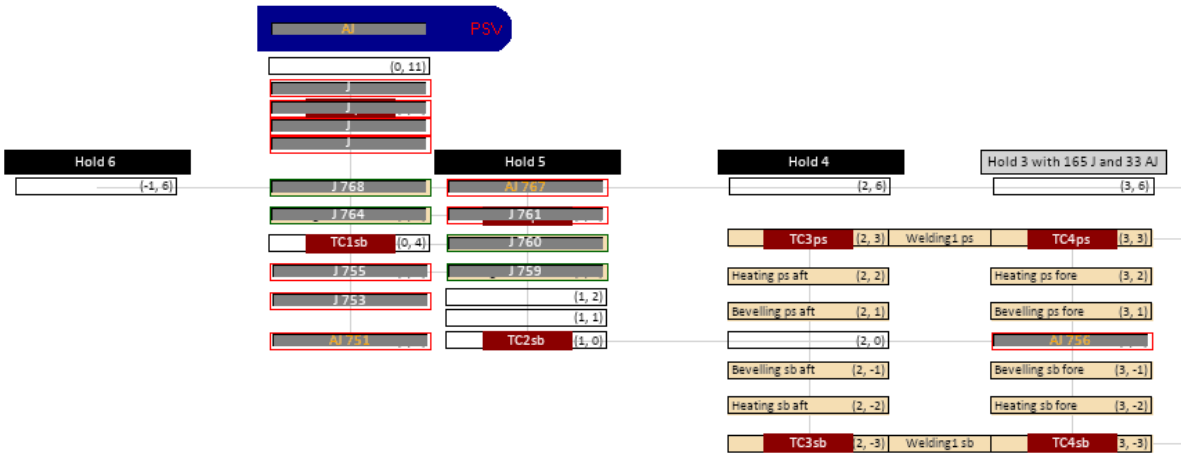


Figure 5.12: Deadlock in pipe yard with DJF supply line for route (d))

Conclusion for DJF supply line

The simulation results from Tables 5.14, 5.15 and 5.16 all show a comparable decreased DJF wait time for average and fast PTC rates as expected. Under slow PTC rates however there is a significant decrease observed in Tables 5.14 and 5.15, while Table 5.16 shows an increase in DJF wait time. In prior experiments it was already observed that the convergence of this parameter seems inconsistent, so therefore we cannot draw reliable conclusion based on these results.

Table 5.14 also shows a significant decrease of the PTC wait time. This could be explained by the line-up in this route increasing the amount of positions between cleaning and the DJF by one, making the amount of positions even. This means the joints can move in pairs, making for a more efficient pipe

flow.

In conclusion the proposed DJF supply line strategies will be implemented for all route configuration but route (d) as they show to be beneficial for the DJF wait time and do not negatively impact other route parameters. For route configuration their is no DJF supply line implemented, the latest update of this configuration is still the adjustment of the hold policy.

5.5.4. TC support moves

The final experimental factor investigated to improve the route configurations is the implementation of TC support moves to decrease the PTC and DJF wait time. A TC support move involves repositioning idle TCs when no there are no more tasks to assign. The support moves that contribute positively to system performance are identified through analysis of the simulation animations in combination with interactive experimentation. The pipe routing system applies a similar algorithm for assigning TC support moves across all TCs and route configurations.

Each TC is assigned a list of occupancy positions and a designated standard position. When a TC becomes idle with no remaining tasks, the system first checks the occupancy of the positions in the assigned list. The TC is then moved to the first position in the list that is currently occupied by a joint, as this is most likely to generate the next transport task. If none of the positions are occupied, the TC is relocated to its standard position.

The set of occupancy positions and the standard position differ for each route configuration. Figure 5.13 illustrates this by colouring all occupancy positions associated with a TC, while the TC itself is placed at its standard position. The following paragraph presents the occupancy lists and standard positions for route (a), together with the output results obtained from incorporating these TC support moves. Only the simulation results from route (a) are presented as the TC support moves are very similar across al route configurations.

TC support moves for route (a)

The list of occupancy positions and the designated standard position for each TC in route (a), that is illustrated in Figure 5.13a, is summarized in Table 5.17. The standard position of the starboard TC on line $(1, y)$ is out side of the cleaning stations as it would otherwise block the movement of the port-side TC, however if a cleaning station is occupied the port-side TC will not be moving towards this position.

The simulation results with TC support moves, compared to the previous DJF supply line configuration results are presented in Table 5.18.

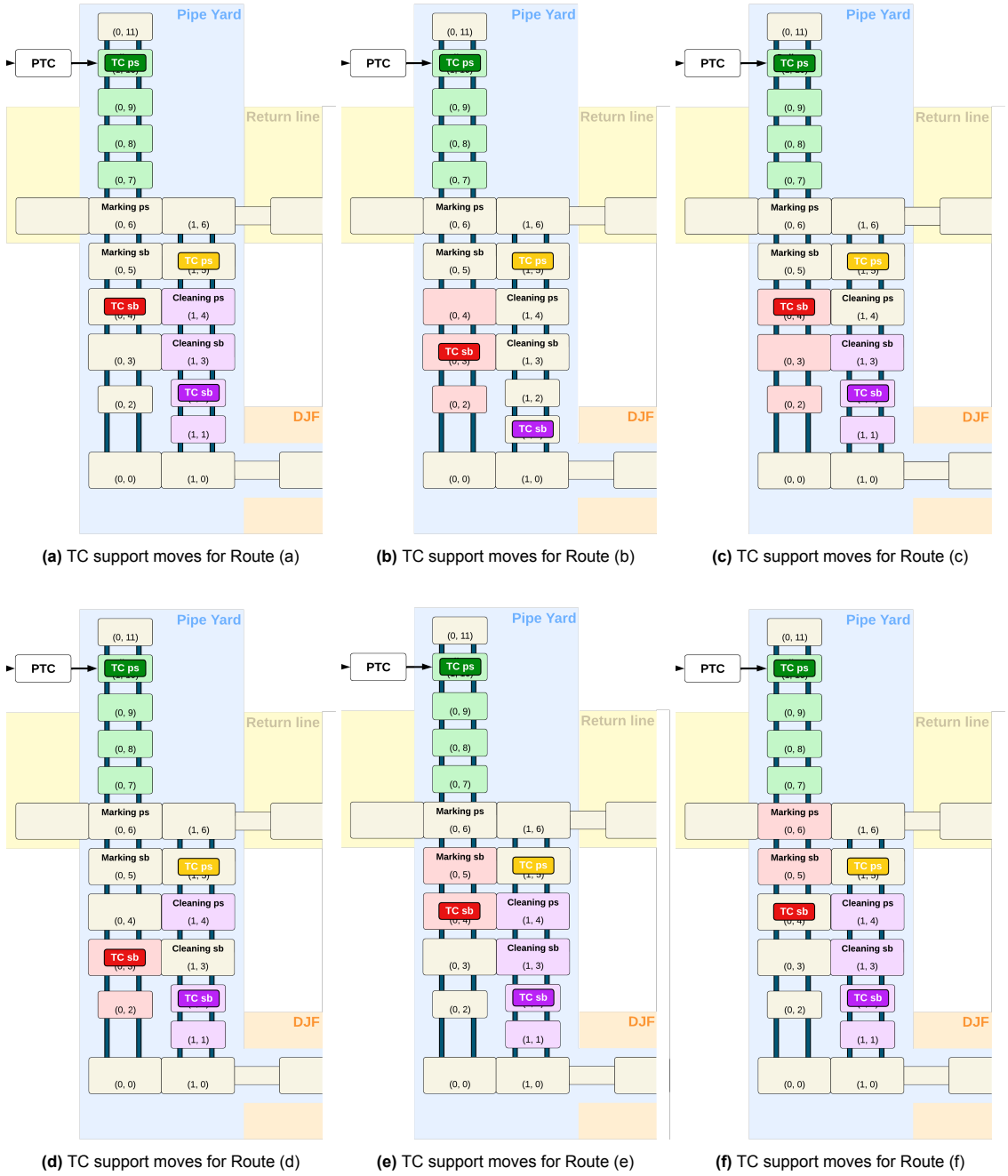
Table 5.17: TC support moves for Route (a)

Transverse Car	Colour	Occupancy positions	Standard position
TC $(0, y)$ port-side	Green	$(0, 7)$, $(0, 8)$, $(0, 9)$, $(0, 10)$	$(0, 10)$
TC $(0, y)$ starboard	Red		$(0, 5)$
TC $(1, y)$ port-side	Yellow		$(1, 5)$
TC $(1, y)$ starboard	Purple	$(1, 1)$, $(1, 2)$, $(1, 3)$, $(1, 4)$	$(1, 2)$

Table 5.18: Simulation results with TC support moves for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4885	*0%*	4895	*0%*	4882	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*0%*	9	*-36%*
DJF wait time	min / day	57	*+14%*	34	*-6%*	34	*+6%*
Waiting area wait time (KPI)	min / day	9	*0%*	8	*-11%*	8	*-11%*

[1] Deviation from the DJF supply line configuration results for route (a) presented in Table 5.14



(a) TC support moves for Route (a) **(b)** TC support moves for Route (b) **(c)** TC support moves for Route (c)

(d) TC support moves for Route (d) **(e)** TC support moves for Route (e) **(f)** TC support moves for Route (f)

Figure 5.13: TC support moves for all route configurations

Conclusion for TC support moves

Table 5.18 shows that the PTC rates decrease as expected due to the premature movement of the TC to the correct position. The DJF waiting time does not show notable differences, but analysis of the simulation animations confirms that the other TCs also correctly anticipate their next required position. This effect is not reflected in the results because TC movement speed was not a limiting factor influencing DJF waiting times. Overall, it can be concluded that the proposed TC support moves are beneficial for all routing configurations.

5.5.5. Conclusion of the interactive search experimentation

To conclude the interactive search experimentation, all explored routing strategy adjustments an additions are summarized in Table 5.19 with their goals, results and implementation for the conclusive batch experimentation.

Table 5.19: Conclusions for routing strategies from interactive search experimentation

Enhanced hold policy	
Goal	Prevent deadlocks in waiting area and reduce PTC wait time
Result	Completely prevents deadlocks in the waiting area. Fully eliminates PTC wait time for all routes under average PTC rates, and for route (f) even under fast PTC rates. For other routes a small PTC wait time remains for fast PTC rates.
Implementation	All route configurations
DJF anode distribution	
Goal	Reduce waiting area wait time
Result	Three different distribution strategies all had no significant impact on the waiting area wait time.
Implementation	None
DJF supply line	
Goal	Prevent combination of two anode single joints and reduce DJF wait time
Result	Reduces DJF wait time for all routes, but the prevention anode single joint combinations causes deadlocks in the pipe yard for route configuration (d).
Implementation	All route configurations but route (d)
TC support moves	
Goal	Reduce PTC wait time and DJF wait time
Result	Further reduces PTC wait time for all route configurations. Does not actually reduce the DJF wait time, but animation show that it does improve movement efficiency.
Implementation	All route configurations

5.6. Conclusive batch experimentation

A final batch of experiments is conducted in which all updates from the interactive search experimentation are implemented in the route configurations. Tables 5.20 to 5.25 present the performance results of each configuration, benchmarked against the reference base performance shown in Table 5.3.

The results demonstrate that all configurations achieve comparable performance with respect to the KPIs. Each configuration eliminates PTC wait time under slow and average PTC rates, while route (f) distinguishes itself by also eliminating waiting time under fast PTC rates. The waiting area wait time is not significantly reduced. However, since this parameter was already relatively low for the base performance and it is consistently lower across all final configurations, it is concluded that the performance of all configurations is satisfactory and that this KPI also provides limited potential for further improvement.

Table 5.20: Simulation results with final configuration for Route (a)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	208	*0%*	208	*0%*	207	*0%*
Main pipeline length	m / day	4896	*0%*	4894	*0%*	4890	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	7	*-98%*
DJF wait time	min / day	96	*+81%*	33	*-28%*	32	*-43%*
Waiting area wait time (KPI)	min / day	7	*-36%*	7	*-36%*	8	*-33%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.21: Simulation results with final configuration for Route (b)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4883	*0%*	4886	*0%*	4868	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	19	*-93%*
DJF wait time	min / day	81	*+53%*	35	*-24%*	32	*-43%*
Waiting area wait time (KPI)	min / day	13	*+18%*	10	*-9%*	9	*-25%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.22: Simulation results with final configuration for Route (c)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	208	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4901	*0%*	4873	*0%*	4881	*0%*
PTC wait time (KPI)	min / day	0	*0%*	1	*-99%*	15	*-95%*
DJF wait time	min / day	53	*0%*	35	*-24%*	34	*-39%*
Waiting area wait time (KPI)	min / day	10	*-9%*	9	*-18%*	10	*-17%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.23: Simulation results with final configuration for Route (d)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	208	*0%*	208	*0%*	208	*0%*
Main pipeline length	m / day	4899	*0%*	4899	*0%*	4901	*0%*
PTC wait time (KPI)	min / day	0	*0%*	1	*-99%*	16	*-95%*
DJF wait time	min / day	92	*+74%*	52	*+13%*	51	*-9%*
Waiting area wait time (KPI)	min / day	7	*-36%*	10	*-9%*	6	*-50%*
Firing line wait time	min / day	28	*-7%*	30	*+3%*	24	*-25%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.24: Simulation results with final configuration for Route (e)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	208	*0%*	207	*0%*	207	*0%*
Main pipeline length	m / day	4897	*0%*	4890	*0%*	4887	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	18	*-94%*
DJF wait time	min / day	45	*-15%*	34	*-26%*	34	*-39%*
Waiting area wait time (KPI)	min / day	9	*-18%*	6	*-45%*	9	*-25%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

Table 5.25: Simulation results with final configuration for Route (f)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	208	*0%*
Main pipeline length	m / day	4889	*0%*	4890	*0%*	4893	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	0	*-100%*
DJF wait time	min / day	40	*-25%*	34	*-26%*	33	*-41%*
Waiting area wait time (KPI)	min / day	9	*-18%*	8	*-27%*	8	*-33%*

[1] Deviation from the base configuration results for route (a) presented in Table 5.3

5.7. Conclusion

This section described simulation experiments conducted to evaluate different routing strategies under varying scenarios, aiming to provide a deeper understanding of the system, rather than meet a fixed performance target. Prior to the experiments, a maximum buffer capacity was added to ensure realistic results for longer runs, which also required an adjusted hold policy to prevent deadlocks in the waiting area by regulating the ratio of anode joints in the DJF.

The experimental plan consisted of three experiment phases:

1. Base performance batch experimentation
2. Interactive search experimentation
3. Conclusive batch experimentation

The batch experimentation phases were performed with three PTC delivery rate scenarios and six route configurations, resulting in a total of 18 experiments. Between the batch experimentation phases the route configuration were improved through interactive search experimentation based on the information provided by the base performance simulation results.

The system performance for each experiment was evaluated based on five output parameters: DJF production, main pipeline length, PTC wait time, DJF wait time, and waiting area wait time. The PTC wait time and waiting area wait time form the actual KPIs of the system. Overall, it was concluded that a well-balanced hold policy is the most impactful measure for improving system performance, as it is essential to prevent deadlocks in the waiting area and has the greatest influence on the PTC waiting time KPI. There were no effective measures identified to eliminate the waiting area wait time KPI, but constantly storing two extra anode joints within the system showed a slight decrease.

The conclusive batch experimentation revealed that the interactive search experimentation has significantly improved the performance of all route configuration. It was concluded that route configuration (f) performs the best based on the set KPIs as it is the only route configuration that fully eliminates the PTC wait time for all delivery rate scenarios, due to the fact that joints can always be directed to the hold without moving against the pipe flow. By contrast, in the other routes, some joints sent to the hold must move against the pipe flow which can cause some congestion near the PTC.

Only route configuration (d) is rejected as a solution as the crossing pathways of the route lead to deadlocks in the pipe yard for a desired element of the routing strategy.

Apart from these observation the differences between the route configuration were not significant and all routes performed well as for all configuration the wait times under average PTC rates cannot not be improved much more. There for route configuration (a), (b), (c), (e) and (f) are all concluded to be appropriate routes, with route configuration (f) performing the best for fast PTC rates.

6

Conclusion

This research explored the feasibility of automating joint movements on a pipelaying vessel. A conceptual pipe routing system was developed and implemented in a model of the pipe handling system on Allseas' vessel *Solitaire*, to gain insight into system behaviour, and identify an effective routing solution.

Analysis of the pipelaying operations on *Solitaire* formulated the pipe handling system. The key performance indicators (KPI) for system performance are the firing line waiting time, reflecting continuity of supply, and the pipe transfer crane (PTC) waiting time, reflecting pipe supply vessel turnaround time. The routing system is implemented as an intermediate control layer that translates database information into movement requests for the pipe handling PLC, without directly controlling equipment.

The pipe handling system is modelled using Discrete Event Simulation (DES) with Salabim. The routing problem is defined as a sequential decision-making problem, suited for future optimization through Reinforcement Learning (RL). In this study it is solved with rule-based heuristics within a model-free Markov Decision Process framework, while the solution space is reduced by dividing the routing problem into subproblems and limiting the action space accordingly.

The DES model incorporates scaled dimensions and actual equipment specifications. Pipeline-specific data is based on the South East Extension project, with missing process distributions generated through a self-conducted data analysis. Simplifications were introduced for the hold system and waiting area, with the firing line waiting time substituted for waiting area waiting time, in order to focus on routing strategies in the pipe yard.

A near-optimal routing strategy is developed through experimentation with the model. Three PTC rate scenarios and six base route configurations have been evaluated. Initial simulation results showed similar performance across all routes. Interactive search experimentation then improved performance through an enhanced hold policy, pre-lining of joint pairs in the pipe yard, and TC support moves.

The strategy updates significantly improved performance for all route configurations, however, the final batch experiments identified route (f) as the most effective strategy with the results presented in Table 6.1. With this configuration the PTC wait time is fully eliminated, whereas other routes retained approximately 15 min/day under fast PTC rates. This improvement results from route (f) allowing joints to move towards the hold without moving against the pipe flow. Reductions in waiting area waiting time were minor and comparable across all configurations. Overall, all route configurations achieved satisfactory system performance, with little room for further improvement given the KPIs defined in this study. Only route configuration (d) is rejected as a solution due to reoccurring deadlocks.

Table 6.1: Simulation results for the final route configuration (f) compared to the baseline performance

		Slow PTC rate (18 SJ/h)			**Avg PTC rate (24 SJ/h)**			**Fast PTC rate (30 SJ/h)**		
Output parameter	**Unit**	**Base**	**Route (f)**	**Dev.**	**Base**	**Route (f)**	**Dev.**	**Base**	**Route (f)**	**Dev.**
PTC wait time (KPI)	min / day	0	0	*0%*	69	0	*-100%*	291	0	*-100%*
DJF wait time	min / day	53	40	*-25%*	46	34	*-26%*	56	33	*-41%*
Waiting area wait time (KPI)	min / day	11	9	*-18%*	11	8	*-27%*	12	8	*-33%*

This research demonstrated that simulation provides a deeper understanding of the pipe handling system and the impact of routing strategies. The heuristic approach successfully automated sequential decision-making and regulated joint movements. Nevertheless, the main challenge that remains is full scenario coverage, given the wide variation in pipeline specifications, environmental influences, and unpredictable events. A future pipe routing system based on RL could potentially overcome this by adapting to all scenarios and identifying solutions beyond those considered in the heuristic approach.

7

Recommendations

Scientific recommendations

The pipelaying industry, and pipe handling systems in general, are rarely addressed in literature. A key distinction from systems like automated container terminals or automated guided vehicles, is that in the pipe handling system the movable products are identical with no predefined destinations, as stations require a number of products, but no specific ones. Combined with stochastic processes, this makes forecasting multiple routing decisions impractical. More research can be published on similar systems that operate under an unpredictable horizon.

This study considered evolution to control by a RL model, by formulating the problem within a model-free MDP framework but solving it with rule-based heuristics. In the current setup, the pipe routing system is presented with a single action to perform; however, this could be extended to a list of possible actions from which an RL model would select. The performance of such an RL model could then be directly compared to the rule-based heuristics, as both are implemented in the same simulation model.

Since RL is expected to play an important role in the continuous improvement of automated systems, it would be interesting to see if this framework indeed supports the development of RL models for complex intralogistics and pipelaying systems, as this could advance the practical application of RL.

Practical recommendations for Allseas

In general, it is recommended that Allseas adopt simulation technology in their preparation procedures for pipelaying projects. This study has shown that simulation can provide a deeper understanding of deck operations and generate practical insights that would otherwise be left to interpretation by deck operators. The Salabim model presented here demonstrates this potential, but its assumptions and simplifications should be replaced to achieve a realistic representation of the actual system.

During this study it also became clear that insufficient data is currently available on the processing stations within the DJF to perform a detailed simulation analysis. It is therefore recommended that Allseas evaluate the future role of simulation studies within the company and, if deemed valuable, invest in the collection of more accurate process data aligned with the goals of future analyses.

The heuristic decision-making proposed in this study has proven to be a simple and effective approach for evaluating routing strategies. It is recommended that this method is used for further research on standardized scenarios to develop a comprehensive overview of routing strategies applicable to general pipelaying operations. Such an overview could also serve as a guideline for deck operators.

For practical implementation of a pipe routing system with the aim of full automation, more advanced methods decision-making methods should be considered. While the heuristic approach requires the developer to anticipate and encode all possible scenarios during the design phase, unexpected events will inevitably occur in practice. Reinforcement Learning offers a promising alternative, as it matches the sequential decision-making nature of the problem, can adapt to unforeseen situations, and may identify solutions beyond those recognized by heuristics. However, research on the practical application of Reinforcement Learning in such contexts is still limited, and substantial further investigation by Allseas would be required before it can be applied in real-world operations.

References

Angelle, J (2020). “Proof of value in automating the drill floor”. In: *Offshore Technology Conference Brasil 2019, OTCB 2019*. Offshore Technology Conference. ISBN: 9781613996713.

Borshchev, Andrei (June 2014). “Multi-method modelling: AnyLogic”. In: *Wiley Blackwell 6* 9781118349021, pp. 248–279. DOI: 10.1002/9781118762745.CH12.

Buchsbaum, Paulo (2012). “Modified PERT simulation”. In: *Great Solutions: Rio de Janeiro, Brazil*.

Charlton, Aidan et al. (2020). “Applying an Industry 4.0 Philosophy to Pipeline Integrity: The Future Beyond Digitalisation”. In: *Proceedings of the Pipeline Technology Conference (PTC)*. Penspen. Berlin, Germany: Pipeline Technology Conference.

Craig, C et al. (2023). “Fully Automated Land Rig Pipe Handling: Learnings from the First Year in Operation”. In: *Society of Petroleum Engineers - ADIPEC, ADIP 2023*. Society of Petroleum Engineers. ISBN: 9781959025078. DOI: 10.2118/216318-MS.

Dagkakis, G. et al. (2016). “A review of open source discrete event simulation software for operations research”. In: *Journal of Simulation* 10 (3), pp. 193–206. ISSN: 17477786. DOI: 10.1057/JOS.2015.9.

Eber, Julian et al. (June 2023). “Guided Reinforcement Learning: A Review and Evaluation for Efficient and Effective Real-World Robotics [Survey]”. In: *IEEE Robotics and Automation Magazine* 30 (2), pp. 67–85. ISSN: 1558223X. DOI: 10.1109/MRA.2022.3207664.

Filom, Siyavash et al. (May 2022). “Applications of machine learning methods in port operations – A systematic literature review”. In: *Transportation Research Part E: Logistics and Transportation Review* 161, p. 102722. ISSN: 1366-5545. DOI: 10.1016/J.TRE.2022.102722.

Gray, Michael A. (Nov. 2007). “Discrete event simulation: A review of simevents”. In: *Computing in Science and Engineering* 9 (6), pp. 62–66. ISSN: 15219615. DOI: 10.1109/MCSE.2007.112.

Ham, Ruud van der (2020). *Simulation of Logistic Systems in Python with Salabim*. https://pyvideo.org/europython-2020/simulation-of-logistic-systems-in-python-with-salabim.html. EuroPython 2020, Online; accessed 2025-04-23.

Hillier, Frederick S et al. (2015). *Introduction to operations research*. McGraw-Hill.

Kanervisto, Anssi et al. (Apr. 2020). “Action Space Shaping in Deep Reinforcement Learning”. In: *IEEE Conference on Computatonal Intelligence and Games, CIG* 2020-August, pp. 479–486. ISSN: 23254289. DOI: 10.1109/CoG47356.2020.9231687. URL: https://arxiv.org/pdf/2004.00980.

King, Harry (2024). *Comment on a LinkedIn Post by Paul Corry*. Accessed: 2025-03-12. URL: https://www.linkedin.com/feed/update/urn:li:activity:7108291487481659392?commentUrn=urn%3Ali%3Acomment%3A%28activity%3A7108291487481659392%2C7108958242227032064%29&dashCommentUrn=urn%3Ali%3Afsd_comment%3A%287108958242227032064%2Curn%3Ali%3Aactivity%3A7108291487481659392%29.

Kissell, Robert et al. (Jan. 2017). “Advanced Math and Statistics”. In: *Optimal Sports Math, Statistics, and Fantasy*, pp. 103–135. DOI: 10.1016/B978-0-12-805163-4.00004-9.

Landry, Maurice et al. (1983). “Model validation in operations research”. In: *European Journal of Operational Research* 14.3, pp. 207–220. ISSN: 0377-2217. DOI: https://doi.org/10.1016/0377-2217(83)90257-6.

Lang, Sebastian et al. (Jan. 2021). “Open-source discrete-event simulation software for applications in production and logistics: An alternative to commercial tools?” In: *Procedia Computer Science* 180, pp. 978–987. ISSN: 1877-0509. DOI: 10.1016/J.PROCS.2021.01.349.

Maidstone, Robert (2012). “Discrete event simulation, system dynamics and agent based simulation: Discussion and comparison”. In: *System* 1.6, pp. 1–6.

Miller, Tim (2023). *Markov decision processes*. URL: https://gibberblot.github.io/rl-notes/single-agent/MDPs.html.

Monacchi, Giulio et al. (2023). “An Innovative Set of Tools for a Sustainable and Safe Offshore Pipelaying”. In: *Proceedings of the Annual Offshore Technology Conference* 2023-May. ISSN: 01603663. DOI: 10.4043/32448-MS.

Moralez, N et al. (2020). “Intelligent pipe-handling: A case study for automation”. In: *SPE/IADC Drilling Conference, Proceedings*. Vol. 2020-March. Society of Petroleum Engineers (SPE). ISBN: 9781613996874.

Otterlo, Martijn et al. (Jan. 2012). “Reinforcement Learning and Markov Decision Processes”. In: *Reinforcement Learning: State of the Art*, pp. 3–42. DOI: 10.1007/978-3-642-27645-3_1.

Peleg, Micha (2019). “Beta distributions for particle size having a finite range and predetermined mode, mean or median”. In: *Powder Technology* 356, pp. 790–794. ISSN: 0032-5910. DOI: https://doi.org/10.1016/j.powtec.2019.09.015.

RiskAMP (n.d.). *Beta-PERT Distribution*. URL: https://www.riskamp.com/beta-pert/.

Robinson, S. (2004). *Simulation: The Practice of Model Development and Use*. John Wiley & Sons. ISBN: 9780470847725.

Siebers, P. O. et al. (2010). “Discrete-event simulation is dead, long live agent-based simulation!” In: *Journal of Simulation* 4 (3), pp. 204–210. ISSN: 17477786. DOI: 10.1057/JOS.2010.14.

Stern, Roni (2019). “Multi-agent path finding–an overview”. In: *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*, pp. 96–115.

TU Delft (2020a). *5.4 Discrete-event Modelling & Simulation*. https://ocw.tudelft.nl/course-lectures/5-4-discrete-event-modelling-simulation/. Lecture 5.4 from the course NGI: Modelling Complexity.

— (2020b). *5.5 System Dynamics Modelling & Simulation*. https://ocw.tudelft.nl/course-lectures/5-5-system-dynamics-modelling-simulation/. Lecture 5.5 from the course NGI: Modelling Complexity.

Van Der Ham, Ruud (2018). “salabim: discrete event simulation and animation in Python”. In: *Journal of Open Source Software* 3.27, p. 767.

Veeke, Hans PM et al. (2008). *The Delft systems approach: Analysis and design of industrial systems*. Springer Science & Business Media.

Vieira, António Amaro Costa et al. (2020). “A ranking of the most known freeware and open source discrete-event simulation tools”. In: *31st European Modeling and Simulation Symposium, EMSS 2019*, pp. 103–110. DOI: 10.46354/i3m.2019.emss.017.

Wang, Huanhuan et al. (2013). “A framework for integrating discrete event simulation with agent-based modeling”. In: *Proceedings of 2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering, ICIII 2013* 3, pp. 176–180. DOI: 10.1109/ICIII.2013.6703542.

Wasserman, Larry (2004). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.

Wilson, Alec et al. (Sept. 2024). “Applying Action Masking and Curriculum Learning Techniques to Improve Data Efficiency and Overall Performance in Operational Technology Cyber Security using Reinforcement Learning”. In: *CAMLIS*.

Wuest, Thorsten et al. (2016). “Machine learning in manufacturing: advantages, challenges, and applications”. In: *Production & Manufacturing Research* 4.1, pp. 23–45.

A

Research Paper

Conceptual design of a pipe routing system on a pipelaying vessel

L. Sijm, M.B. Duinkerken, and X.L. Jiang

Department of Maritime and Transport Technology, Delft University of Technology
Delft, The Netherlands

Abstract

The adoption of new technologies on pipelaying vessels is closely linked to the demand for full automation of pipe handling operations. However, pipe handling systems are rarely addressed in research, and, to the best of our knowledge, no studies exist that describe comparable intralogistics systems with similar control structures and movement restrictions. As a result, there is no clear reference for how such automation can be realized. This study addresses this gap by developing a conceptual pipe routing system for the pipelaying vessel *Solitaire*, operated by Allseas, and implements it within a Discrete Event Simulation model of the pipe handling system. The model is used to evaluate multiple routing strategies under different pipe supply scenarios and to determine the most effective routing strategy for the case study. The results show that a heuristic approach to sequential decision-making, framed within a model-free Markov Decision Process, can automate pipelaying operations and lay the groundwork for future optimization through reinforcement learning.

Keywords: Pipelaying vessel; Pipe handling; Automation; Heuristics; Sequential decision-making; Discrete Event Simulation; Salabim.

1. Introduction

With the rise of Industry 4.0, automation has become widely implemented across many industrial sectors, fundamentally transforming their operations. The pipelaying industry, however, remains relatively conservative (Charlton & Curson, 2020). A similar trend is observed in the drilling sector of the oil and gas industry which involves comparable pipe handling operations. These industries have not followed other batch and process industries such as the automotive industry in terms of adopting new technologies (Angelle, 2020). An explanation for this could be that, although pipe handling involves repetitive tasks, the exact operations depend on the specific processes and are carried out in challenging environments (Moralez et al., 2020). Nevertheless, recent studies (Angelle, 2020, Moralez et al., 2020, Craig et al., 2023) show progress toward more advanced automated pipe handling systems aimed at reducing human intervention. This trend is especially relevant because the safety risks for personnel involved in pipe handling operations remain significant (Monacchi et al., 2023)

1.1. Case Study at Allseas

Also Allseas, a world-leading contractor in the offshore energy market, aims to make a step towards automation of pipe handling operations as the pipe handling equipment on their pipelay vessel *Solitaire* is due for renovation. The new configuration of the equipment will increase the complexity of the pipe flow on the vessel, which is currently controlled by operator decision-making. This raises the need for a pipe routing system that automates the decision-making process to autonomously regulate the pipe flow on board, which is defined by this study as the pipe routing problem.

However, Allseas is uncertain how to develop a routing strategy for this pipe routing system, as no automation studies exist on similar pipe handling systems, nor are there studies in literature describing comparable intralogistics systems with similar control structures and movement restrictions.

This study addresses this research gap by developing a conceptual pipe routing system to regulate pipe movement on a pipelaying vessel and evaluating its performance for different routing strategies using a Discrete Event Simulation model of the pipe handling system on *Solitaire*. The study aims to demonstrate how such a system can be automated and to identify opportunities for future optimization.

1.2. Report structure

The structure of this research paper is as follows. Section 2 introduces the pipe handling system of the case study. Section 3 outlines the methodology employed by the pipe routing system to model and address the pipe routing problem. Section 4 details the development of the DES model used to implement and evaluate the conceptual pipe routing system. Finally, Section 5 presents the experimental results, which form the basis for identifying the best performing routing strategy for the case study.

2. System analysis

2.1. Pipelaying operations on Solitaire

Solitaire is a pipelaying vessel with two factories: the double joint factory (DJF) and the firing line. In the DJF, single lengths of steel pipe (joints) are welded together to form double joints, which are then supplied to the firing line. In the firing line, the double joints are welded together to form the main pipeline. At the end of the firing line, the main pipeline is lowered into the water until it reaches the seabed.

Appendix A Figure A.1 presents the pipelaying operations in an IDEF0 diagram, Figure 2.1 is a copy of

the A0 level of this diagram, showing the pipelaying operations can be divided in five subsystems: A1: pipe delivery, A2: pipe yard operations, A3: DJF operations, A4: firing line operations and A5: reprocessing operations. Joint arrive on Pipe Supply Vessels (PSV) and are delivered by deck-mounted pipe transfer cranes (PTC). After delivery the joints pass several processing stations in the pipe yard before being transported to the DJF. In the DJF the joints pass several processing and welding stations before they are transported as double joints to the firing line. All welds in both the DJF and firing line are tested and in case of a weld reject the double joint is send back towards the pipe yard through the reprocessing operations.

The pipe routing system will regulate the joint transport within the subsystems A2, A3 and A5, which are collectively referred to as the pipe handling system.

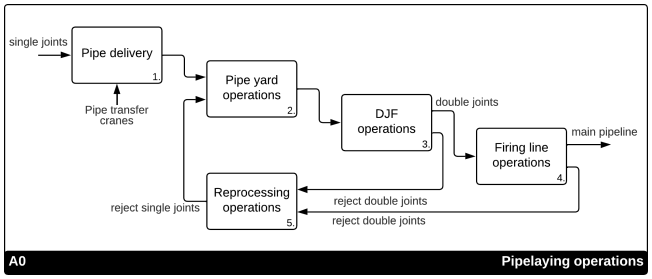


Figure 2.1: IDEF0 diagram Function A0

2.2. Pipe handling system

Figure 2.2 shows a black-box model of the pipe handling system. Joints are delivered by a PTC, processed by the pipe handling system and then delivered as double joints through the pipe elevator to the firing line. To regulate the operations within the pipe handling system, the system requires a pipe routing system. The main goal of the pipe handling system is to ensure that the firing line is continuously fed with the required double joints, therefore the key performance indicator (KPI) is the firing line wait time. Another goal for the system is to minimize the PSV turnaround time, therefore the PTC wait time is also an KPI.

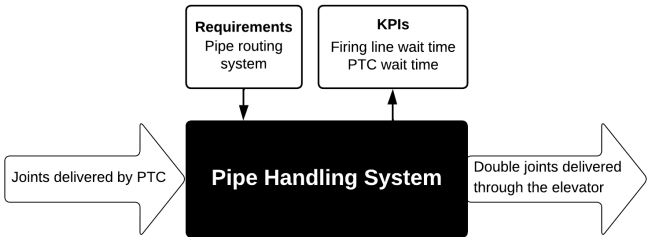


Figure 2.2: Black box model for pipe handling system

As earlier mentioned the pipe yard operations and DJF operations consist of several processing stations that the joints need to pass by means of the transport equipment. There are four types of transport equipment:

- FPS: fixed pipe support for parking joints
- LCO: longitudinal conveyors for parking and transport in longitudinal direction
- TC: transverse cars for transport in transverse direction
- OC: overhead crane for transport in transverse direction

Figure 2.3 shows the general structure of the pipe handling system and its transport equipment with the FPSs and LCOs illustrated in beige to indicate positions where a joint can be parked. The black arrows beneath these positions illustrate TCs that enable transverse transport between the LCOs and FPSs. The black arrows above the positions illustrate OCs. The systems is divided in four different areas: the pipe yard, the DJF, the waiting area and the return line. The waiting area is the area at the end of the DJF where double joints can be stored in a buffer or supplied to the firing line through the elevator. The return line enables reprocessing of rejected double joints, but can also be used for temporarily storage of joints in the holds if they cannot be directly transported to the DJF.

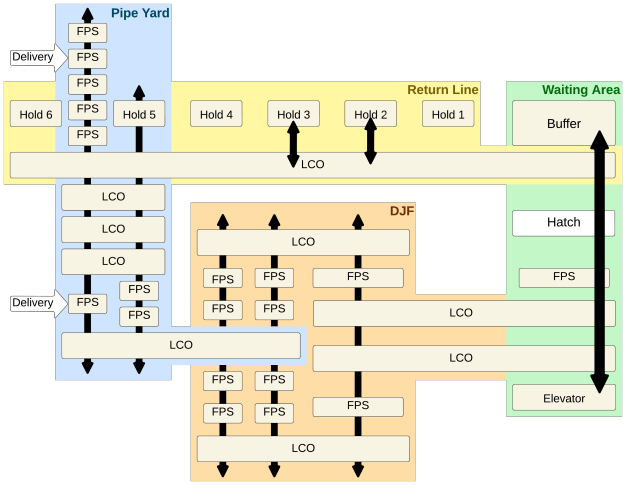


Figure 2.3: Structure of the pipe handling system

2.3. Pipe routing system

The pipe routing system would be implemented in the control structure of the pipe handling system as an intermediate system that is connected to the database containing all pipe and station information. The pipe routing system does not control the equipment itself, but instead only translates the pipe and station information into pipe movement requests for the existing pipe handling PLC controlling the equipment.

3. Methodology

Since no relevant research on the automation of pipe handling systems is available, inspiration must be drawn from operations research methods applied in comparable intralogistics systems. A key distinction from systems like automated container terminals or automated guided vehicles, however, is that in the pipe handling system the movable objects (joints) are identical. Consequently, the routing problem is not constrained by predefined schedules or fixed destinations, making it less suitable for common optimization methods. This makes

that there were no example automation or optimization studies found that could be directly compared to the pipe handling system.

As no direct examples are available for reference, this study first focuses on automating the pipe handling system to gain a deeper understanding of its operation and establish a performance benchmark for identifying optimization potential. While the optimization phase lies beyond the scope of this research, optimization approaches from comparable intralogistics systems were analysed to establish possible methods to model the pipe handling system and solve the pipe routing problem. The following subsections outline the methodology used to model the system and to address the pipe routing problem in the development of an automated solution.

3.1. Modelling the pipe handling system

The pipe handling system is modelled using simulation, as this approach is particularly suited for systems with stochastic behaviour that are too complex to be analysed using purely mathematical methods (Hillier & Lieberman, 2015). Discrete Event Simulation (DES) is identified as the most appropriate method to capture the interactions of multiple components controlled by a centralized control agent (the pipe routing system). A self-conducted comparative study of DES software identified Salabim as the preferred tool. As an open-source Python-based simulation library, it offers full customizability and provides all essential features, including a basic integrated animation function (Van Der Ham, 2018).

3.2. Modelling and solving the pipe routing problem

Among the identified optimization methods, Reinforcement Learning (RL) is concluded to be the most interesting method for optimization as it shares the most key principles with the pipe routing problem. Moreover, the method is able to adapt to unexpected scenarios (Filom et al., 2022) and it may uncover unrecognized relationships within the system (Wuest et al., 2016). Also Moralez et al., 2020 concludes Artificial Intelligence (AI) will play an important role in the continuous improvement of automated pipe handling systems. As the pipe routing problem inherently incorporates many key principles relevant to RL, the system model is designed to support future expansion in this direction.

The pipe routing problem is modelled as a sequential decision making problem following a model-free Markov Decision Process (MDP) framework to allow for future implementation of a RL model to solve the problem. In MDP a decision is only based on the current state of the system, without reference to past information (Hillier & Lieberman, 2015). This characteristic is well suited to the pipe routing problem, as the stochastic system input and process durations make it difficult to schedule multiple decisions in advance. A classic MDP framework formulates the problem as a tuple (S, A, P, R) containing the following elements:

- state space S is the set of all possible states the system can be in,
- actions A allow the decision-maker to transition the current state in another state,
- transition probabilities P define the new state s' after action a given the current state s,
- rewards R specify the benefit or cost of executing action a given the current state s.

In model-free MDP the transition probabilities P and rewards R are replaced by an external model, such as a DES model, which executes an action a in simulation to derive the resulting state s'. Otterlo and Wiering, 2012 distinguishes three classes of algorithms (policies) for solving sequential decision problems (choose actions): programming, search and planning, and learning algorithms. Although the programming approach requires the programmer to anticipate and encode all possible states during the design phase, this method is adopted in this study, as the policy can be derived from the operational experience of the existing pipe handling system, which is currently managed through human decision-making.

This results in the decision framework illustrated in Figure 3.1. The DES model provides the system state s, comprising the status of joints, the occupancy of positions and the status of processing stations and transport equipment. The pipe routing system is the decision-maker that selects action a from its policy, given state s. Action a represents the next joint movement to be executed by the pipe handling system in the DES model. Once the action is performed, the DES model transitions to a new state s', which is then returned to the pipe routing system. The policy will from now on be referred to as the routing strategy of the pipe routing system.

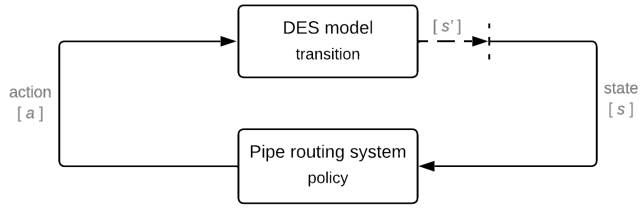


Figure 3.1: Framework for the pipe routing problem

4. Simulation model

As outlined in the methodology, the pipe handling system is modelled in Python using Salabim as a DES framework. In this language-based modelling approach, the model is constructed as a set of components with attributes and process cycles. These components interact with the environment and with each other's process cycles through functions such as *hold*, *wait*, *passivate*, and *activate*. This section presents the formulation of the simulation model and the routing strategy, followed by verification and validation. Emphasis is placed on the centralized control structure within the model rather than the detailed system content.

4.1. Pipe handling system

The elements and structure of the pipe handling as described in Section 2 and illustrated in Figure 2.3, are

represented in the model with a number of simplifications and assumptions. The most relevant of these are summarized below:

- LCOs and FPSs are represented as positional references rather than components.
- LCOs are divided into fixed positions; joints cannot remain between positions.
- Movement paths in the DJF and waiting area are fixed, limiting the scope to only evaluate routing strategies in the pipe yard.
- The continuous supply to the firing line is represented as a continuous supply to the waiting area to exclude the influence of the routing strategy within the waiting area. Consequently, the KPI is defined as the waiting area wait time.

4.2. Pipe routing system

The pipe routing system is implemented in the simulation as a component of the overall pipe handling system. It observes the system state, defined by the status of all joints within the system and the occupancy of positions. Based on this state, it applies a user-defined routing strategy to assign the next action, specifying which joint to move, its destination, and the resource required. After assigning the action, the pipe routing system observes the updated state and repeats this process until no further actions are available. At that point, the pipe routing system passivates and is reactivated when the system state is updated by another component, such as a PTC or processing station.

4.3. Model data

The model data used for calculating movement and process durations is stored within the model environment and component attributes. Positional coordinates are scaled to replicate the actual layout of the pipe handling system, enabling movement durations to be derived from equipment specifications rather than predefined values. Component attributes are based on actual equipment data, but the process durations depend on pipeline specifications.

For this study, an example project involving a 36-inch diameter pipeline is selected. The PTC process duration is estimated by a self-defined triangular distribution, based on a given average delivery rate. The process durations in the DJF are estimated by self-defined beta distributions based on a performed data analysis and given average process durations. The process duration for the firing line is based on empirical data from the example project. The combination of all these distributions imitate the stochastic behaviour of the pipe handling system.

4.4. Model input and simulation output

The model input consists of simulation parameters, map specifications, and the routing strategy. The routing strategy is based on a user-defined move map, which defines the destination and required resources for each joint depending on its current position. Additional routing strategy details can be incorporated into the process cycle of the pipe routing system component.

The simulation output includes detailed data reports and an animation of the simulation events. The reports provide insights into pipe flow, station utilization, and production quantities. Combined with the live animation or simulation snapshots, such as Figure 4.1, the output offers a deeper understanding of the system and the effects of the routing strategy.

4.5. Verification and validation

Both the conceptual model and the computer model were verified and validated following the methodology of Robinson, 2004. The simulation animation proved particularly valuable for verification and white-box validation. The conclusive black-box validation demonstrated that the simulation output was comparable to the actual layrate data from the example project, with only a 3% deviation. In addition, the DJF production in the simulation differed only 3.5% from a prior, more simplified, simulation study of the pipe handling system using similar input parameters.

5. Experimentation with the simulation model

Using the described simulation model, experiments are conducted to evaluate and improve the performance of various routing strategies for the pipe handling system on *Solitaire*. Rather than aiming for a fixed performance target, the objective is to gain insight into how different strategies affect system performance under varying pipe supply scenarios and to identify the most effective routing strategy. The following experimental plan is followed to achieve this:

1. Establish relevant pipe supply scenarios and possible route configurations
2. Establish simulation run setup

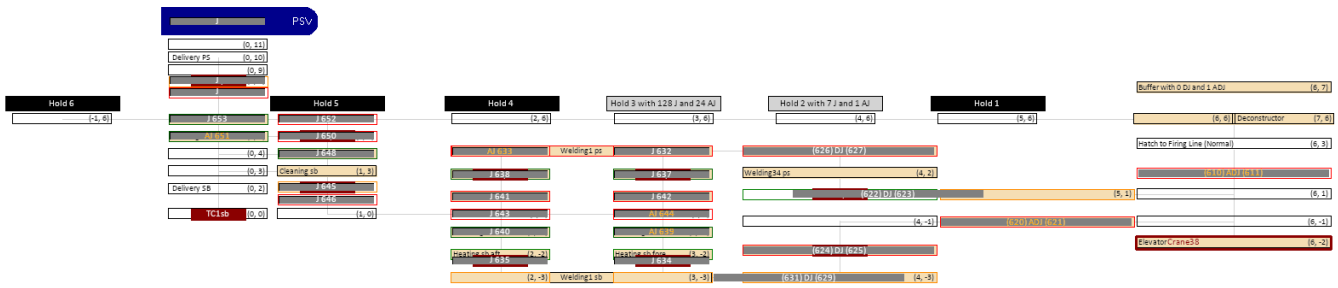


Figure 4.1: Snapshot from the simulation animation in Salabim

3. Evaluate baseline performance for all route configurations under all pipe supply scenarios

4. Improve each route configuration through interactive search experimentation

5. Evaluate performance for all improved route configurations under all pipe supply scenarios

5.1. Experimental scenarios and configurations

The scope of the experiments is limited to pipe supply from port-side to allow for a more detailed analysis. Three average PTC delivery rates are considered: a slow rate of 18 single joints per hour, which just meets DJF demand; a realistic rate of 24 joints per hour; and a maximum rate of 30 joints per hour.

To reduce the solution space of possible configurations, the routing problem is divided into subproblems, where joints move from station to station. Their navigable space is restricted to the area between their current and next station, and movement against the general pipe flow is not allowed. Under these constraints, six base route configurations are formulated, illustrated in Appendix B Figure B.1.

The three pipe supply scenarios combined with the six route configurations result in a total of eighteen comparable experiments.

5.2. Simulation run setup

The experiments are analysed individually using non-terminating simulations, with results expressed as mean daily outputs during continuous pipelaying operations. The DJF production and main pipeline length parameters are evaluated to validate that the model generates a realistic production output. The waiting area wait time and the PTC wait time are the key performance indicators targeted for minimization. Additionally, also the DJF wait time is considered as a valuable output parameter to minimize.

A warm-up period of 4 hours and a run length of 21 days are required to obtain accurate and comparable results, although the level of convergence differs between parameters and should be considered when interpreting deviations.

5.3. Base performance

The simulation results in Table 5.1 present the baseline performance, which serves as a reference and to identify opportunities for improving the routing strategy. Since the base route configurations from Appendix B yield similar performance results, all configurations will be further developed through interactive search experimentation.

5.4. Interactive search experimentation

Based on the base performance analysis, four directions for improvement are determined to explore: the hold policy, the anode distribution in the DJF, the infeed line of the DJF and the TC moves. Each direction is seen as an update that can be applied to the route configurations.

5.4.1. Hold policy

The hold policy determines when normal and/or anode joints are sent to the hold to regulate the anode ratio in the DJF and to solve congestion in the pipe yard. The aim was to improve this policy to prevent deadlocks in the waiting area caused by an unbalanced anode ratio and to reduce PTC waiting time. The enhanced policy successfully eliminated deadlocks and removed PTC waiting time for all routes under a realistic PTC rate, while also significantly reducing it under a fast rate. For this reason, the enhanced hold policy is applied to all route configurations.

5.4.2. DJF anode distribution

Waiting area wait time occurs because normal and anode joints do not always arrive in perfect sequence. To address this, three distribution strategies were tested to balance anode joints between the two welding lines leading to the waiting area. However, none of these strategies had a significant effect, and therefore they are not applied to the route configurations.

5.4.3. DJF infeed line

Since operations in the pipe yard are generally faster than those at the DJF processing stations, it appeared logical to use the pipe yard to pre-align a pair of single joints that can be transported simultaneously to the DJF supply positions as soon as they become available. The objective of this strategy was to reduce DJF waiting time and to prevent the combination of two anode single joints. The results confirmed a reduction in DJF waiting time, except for route configuration (d), where the prevention of two anode single joints caused deadlocks in the pipe yard. Therefore, the DJF infeed line strategy is applied to all configurations except route (d).

5.4.4. TC support moves

TC support moves are introduced to reposition idle TCs when no tasks are available, with the aim of reducing PTC and DJF waiting times by anticipating their next required position. The results showed a clear improvement in PTC waiting time but no significant effect on DJF waiting time. Animation analysis confirmed that TC movement speed was not a limiting factor, although the TCs did correctly anticipate their next moves. Therefore, TC support moves are applied to all route configurations.

Table 5.1: Reference simulation results for baseline performance

Output parameter	**Unit**	**Slow PTC rate (18 SJ/h)**	**Avg PTC rate (24 SJ/h)**	**Fast PTC rate (30 SJ/h)**
DJF production	joints / day	207	207	207
Main pipeline length	m / day	4886	4889	4880
PTC wait time (KPI)	min / day	0	69	291
DJF wait time	min / day	53	46	56
Waiting area wait time (KPI)	min / day	11	11	12

Table 5.2: Best obtained simulation results with improved route configuration (f)

		Slow PTC rate (18 SJ/h)		Avg PTC rate (24 SJ/h)		Fast PTC rate (30 SJ/h)	
Output parameter	**Unit**	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]	**Result**	**Dev.**[1]
DJF production	joints / day	207	*0%*	207	*0%*	208	*0%*
Main pipeline length	m / day	4889	*0%*	4890	*0%*	4893	*0%*
PTC wait time (KPI)	min / day	0	*0%*	0	*-100%*	0	*-100%*
DJF wait time	min / day	40	*-25%*	34	*-26%*	33	*-41%*
Waiting area wait time (KPI)	min / day	9	*-18%*	8	*-27%*	8	*-33%*

[1] Deviation from the DJF supply line configuration results for route (a) presented in Table 5.1

5.5. Conclusive batch experimentation

After applying all routing strategy updates from the interactive search experimentation, the route configurations were evaluated once more trough a comparative batch of experiments. The strategy updates significantly improved performance for all route configurations, however, the final batch experiments identified route (f) as the most effective strategy with the results presented in Table 5.2, compared against the baseline performance. With this configuration the PTC wait time is fully eliminated, whereas other routes retained approximately 15 min/day under fast PTC rates. This improvement results from route (f) allowing joints to move towards the hold without moving against the pipe flow. Reductions in waiting area waiting time were minor and comparable across all configurations.

Overall, all route configurations achieved satisfactory system performance, with little room for further improvement given the KPIs defined in this study. Only route configuration (d) is rejected as a solution due to reoccurring deadlocks.

6. Conclusion

Publications on the automation of pipelaying operations are very limited. This study addressed this research gap by developing a conceptual pipe routing system to regulate joint movements within a pipe handling system on vessels, using Allseas' pipelaying vessel *Solitaire* as a case study. The pipe routing problem was identified as a sequential decision-making problem, making it well suited for optimization through Reinforcement Learning. However, the research should first focus on achieving automation and providing a better understanding of the system to identify the potential for optimization. With this in mind, a heuristic approach within a model-free Markov Decision Process framework was applied to automate decision-making.

The pipe routing system was implemented in a Discrete Event Simulation model of the pipe handling system to evaluate its performance under different routing strategies and determine a most effective strategy for the case study. The experiments demonstrated that simulation provides valuable insights into system behaviour and the impact of routing strategies, ultimately achieving complete elimination of the pipe transfer crane waiting time, a key performance indicator for this study.

This research demonstrated that simulation provides a deeper understanding of the pipe handling system and the impact of routing strategies. The heuristic approach successfully automated sequential decision-making and regulated joint movements. Nevertheless, the main challenge that remains is full scenario coverage, given the wide variation in pipeline specifications, environmental influences, and unpredictable events.

A future pipe routing system based on RL could potentially overcome this by adapting to a wide range of scenarios and identifying solutions beyond those considered in the heuristic approach. Since this research has already proposed formulating the pipe routing problem within a model-free MDP framework, an RL model could be directly integrated into the existing setup, allowing for a straightforward performance comparison with the heuristic approach. It would be interesting to see if such a direct comparison would advance the development of an RL model, as this approach could then also support the practical application of RL in other industries.

References

Angelle, J. (2020). Proof of value in automating the drill floor. *Offshore Technology Conference Brasil 2019, OTCB 2019*.

Charlton, A., & Curson, N. (2020). Applying an industry 4.0 philosophy to pipeline integrity: The future beyond digitalisation. *Proceedings of the Pipeline Technology Conference (PTC)*.

Craig, C., Gupta, A., Yenzer, D., Hasler, D., & Towns, J. (2023). Fully automated land rig pipe handling: Learnings from the first year in operation. *Society of Petroleum Engineers - ADIPEC, ADIP 2023*. https://doi.org/10.2118/216318-MS

Filom, S., Amiri, A. M., & Razavi, S. (2022). Applications of machine learning methods in port operations – a systematic literature review. *Transportation Research Part E: Logistics and Transportation Review*, *161*, 102722. https://doi.org/10.1016/J.TRE.2022.102722

Hillier, F. S., & Lieberman, G. J. (2015). *Introduction to operations research*. McGraw-Hill.

Monacchi, G., Arcangeletti, G., Pigliapoco, M., Catena, P., Ziero, L., Castriotta, G., & Loi, A. (2023). An innovative set of tools for a sustainable and safe offshore pipelaying. *Proceedings of the Annual Offshore Technology Conference*, *2023-May*. https://doi.org/10.4043/32448-MS

Moralez, N., Ronquillo, N., Lisi, D., Lynch, M., Parker, C., Sutler, J., & Machum, M. (2020). Intelligent pipe-handling: A case study for automation. *SPE/IADC Drilling Conference, Proceedings*, *2020-March*.

Otterlo, M., & Wiering, M. (2012). Reinforcement learning and markov decision processes. *Reinforcement Learning: State of the Art*, 3–42. https://doi.org/10.1007/978-3-642-27645-3_1

Robinson, S. (2004). *Simulation: The practice of model development and use*. John Wiley & Sons.

Van Der Ham, R. (2018). Salabim: Discrete event simulation and animation in python. *Journal of Open Source Software*, *3*(27), 767.

Wuest, T., Weimer, D., Irgens, C., & Thoben, K.-D. (2016). Machine learning in manufacturing: Advantages, challenges, and applications. *Production & Manufacturing Research*, *4*(1), 23–45.

A. Pipe laying operations on *Solitaire*

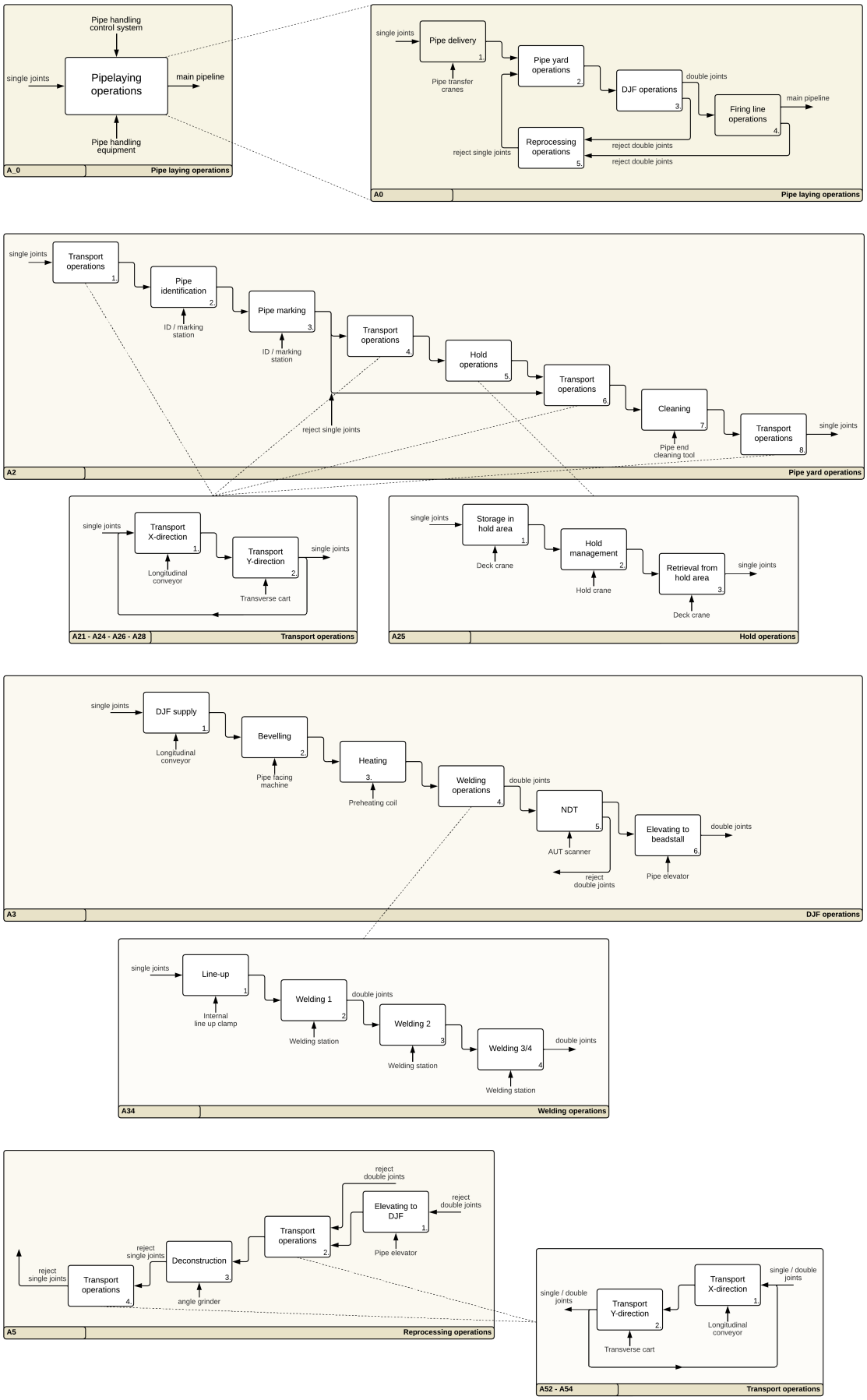
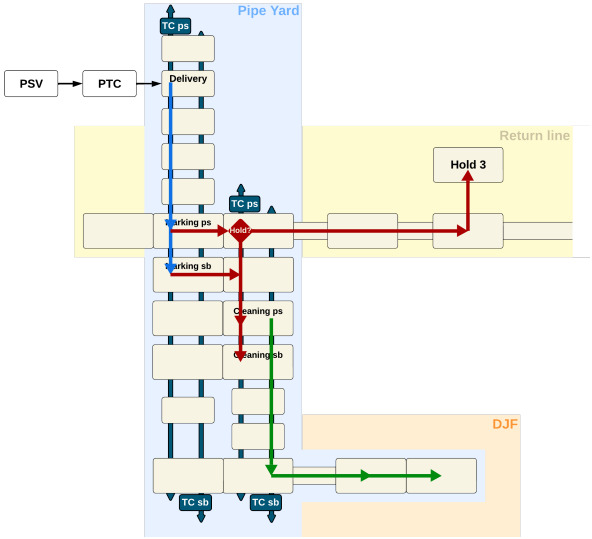
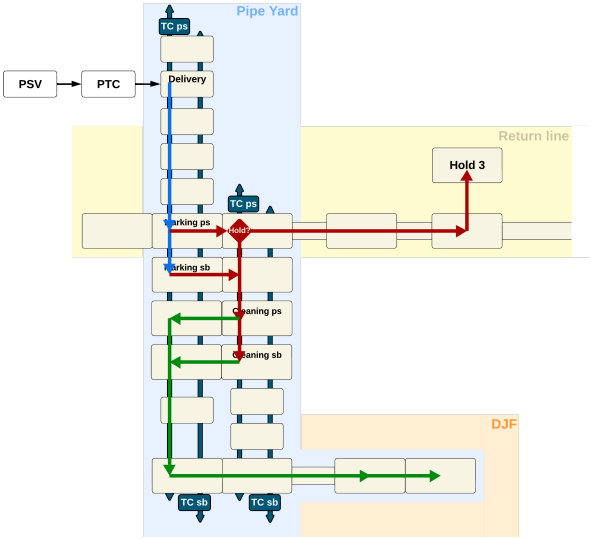


Figure A.1: IDEFO diagram for pipelaying operations on *Solitaire*

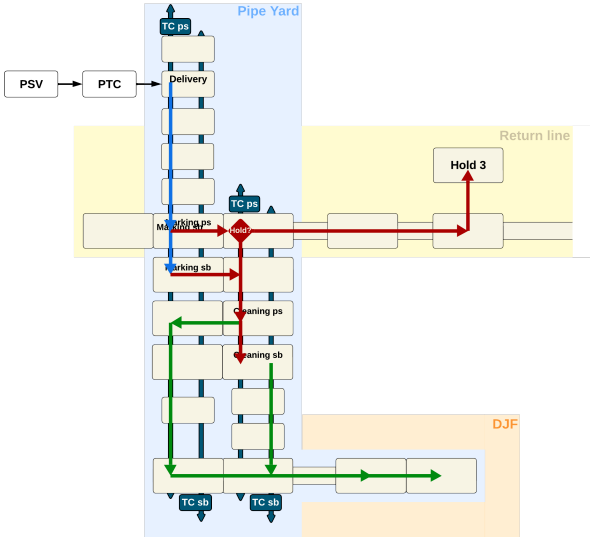
B. Base route configurations



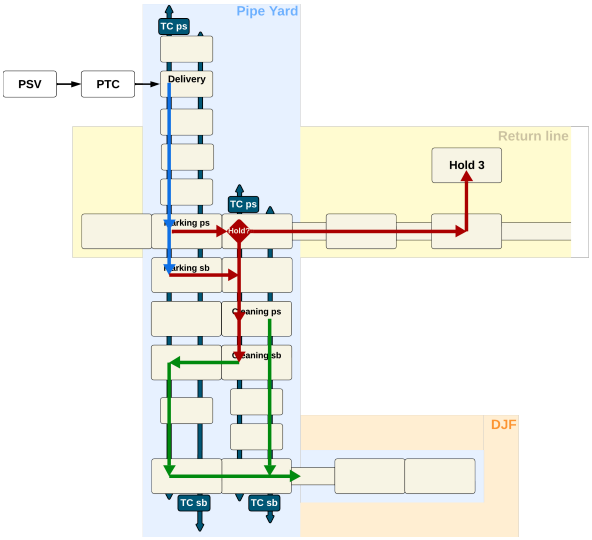
(a) Route PS (a)



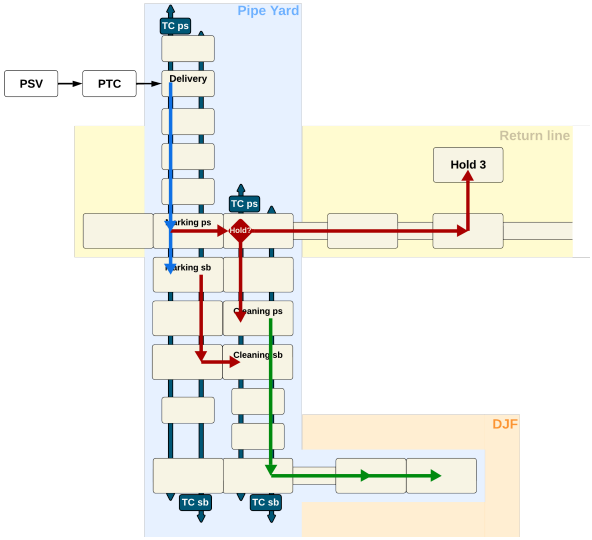
(b) Route PS (b)



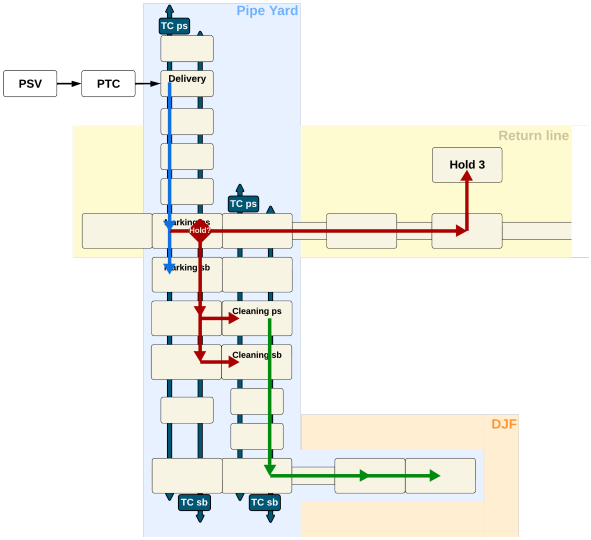
(c) Route PS (c)



(d) Route PS (d)



(e) Route PS (e)



(f) Route PS (f)

Figure B.1: Base route configurations for pipe supply from port-side

B

Selection of simulation software

Since the system to be modelled does not consist solely of straightforward, sequential processes, it is important to carefully select the DES software package for modelling the pipe yard system and its control structure. The most suitable simulation package for this study will be determined in this appendix following the process of software selection described by Robinson 2004. This book summarizes the process in the following 5 steps;

1. establish the modelling requirements,
2. survey and shortlist the software,
3. establish evaluation criteria,
4. evaluate the software in relation to the criteria,
5. software selection.

B.1. Modelling requirements

This section summarizes requirements and considerations for choosing a suitable DES software package. The software is required to model the pipe yard system as described in the main report Chapter 2. Although the model can potentially be useful for different research applications within Allseas, it will be designed for this study without restrictions for future applications. As this study is defined as a feasibility study resulting in a consult for further research, there are no requirements for application of the model in practice.

The developer possesses basic knowledge of model development and familiarity with Python as a programming language. Although an intuitive program with limited required programming knowledge is preferred for both the developer and the evaluation of the model by other personnel, modelling flexibility is also important considering the implementation of the centralized control agent controlling events within the model. On top of the implementation of the control structure, there is also a demand for optimizing the decision-making strategy within the control structure. But the actual method and requirements remain unclear and this stage, thus software packages offering a wide range of solutions are preferred on this aspect. Furthermore, animation is an important feature of the simulation software as one of the modelling objectives is to get a better understanding of decision-making strategies based on system state and the effects on pipe flow, which can best be evaluated by visualizing the joint movements.

Allseas is currently not willing to pay for commercial software packages, but packages that are available through trials or student licensing can be considered as the model will be designed for this study only. Open Source (OS) software can be strong alternative to commercial packages considering future use of the model.

B.2. Software shortlist

This section summarizes the shortlist of simulation software that have been considered for this project, including: SimPy, Salabim, JaamSim, AnyLogic, Tecnomatix Plant Simulation and MATLAB SimEvents. These packages are selected based on the developers educational program, experience within Allseas and findings in literature. All software packages are introduced through findings from comparative studies and personal user experience.

SimPy and Salabim

SimPy and Salabim are both OS DES frameworks for Python that are featured in the developers educational program, and Salabim has been used in an educational project by the developer before. Although SimPy is actually the most popular free DES software available online, Vieira et al. 2020, the developer of Salabim claims to have developed a successor with a more user friendly modelling paradigm. Also, Salabim provides queues, states, data monitors, statistical distributions and animation, none of which are present in SimPy Van Der Ham 2018. As both programs are language based without a graphical user interface (GUI) they seem mostly directed to developers. However, when implementing complicated decision logic into a model, language based programs can actually be more user-friendly than specialized GUI packages that are restricted to their intend of use (Ham 2020). The use of Python is attractive to developers that are not necessarily software experts as it is an intuitive language with many popular scientific packages available (Dagkakis et al. 2016). However, as a scripting language Python is by default slower than static languages like Java and C++ (Dagkakis et al. 2016). King 2024 showed an experiment on a single server queue system with 500.000 arrivals showing a ten times longer computation time for Python compared to Java.

JaamSim

Vieira et al. 2020 concludes that the use of OS software is still limited, looking at the amount of papers published by Winter Simulation Conference (world's largest simulation conference), and also Dagkakis et al. 2016 concludes that there is no OS DES tool that can actually compete with commercial software in OR applications, with a possible exception of JaamSim that had just recently been made OS. JaamSim is also one of the most popular free DES software available as it provides the most impressive GUI and it is the only OS tool that is clearly industry driven (Dagkakis et al. 2016). The drag and drop interface, combined with the possibility for developers to add new objects, make the program applicable for different levels of potential users. For this reason Allseas has used JaamSim in a prior to study to determine the most efficient layout of the pipe yard, however the study was limited to simplified joint

movements within the yard and the project got stuck when trying to implement more complex movement strategies. JaamSim is licensed under the General Public License (GPL) and all projects based on this software will have to use the same license, meaning you have to share the source code and include the original copyright notice when distributing your project.

AnyLogic

AnyLogic is a commercial simulation platform offering all three modelling methods (SD, DES and ABM) and even multi-method modelling. Borshchev 2014 claims while using a traditional single-method tool, the modeller inevitably either starts using workarounds, or just leaves part of the problem outside the scope of the model as the problem cannot be completely conformed to the modelling paradigm. This seems to comply with the pipe routing problem and its centralized control agent, which is why the software was tested through the free available personal learning license. As expected the software offered an intuitive GUI for quickly building a model using the useful material handling library. Although the program allows for numerous control features including external language based scripts, the developer concluded that the Java programming language and the more complicated paradigm of integration between external scripts does not fit the expertise of the developer. The program is still an interesting option for Allseas for future projects.

Tecnomatix Plant Simulation

Another shortlisted commercial package is Tecnomatix Plant Simulation by Siemens, a widely known simulation tool for analysing production and logistics processes in the automotive industry. As can be expected from a commercial program, it provides an intuitive GUI with the possibility to develop custom process logic and new users can quickly learn how to create models thanks to the extensive and detailed online help, the many example models and the large active user community (Lang et al. 2021). Allseas was interested in this software in the past, but took it out of consideration as they could not receive a trial license. The developer could be eligible for a student license through the TU Delft, but after the experiences with JaamSim and AnyLogic it seems unnecessary to try another GUI simulation package offering similar features.

MATLAB SimEvents

MATLAB SimEvents is not mentioned in comparative studies like the other tools as MATLAB is not purely a DES program. SimEvents is actually an add-on library within the Simulink environment, which is both an advantage and a disadvantage. SimEvents is missing some direct mechanisms for important features, which can be compensated by the resources within Simulink to build your own mechanisms (Gray 2007). Although the MATLAB language has advantages for the control structure as it is familiar to the developer and it offers elaborate add-ons like optimization libraries, the actual DES tools are concluded to be too limiting to construct the model of the pipe yard.

B.3. Evaluation criteria

In this section the shortlisted software will be evaluated based on: ease of model development, integrated animation, optimization solutions for decision-making, modelling support and potential for future use of the model within Allseas. The ease of model development is divided into the model content and the control structure in the criteria. The first only focuses on modelling all the elements and their relationships like movement restrictions for joints requiring a TC. The latter focusses on implementing the control structure that assigns movements to joints in the desired direction. Modelling support refers to software documentation, available experience from supervisors and also AI-powered assistance as this can highly influence the ease of model development.

B.4. Software evaluation

All criteria receive a weight between 1 and 5 indicating their relative importance. The software receive a score between 1 and 5 for each criteria, based on the findings from comparative studies and personal user experience summarized in the previous section. It should be noted that the scoring is subjective and shaped by how closely the programs relate to the developer's area of expertise. The overall evaluation is summarized in Table B.1.

Table B.1: Discrete event simulation software comparison

Criteria	***w***	**SimPy**	**Salabim**	**JaamSim**	**AnyLogic**	**Plant Sim.**	**SimEvents**
Language		Python	Python	Java	Java	SimTalk	MATLAB
Licensing		OS	OS	OS	Commercial	Commercial	Commercial
Modelling approach		Language	Language	GUI	Hybrid	Hybrid	Hybrid
Model content	*4*	2	3	4	5	5	1
Control structure	*5*	4	5	1	2	2	3
Animation	*3*	1	3	3	5	5	1
Optimization	*3*	4	4	1	3	3	4
Support	*2*	4	4	2	3	3	4
Future use	*1*	4	4	3	1	1	3
Total score:		55	70	40	61	61	45

B.5. Modelling software choice

Based on Table B.1 it can be concluded that Salabim is the most suitable DES software package and will be used for this project. During the testing and evaluation of the shortlisted software options it became evident that the implementation of the desired control structure within a DES environment is unconventional. In specialized GUI packages, DES models are typically thought of as networks of queues and servers with entities moving through (Maidstone 2012), resulting in limited solutions to control the entity flow. As such, a language-based modelling approach offers greater flexibility than a GUI with pre-built mechanisms. While commercial software may offer similar flexibility through custom extensions, these implementations seem more complex and do not align with the developer's current expertise and available support.

Salabim, as a language-based simulation library in Python, provides all the essential features, including a basic integrated animation function. Since it is purely code-driven, modelling the system's content may require more effort. However, with available guidance from supervisors and support from AI-powered tool, this challenge can be effectively managed. Additionally, building the model from scratch allows for greater flexibility and will simplify the implementation of the control structure.

C

Conceptual model formulation

The conceptual model is formulated using a Programming Descriptive Language (PDL) that describes the set-up of the computer model using a combination of the English language and some key programming words. The PDL uses colours to highlight certain keywords like: component identifiers, simulation actions, and Python keywords.

The model is structured by formulating components for all the elements of the pipe yard system and its environment as defined in Section 2.3. Each component is formulated by a set of attributes and a process. Components can interact with eachother through process interaction, which can be recognized by a combination of a component identifier and a simulation action. Components can also interact with the environment that manages attributes and parameters that are relevant to all components.

C.1. Environment

The environment stores the following attributes that can be referenced by the components.

Attributes

```
position_map
routing_positions
occupancy_map
move_map_to_marking
move_map_to_hold
move_map_to_cleaning
move_map_to_djf
move_map_djf
```

C.2. Components

All components have attributes and a main process that describes the actions of a component and how it interacts with other components. A type of component may be generated multiple times, either at the start of during the simulation. Most attributes are written to self within the component, indicating that this attribute is specific for this generated component and the value may be different than a similar component with the same attribute. The main process for each component is an infinite loop that may be controlled by functions like hold, wait or passivate and can be reactivated by other components.

C.2.1. PSV

Attributes

```
joint_amount = parameter
anode_ratio = parameter
joint_list = []
```

Process

```
set anode_amount to joint_amount * anode_ratio
set regular_amount to joint_amount - anode_amount
for regular_amount:
    generate Joint with anode is False
for anode_amount:
    generate Joint with anode is True
add Joints to self.joint_list with regular anode interval
passivate self
```

C.2.2. PTC

Attributes

```
position = parameter
PSV // reference to PSV
counter = 0 // counter for amount of joints delivered
lifting_time_min = parameter // for triangular distribution
lifting_time_mode = parameter // for triangular distribution
lifting_time_max = parameter // for triangular distribution
drop_time = parameter / constant
wait_memory = []
```

Process

```
while there are joints in self.PSV.joint_list:
    get first Joint from  joint_list
    add 1 to self.counter
    set Joint.id to self.counter
    hold self for self.Subprocess_lifting_time
```

```
    if self.position is not free:
        wait self until self.position is free
        store waiting time in self.wait_memory

    set self.position in occupancy_map to reserved
    hold self for drop_time

    set Joint.position to self.position
    set self.position in occupancy_map to occupied
    set Joint.waiting_for_move to True
    add Joint to PipeRoutingSystem.joint_list
passivate self
```

Subprocess_Lifting_time

```
return random from triangular distribution with min=self.lifting_time_min, mode=
    self.lifting_time_mode, max=self.lifting_time_max
```

C.2.3. PipeRoutingSystem

Attributes

```
joint_list = []
joint_with_remainder_list = []
```

Process

```
set move_assigned to False
for each Joint waiting for a move in joint_list:
    if Joint is not marked:
        get Joint.destination and Joint.resource from move_map_marking
    elif Joint assigned to hold:
        get Joint.destination and Joint.resource from move_map_hold
    elif Joint is marked and not cleaned:
        get Joint.destination and Joint.resource from move_map_cleaning
    elif Joint is marked and cleaned:
        get Joint.destination and Joint.resource from move_map_djf
    if Joint.destination is None:
        continue

    get move and remainder from self.Subprocess_Path(Joint.position, Joint.
        destination)

    if there is a move and resource is None or available:
        set Joint.move to move
        set Joint.remainder to remainder
        for each position in move:
            set position in occupancy map to reserved
        activate Joint
        set move_assigned to True

        if there is a remainder:
            add Joint to self.joint_with_remainder_list

        break

if move_assigned is False:
    if TC2ps is passive and in cleaning position:
        set TC2ps.new_y to position above cleaning
        activate TC2ps
    passivate self
```

Subprocess_Path (start_position, end_position)

```
get start_x and start_y from start_position
get end_x and end_y from end_position
move = []
remainder = []

if start_x is not equal to end_x:
    add all reachable positions between (start_x, start_y) and (end_x, start_y) to
        move
    add all unreachable positions between (start_x, start_y) and (end_x, start_y)
       to remainder
if start_y is not equal to end_y:
    add all reachable positions between (start_x, start_y) and (start_x, end_y) to
        move
    add all unreachable positions between (start_x, start_y) and (start_x, end_y)
       to remainder

return move and remainder
```

Subprocess_Path_update

```
for each Joint in self.joint_with_remainder_list:
    if Joint.path_updateable is True:
        get move_extend and new_remainder from Subprocess_Path(Joint.move[-1],
            Joint.destination)

        if there is a move_extend:
            extend Joint.move with move_extend
            set Joint.remainder to new_remainder
            for each position in move_extend:
                set position in occupancy_map to reserved
            if there is no new_remainder:
                remove Joint from self.joint_with_remainder_list
```

Subprocess_joint_to_hold

```
// Activated by marking station port side
if there is a Joint waiting for a move next to marking station:
    set Joint.to_hold to True
```

C.2.4. Joint

Attributes

```
id = None // Assigned by PTC after delivery
length = joint_length
anode = boolean // True or False
position = parameter // (x, y)
x, y = position_map(position) // Get x and y coordinates from position map
waiting_for_move = False // Indicates if Joint is waiting for move
move = None
destination = None
path_updateable = False // Indicates if Joint path can be updated during move
marked = False // Indicates if Joint is marked
cleaned = False // Indicates if Joint is cleaned
to_hold = False // Indicates if Joint is sent to hold
```

Process

```
passivate self // Activated by PipeRoutingSystem after assigned a move

set self.waiting_for_move to False
set self.path_updateable to True
```

```
new_position is last position in self.move
get new_x and new_y from new_position in position_map

if new_x is not equal to self.x:
    Subprocess_joint_lco_move(self, self.move)
    set occupancy self.position to occupied

if new_y is not equal to self.y:
    resource is self.resource
    request resource
    set resource.joint to self
    set resource.new_y to self.y
    activate resource
    passivate self // Reactivated when resource has lifted joint

    Subprocess_joint_y_move(self, TC, self.move)
    passivate self // Reactivated when resource has dropped joint

    release resource
    set occupancy self.position to occupied

set self.destination to None
set self.move to None

if self.position is a processing station:
    passivate self // Reactived by processing station

if self.move is None:
    set self.waiting_for_move to True
    activate PipeRoutingSystem
```

C.2.5. DoubleJoint

Attributes

```
aft // Reference to aft joint
fore // Reference to fore joint
id // Double joint id is combination of single joint id's (aft.id, fore.id)
length = 2 * joint_length
anode // True if aft.anode or fore.anode is True, else False
position // Grid coordinate (x, y)
x, y // Map coordinate from position in position_map
waiting_for_move = False // Indicates if Joint is waiting for move
marked = True // Indicates if DoubleJoint is marked
cleaned = True // Indicates if DoubleJoint is cleaned
to_hold = False // Indicates if DoubleJoint is sent to hold
```

Process

```
passivate self // Activated by processing stations

set self.waiting_for_move to False
get new_x and new_y from self.destination in position_map
// Longitudinal move
if new_x is not equal to self.x:
    Subprocess_joint_lco_move(self, self.move)
    set occupancy self.position to occupied
// Transverse move
if new_y is not equal to self.y:
    resource is self.resource
    request resource
    set resource.joint to self
    set resource.new_y to self.y
```

```
    activate resource
    passivate self // Reactivated when resource has lifted joint

    if resource is a crane:
        Subprocess_joint_y_move(self, Crane, self.move)
    else:
        Subprocess_joint_y_move(self, TC, self.move)
    passivate self // Reactivated when resource has dropped joint

    release resource
    set occupancy self.position to occupied

set self.destination to None
set self.move to None

if self.position is a processing station:
    passivate self // Reactived by processing station

if self.move is None:
    set self.waiting_for_move to True
    if self in PipeRoutingSystem.joint_list:
        activate PipeRoutingSystem
    elif self in WaitingArea.doublejoint_list:
        activate WaitingArea
```

C.2.6. TC

Attributes

```
position = parameter // (x, y)
lift_time = parameter \\ constant
drop_time = parameter \\ constant
joint // reference to joint being moved
```

Process

```
passivate self // activated by Joint request or PipeRoutingSystem
Subprocess_y_move(self, TC, self.new_position) \\ new_position set by activation
if requested for Joint move:
    hold self for lift_time
    activate Joint
    Subprocess_y_move(self, TC, Joint.move)
    hold self for drop_time
    activate Joint
```

C.2.7. MarkingStation

Attributes

```
position = parameter // (x, y)
marking_time = parameter
```

Process

```
wait self until self.position is occupied by Joint
hold self for marking_time
set Joint.marked to True
activate PipeRoutingSystem.Subprocess_joint_to_hold
activate Joint
```

C.2.8. CleaningStation

Attributes

```
position = parameter // (x, y)
cleaning_time = parameter
```

Process

```
wait self until self.position is occupied by Joint
hold self for cleaning_time
set Joint.cleaned to True
activate Joint
```

C.2.9. Hold

Attributes

```
position = parameter // (x, y)
storage_location = parameter // location name
storage_active = boolean // True or False
retrieval_active = boolean // True or False
lift_time = parameter
storage_time = parameter
retrieval_time = parameter
storage_list = []
```

Process

```
while self.storage_active is True:
    wait self until self.position is occupied by Joint
    if Joint.to_hold is True:
        hold self for lift_time
        set Joint.position to storage_location
        set self.position in occupancy_map to free
        hold self for storage time
        add Joint to self.storage_list
        remove Joint from PipeRoutingSystem.joint_list
```

C.2.10. DJF

Attributes

```
position_aft = parameter // supply position aft (x, y)
positoin_fore = parameter // supply position fore (x, y)
wait_memory_ps = []
wait_memory_sb = []
```

Process

```
wait self until self.position_aft and self.position_fore are occupied in
    occupancy_map
// or be activated by Heating Station after bevelling position is free
if there is a Joint_aft at position_aft and a Joint_fore at position_fore:
    if the DJF was waiting:
        store waiting time in self.wait_memory

    if bevelling positions at port side are free in occupancy_map:
        set Joint_aft.destination to Bevelling_ps_aft.position and set Joint_aft.
            resource to TC3ps
        set Bevelling_ps_aft.Joint to Joint_aft
        set Joint_fore.destination to Bevelling_ps_fore.position and set
            Joint_fore.resource to TC4ps
        set Bevelling_ps_fore.Joint to Joint_fore
        for Joint_aft and Joint_fore:
            remove joint from PipeRoutingSystem.joint_list
            set Joint.djf to True
            set Joint.move to Joint.destination
```

```
            set Joint.destination in occupancy_map to reserved
            activate Joint
        if port side was waiting:
            store waiting time in self.wait_memory_ps
    elif bevelling positions at starboard are free in occupancy_map:
        set Joint_aft.destination to Bevelling_sb_aft.position and set Joint_aft.
            resource to TC3sb
        set Bevelling_sb_aft.Joint to Joint_aft
        set Joint_fore.destination to Bevelling_sb_fore.position and set
            Joint_fore.resource to TC4sb
        set Bevelling_sb_fore.Joint to Joint_fore
        for Joint_aft and Joint_fore:
            remove joint from PipeRoutingSystem.joint_list
            set Joint.djf to True
            set Joint.move to [Joint.destination]
            set Joint.destination in occupancy_map to reserved
            activate Joint
        if starboard was waiting:
            store waiting time in self.wait_memory_sb
    else:
        passivate self // no bevelling position free, reactivated by heating
            station when free
else:
    if bevelling positions at port side are free:
        port side start waiting
    elif bevelling positions at starboard are free:
        starboard start waiting
```

C.2.11. BevelStation

Attributes

```
position = parameter // (x, y)
bevel_time = parameter // probability distribution
```

Process

```
wait self until self.position in occupancy_map is occupied by Joint
hold self for bevel_time
activate Joint
```

C.2.12. HeatingStation

Attributes

```
position = parameter // (x, y)
heating_time = parameter
next_station // reference to next station
```

Process

```
wait self until self.position in occupancy_map is occupied by Joint
activate DJF
hold self for heating_time
wait self until next_station.position in occupancy_map is free
activate Joint
```

C.2.13. WeldingStation1

Attributes

```
position = parameter // (x, y)
position_aft = parameter // (x, y)
position_fore = parameter // (x, y)
welding_time = parameter
next_station // reference to next station
```

Process

```
wait self until self.position_aft and self.position_fore in occupancy_map are
    occupied by joint_aft and joint_fore
set self.position in occupancy map to occupied
hold self for welding_time

// remove single joints from simulation
remove joint_aft and joint_fore from PipeRoutingSystem.joint_list
set self.position_aft in occupancy map to free
set self.position_fore in occupancy map to free
if joint_aft.anode is False and joint_fore.anode is False
    generate Doublejoint with anode is False
elif joint_aft.anode is True or joint_fore.anode is True:
    generate DoubleJoint with anode is True
add DoubleJoint to PipeRoutingSystem.joint_list
wait self until next_station.position in occupancy_map is free

set DoubleJoint.destination to move_map_djf(self.position)
set DoubleJoint.move to [DoubleJoint.destination]
set DoubleJoint.destination in occupancy_map to reserved
activate DoubleJoint
```

C.2.14. WeldingStation2

Attributes

```
position = parameter // (x, y)
welding_time = parameter
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
hold self for welding_time
activate DoubleJoint
```

C.2.15. WeldingStation34

Attributes

```
position = parameter // (x, y)
welding_time = parameter
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
hold self for welding_time
activate DoubleJoint
```

C.2.16. NDT

Attributes

```
position = parameter // (x, y)
testing_time = parameter
reject_probability = parameter between 0 and 1
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
hold self for testing_time
if random < self.reject_probability: // random number between 0 and 1
    set DoubleJoint.reject to True
activate Waiting Area
activate DoubleJoint
```

C.2.17. Crane

Attributes

```
position = parameter // (x, y)
hoist_time = parameter
hoist_time_firingline = parameter
idle = Boolean // True or False
DoubleJoint // reference to doublejoint being moved
```

Process

```
set self.idle to True
passivate self // activated by Waiting Area
set self.idle to False
Subprocess_y_move(self, Crane38, self.new_position) // new_position set by
    activation

if requested for FiringLine rejects:
    set lift_time to 2 * self.hoist_time_firingline
    hold self for lift_time // lowering the empty crane and lifting with double
        joint
    while Deconstructor.position is not free:
        wait self until Deconstructor.position is free
    set Deconstructor.position to reserved
elif requested for DoubleJoint move:
    set lift_time to 2 * self.hoist_time
    hold self for lift_time // lowering the empty crane and lifting with double
        joint
else:
    return

activate DoubleJoint
Subprocess_y_move(self, Crane38, self.DoubleJoint.move)
hold self for self.hoist_time // dropping the double joint
activate DoubleJoint
hold self for self.hoist_time // lifting the empty crane
```

C.2.18. Buffer

Attributes

```
position = parameter // (x, y)
retrieval = False // Indicates if buffer is retrieving a joint for waiting area
DJ_list = [] // list of double joints in buffer
ADJ_list = [] // list of anode double joints in buffer
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
// or be activated by something else
if there is a DoubleJoint:
    set DoubleJoint.position to buffer
    if DoubleJoint.anode = False:
        add DoubleJoint to self.DJ_list
    elif DoubleJOint.anode = True:
        add DoubleJoint to self.ADJ_list
    set self.position in occupancy_map to free
elif self.retrieval is True:
    if FiringLine.anode is False:
        get DoubleJoint from self.DJ_list
    elif FiringLine.anode is True:
        get DoubleJoint from self.ADJ_list
    set DoubleJoint.position to self.position
```

```
    set DoubleJoint.destination to Elevator.position
    set DoubleJoint.move to [DoubleJoint.destination]
    set DoubleJoint.resource to Crane38
    set DoubleJoint.destination in occupancy_map to reserved
    activate DoubleJoint
    set self.retrieval to False
```

C.2.19. WaitingArea

Attributes

```
crane // reference to Crane38
reserve_anode = Boolean // True or False
doublejoint_list = []
```

Process

```
set doublejoint_sb to DoubleJoint at starboard position or None
set doublejoint_ps to DoubleJoint at port side position or None
set doublejoint_reserve to DoubleJoint at reserve position or None

if WaitingArea was not waiting for normal doublejoint:
    if FiringLine is waiting and Elevator is free but normal doublejoint not
        available:
    start waiting
elif waitingArea was waiting for normal doublejoint:
    if normal doublejoint is available:
        store wait time
if WaitingArea was not waiting for anode doublejoint:
    if FiringLine is waiting and Elevator is free but anode doublejoint not
        available:
    start waiting
elif waitingArea was waiting for anode doublejoint:
    if normal doublejoint is available:
        store wait time

if FiringLine has rejects:
    for all DoubleJoint in FiringLine.reject_list
        while self.crane is not idle:
            wait self until self.crane is idle
        set DoubleJoint.destination to Deconstructor.position
        set DoubleJoint.move to [DoubleJoint.destination]
        set DoubleJoint.resource to Crane38
        activate DoubleJoint
        remove DoubleJoint from FiringLine.reject_list
    activate FiringLine
elif Elevator.position is free and doublejoint_sb is no reject and correct anode:
    self.Subprocess_move(doublejoint_sb, Elevator.position)
elif Elevator.position is free and doublejoint_ps is no reject and correct anode:
    self.Subprocess_move(doublejoint_ps, Elevator.position)
elif Elevator.position is free and doublejoint_reserve is no reject and correct
    anode:
    self.Subprocess_move(doublejoint_reserve, Elevator.position)
elif Elevator.position is free and FiringLine.anode is False and there are double
    joints in Buffer.DJ_list:
    set Buffer.retrieval to True
    activate Buffer
elif Elevator.position is free and FiringLine.anode is True and there are anode
    double joints in Buffer.ADJ_list:
    set Buffer.retrieval to True
    activate Buffer
elif doublejoint_ps is a reject and Deconstructor.position is free:
    self.Subprocess_move(doublejoint_ps, Elevator.position)
```

```
elif doublejoint_sb is a reject and Deconstructor.position is free:
    self.Subprocess_move(doublejoint_sb, Elevator.position)
elif doublejoint_ps is no reject and NDT_ps is waiting:
    if reserve spot is free:
        self.Subprocess_move(doublejoint_ps, reserve position)
    else:
        self.Subprocess_move(doublejoint_ps, Buffer.position)
elif doublejoint_sb is no reject and NDT_sb is waiting:
    if reserve spot is free:
        self.Subprocess_move(doublejoint_sb, reserve position)
    else:
        self.Subprocess_move(doublejoint_sb, Buffer.position)
else:
    passivate self // no actions available for waiting area

while self.Crane is not idle:
    wait self until self.Crane is idle
```

Subprocess_move(doublejoint, destination)

```
set doublejoint.destination to destination
set doublejoint.move to [doublejoint.destination]
set doublejoint.resource to Crane38
set doublejoint.destination in occupancy_map to reserved
activate doublejoint
```

C.2.20. Elevator

Attributes

```
position = position // (x, y)
elevator_time_empty = parameter
elevator_time_loaded = parameter
unloading_time = parameter
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
hold self for self.elevator_time_loaded
hold self for self.unloading_time
wait self until FiringLine is ready to sent Elevator back
hold self for self.elevator_time_empty
set self.position in occupancy_map to free
activate WaitingArea
```

C.2.21. FiringLine

Attributes

```
position = position // (x, y)
processing_time = parameter
processing_time_pull = parameter
reject_length = parameter
reject_probability = parameter between 0 and 1
anode_interval = parameter
anode = Boolean // True or False
counter = 0 // counting length of main pipeline
pipeline_list = [] // list for storing double joints in main pipeline
reject_list = []
wait_memory = []
```

Process

```
if Elevator has not delivered DoubleJoint:
    wait until DoubleJoint is delivered
    store waiting time in self.wait_memory
send Elevator back
set DoubleJoint.position to self.position
add 1 to self.counter
if self.counter / self.anode_interval has no remainder:
    set self.anode to True
else:
    set self.anode to False
add DoubleJoint to self.pipeline_list
if len(self.pipeline_list) > self.reject_length:
    remove first DoubleJoint from self.pipeline_list
    remove first DoubleJoint from DJF.doublejoint_list
hold self for processing_time
if random < self.reject_probability // random number between 0 and 1
    set self.reject_list to self.pipeline_list in opposite order
    subtract len(self.reject_list) from self.counter
    for all DoubleJoint in self.reject_list:
        set DoubleJoint.reject to True
        remove DoubleJoint from self.pipeline_list
        set DoubleJoint.position to self.position
        activate WaitingArea
        while there are DoubleJoint in self.reject_list:
            passivate self
else:
    hold self for processing_time_pull
```

C.2.22. Deconstructor

Attributes

```
position = parameter // (x, y)
position_aft = parameter // (x, y)
position_fore = parameter // (x, y)
deconstruction_time = parameter
```

Process

```
wait self until self.position in occupancy_map is occupied by DoubleJoint
set aft_joint to DoubleJoint.aft
set fore_joint to DoubleJoint.fore
hold self for deconstruction_time
// remove double joint from simulation and add single joints again
remove DoubleJoint from WaitingArea.doublejoint_list
set aft_joint.position to self.position_aft
set aft_joint.position to self.position_fore
for aft_joint and fore_joint:
    set Joint.position in occupancy map to occupied
    set Joint.djf to False
    set Joint.cleaned to False
    set Joint.to_hold to True
    add Joint to front of PipeRoutingSystem.joint_list
    activate Joint
wait self until self.position_aft and self.position_fore in occupancy_map are free
set self.position in occupancy_map to free
activate WaitingArea
```

C.2.23. General Subprocesses

Subprocess_joint_lco_move (self, move)

```
// Acceleration
```

```
dx is direction * lco_acceleration_distance
animate interpolated movement of dx at 0.5 lco velocity
set self.x to interpolated_x
// Constant velocity
for each position in move:
    dx is x_position - self.x
    if position is final position:
        dx is dx - lco_acceleration_distance
    animate interpolated movement for dx at lco velocity
    after leaving old position set old position in occupancy_map to free
    set self.x to interpolated_x
    set self.position to position
// Deceleration
set self.path_updateable to False
dx is direction * lco_acceleration_distance
animate interpolated movement of dx at 0.5 lco velocity
set self.x to x from position in position_map
```

Subprocess_y_acceleration (self, component, direction

```
dy is direction * component_acceleration_distance
animate interpolated movement of dy at 0.5 component_velocity
set self.y to interpolated_y
```

Subprocess_y_deceleration (self, component, direction

```
set self.path_updateable to False
dy is direction * component_acceleration_distance
animate interpolated movement of dy at 0.5 component_velocity
set self.y to y from position in position_map
```

Subprocess_joint_y_move (self, component, move)

```
get direction
// Acceleration
Subprocess_y_acceleration(self, component, direction)
// Constant velocity
for each position in move:
    dy is y_position - self.y
    if position is final position:
        dy is dy - component_acceleration_distance
    animate interpolated movement for dy at component_velocity
    after leaving old position set old position in occupancy_map to free
    set self.y to interpolated_y
    set self.position to position
// Deceleration
Subprocess_y_deceleration(self, component, direction)
```

D

Process duration data

The process durations at the processing stations within the DJF should be derived from empirical data to accurately capture the stochastic nature of the operations in the simulation. However, there is a lack of accurate data available on the DJF within the Pipeline Production Department (PPD) at Allseas. PPD generally assumes the production of the DJF to be sufficient to supply the firing line and has therefore not been interested in accurately logging data for the DJF during projects. An attempt has been done to extract duration data from a datalogger, but the output shows a flattened distribution with an overestimated mean for the duration of the welding stations. This could be explained by inconsistent behaviour of operators at the end of their process, as operators wait for eachother before pressing the ready button. This appendix will describe how the distributions of process durations are estimated based on a performed video analysis and measured average values provided by PPD.

D.1. Video analysis

During the South-East Extension (SEE) project, 20 hours of continuous production has been recorded on Solitaire.The available recordings will be used to verify the estimated average durations provided by PPD, but also to analyse the process to estimate a minimum and maximum value and a distribution function for the stochastic station duration. The video analysis will be performed for bevelling, welding 1, welding 2 and welding 3/4.

The recording is split into four videos of 5 hours and as their are two welding lines, a total of eight videos is available for each station. Five measurements per video provide a total of 40 duration logs from several operators and production phases. Table D.1 presents the mean, maximum and minimum values measured per station. Table D.2, Table D.3, Table D.4 and Table D.5 show all 40 duration logs sampled per station.

Table D.1: Summary of duration logs from video analysis

Data	Bevelling	Welding 1	Welding 2	Welding 3/4
Mean	276 s	750 s	695 s	545 s
Max	505 s	971 s	1014 s	895 s
Min	196 s	553 s	489 s	398 s

Table D.2: Process duration logs for bevelling from video analysis

	Videos port side				Videos starboard			
Data	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**
Sample 1	263	202	292	357	272	287	258	217
Sample 2	258	213	236	236	221	430	287	456
Sample 3	338	217	244	505	268	288	227	250
Sample 4	319	208	196	215	315	293	226	244
Sample 5	494	210	200	277	252	271	257	240

Table D.3: Process duration logs for welding 1 from video analysis

	Videos port side				Videos starboard			
Data	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**
Sample 1	707	836	798	581	800	819	553	732
Sample 2	792	804	647	595	641	883	782	753
Sample 3	788	818	775	803	839	932	971	803
Sample 4	784	811	836	591	831	891	561	746
Sample 5	785	794	771	667	599	569	578	748

Table D.4: Process duration logs for welding 2 from video analysis

	Videos port side				Videos starboard			
Data	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**
Sample 1	523	750	694	631	638	759	640	699
Sample 2	701	941	638	501	539	749	695	827
Sample 3	998	717	692	886	1014	489	798	694
Sample 4	703	722	650	647	747	626	538	687
Sample 5	755	726	505	687	653	694	679	570

Table D.5: Process duration logs for welding 3/4 from video analysis

	Videos port side				Videos starboard			
Data	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**	**00h–05h**	**05h–10h**	**10h–15h**	**15h–20h**
Sample 1	439	454	450	445	464	700	526	449
Sample 2	398	771	560	504	483	711	698	449
Sample 3	716	610	655	895	675	555	776	414
Sample 4	425	428	482	596	432	445	461	452
Sample 5	468	450	656	649	418	608	524	495

D.2. Theoretical probability density function

The stochastic variation of the process durations can be described by an estimated probability density function (PDF), which is used to describe the probability that a continuous random variable will fall within a specified range (Kissell et al. 2017). This section will describe which distribution function will be used to design the probability density function and how this probability density function will be shaped.

D.2.1. Distribution functions

In general process durations are positively skewed as the minimum process duration makes that a process slower than average has a larger potential deviation than a process faster than average. A well-known positively skewed distribution is the log-normal distribution which can be fitted through the data logs. However, their is insufficient data logs available to assume a fitted log-normal to be accurate and for the purpose of this research it would be better to introduce a maximum value to exclude exceptional process durations that are too influential on the simulation results.

A triangular distribution is a very simple solution that aligns with the available data as it only requires the minimum, maximum and mode value. It is often used in simulations when there is insufficient data available (Kissell et al. 2017). The PERT distribution requires the same values, but generates a distribution that more closely resembles a realistic probability distribution (RiskAMP n.d.). PERT is actually a transformation of the more well known Beta distribution that is defined by two shape parameters. It is preferred to define the probability density function as a Beta distribution as this can be implemented directly in the simulation program Salabim.

D.2.2. Deriving Beta function shape parameters

This subsection will describe how the shape parameters of the Beta distribution can be derived based on a mean, minimum and maximum value using the methodology of the PERT distribution as described in Buchsbaum 2012. As the most reliable value available for the station durations is the measured mean value provided by PPD, this will form the basis for shaping the distribution function.

Formula D.1 shows how the mean in modified PERT can be calculated given the minimum value min, maximum value max, most common value $mode$ and shape factor λ. This formula can be transformed to Formula D.2, using the $mean$ as input to calculate the $mode$.

$$mean = \frac{(min + \lambda * mode + max)}{(\lambda + 2)} \quad \text{(Buchsbaum 2012)} \tag{D.1}$$

$$mode = \frac{mean * (\lambda + 2) - min - max}{\lambda} \tag{D.2}$$

The $mean$ and $mode$ for a Beta distribution can be calculated with the shape parameters α and β as shown in Formula D.3 and Formula D.4.

$$mean = \frac{\alpha}{\alpha + \beta} \quad \text{(Peleg 2019)} \tag{D.3}$$

$$mode = \frac{\alpha - 1}{\alpha + \beta - 2} \quad \text{(Peleg 2019)} \tag{D.4}$$

As the $mean$ is given as an input and the $mode$ is calculated based on a given min, max and λ, we are left with two functions and two variables α and β that can be transformed to Formula D.5 and Formula D.6.

$$\alpha = 1 + \lambda \frac{mode - min}{max - min} \tag{D.5}$$

$$\beta = 1 + \lambda \frac{max - mode}{max - min} \tag{D.6}$$

These two formulas calculate the shape parameters that form the Beta function $Beta(\alpha, \beta)$. As the Beta function is a distribution function between 0 and 1, the actual probability density function is described by Formula D.7.

$$PDF = min + Beta(\alpha, \beta) * (max - min) \tag{D.7}$$

To create a distribution function with two peaks, a bimodal Beta function can be formulated given two Beta functions and a weight factor w as shown in Formula D.8.

$$Beta_{bimodal} = w_1 * Beta_1(\alpha_1, \beta_1) + w_2 * Beta_2(\alpha_2, \beta_2) \quad \text{, with } w_1 + w_2 = 1 \tag{D.8}$$

D.3. Probability density functions for process durations

The previous section derived how the PDF for a process duration can be formulated as a Beta distribution given the average duration ($mean$), the minimum duration (min), the maximum duration (max) and a shape factor (λ). This section provides the probability density functions for the process duration of bevelling, welding 1, welding 2 and welding 3/4.

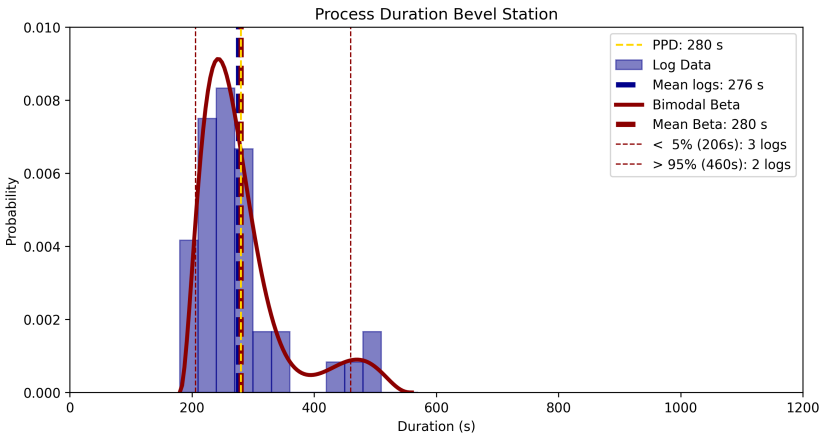
The $mean$ value for all processes have been measured by PPD. The min and max value are estimated based on the data of the video analysis and evaluated using the 5th and 95th percentile. The $mode$ value can be calculated using Formula D.2. The shape factor λ is determined iteratively, shaping the distribution to fit the histogram of the video analysis. To include the second peak in the bevelling duration, this PDF is formulated as bimodal Beta function consisting of two Beta functions with the same min and max value that are combined using a weight factor w. The final values for the Beta distributions are summarized in Table D.6 and the resulting probability density functions are shown in Figure D.1, plotted over the video analysis data.

In Table D.6 it can be seen that the min and max values show similar percentages compared to the $mean$ value. All processes show a decrease around 30% for the min value and the shorter processes show an increase around 100% for the max value while the longer processes show an increase around 75%. Figure D.1 highlights the 5th and 95th percentile and the amount of logs from the video analysis that are within these ranges. Every range is expected to have around 2 logs (5% from 40 logs) which matches with all stations except welding station 3/4 as the video analysis logged many high process durations.

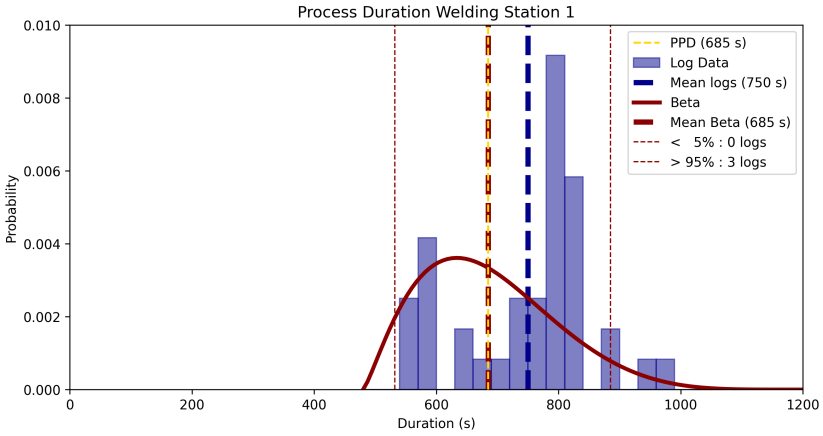
Without performing an actual goodness of fit test, it is clear that the log data does not fit the beta distribution well enough for the welding stations to conclude that the beta distribution is an accurate estimation of reality. The difference between the mean provided by PPD and the video analysis data is too large to formulate a PDF fitting both data. As earlier mentioned the mean value provided by PPD is assumed to be the most reliable and therefore the proposed distributions based on this mean value are the most accurate data that can be generated with the available resources.

Table D.6: Beta distribution values

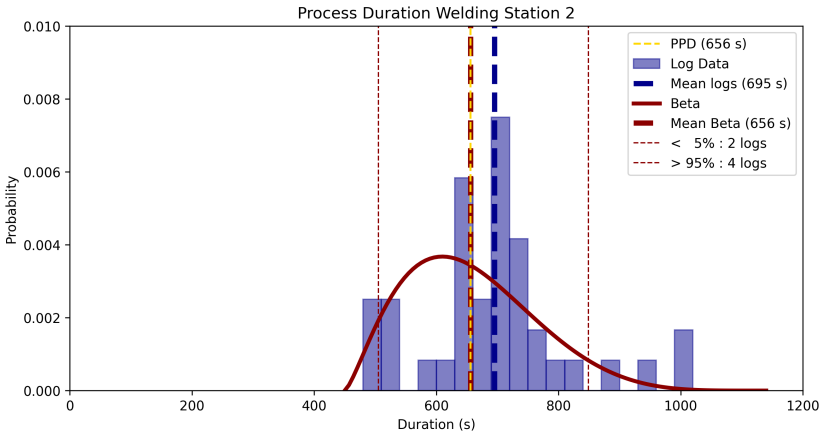
Process	Mean	Min	Max	Mode	λ	Beta:	α	β	w
Bevelling (1)	280	180 *(-36%)*	560 *(+100%)*	243	12		2.99	11.01	0.9
Bevelling (2)	280	180 *(-36%)*	560 *(+100%)*	470	12		10.16	3.84	0.1
Welding 1	685	480 *(-30%)*	1200 *(+75%)*	633	6		2.27	5.72	
Welding 2	656	450 *(-31%)*	1140 *(+74%)*	610	6		2.39	5.61	
Welding 3/4	457	330 *(-28%)*	900 *(+97%)*	404	6		1.78	6.22	



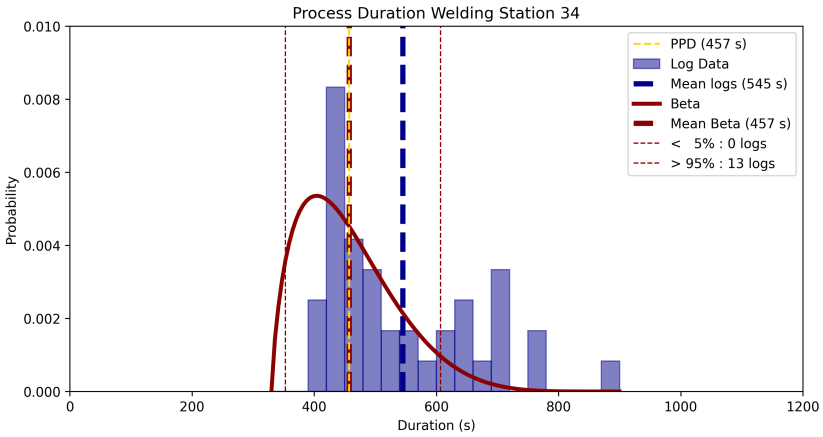
(a) Bevelling



(b) Welding 1



(c) Welding 2



(d) Welding 3/4

Figure D.1: Probability density functions for process durations

E

Experimental run-length selection

This appendix presents the simulation output results used to determine the experimental run-length based on the convergence of the following output parameters: DJF production, main pipeline length, PTC wait time, DJF wait time, and waiting area wait time.

E.1. DJF production

Table E.1: Convergence of the cumulative mean for the DJF production in joints per day

Day	Run 1	Run 2	Run 3	Run 4	Run 5	Min	Max	Convergence
1	207.0	206.0	207.0	205.0	205.0	205.0	207.0	0.98%
2	208.0	208.0	208.5	205.5	205.5	205.5	208.5	1.46%
3	208.3	208.0	207.7	207.0	206.0	206.0	208.3	1.13%
4	208.5	208.0	208.0	207.5	206.5	206.5	208.5	0.95%
5	208.2	207.4	207.4	207.8	206.8	206.8	208.2	0.67%
6	207.5	207.0	207.0	207.8	206.1	206.1	207.8	0.82%
7	208.0	207.1	207.1	208.0	206.4	206.4	208.0	0.75%
8	207.7	207.4	207.4	207.5	206.1	206.1	207.7	0.80%
9	207.8	207.2	207.5	207.7	206.0	206.0	207.8	0.88%
10	207.7	207.2	207.8	207.9	206.1	206.1	207.9	0.87%
11	207.5	207.0	207.5	207.8	206.4	206.4	207.8	0.68%
12	207.7	207.3	207.8	208.2	206.7	206.7	208.2	0.75%
13	207.3	207.0	207.4	208.0	206.7	206.7	208.0	0.63%
14	207.7	207.1	207.9	208.3	206.9	206.9	208.3	0.65%
15	207.9	207.4	207.9	208.1	207.0	207.0	208.1	0.53%
16	208.0	207.4	207.9	208.2	207.0	207.0	208.2	0.57%
17	207.8	207.2	207.7	208.2	206.9	206.9	208.2	0.61%
18	207.8	207.2	207.8	208.2	206.9	206.9	208.2	0.65%
19	207.5	207.0	207.7	208.0	206.8	206.8	208.0	0.55%
20	207.5	207.0	207.5	207.9	206.8	206.8	207.9	0.54%
21	207.5	207.0	207.4	208.0	206.8	206.8	208.0	0.55%
22	207.5	207.1	207.6	208.0	206.9	206.9	208.0	0.51%
23	207.5	207.0	207.5	207.9	206.9	206.9	207.9	0.46%
24	207.4	207.0	207.5	207.9	206.9	206.9	207.9	0.48%
25	207.6	207.2	207.7	207.9	207.0	207.0	207.9	0.40%
26	207.5	207.3	207.6	207.9	207.0	207.0	207.9	0.46%
27	207.5	207.4	207.7	207.9	207.0	207.0	207.9	0.42%
28	207.4	207.3	207.6	207.8	207.0	207.0	207.8	0.42%
29	207.3	207.4	207.6	207.9	207.0	207.0	207.9	0.44%
30	207.5	207.4	207.6	207.8	206.9	206.9	207.8	0.42%
31	207.5	207.4	207.5	207.8	206.9	206.9	207.8	0.44%
32	207.5	207.3	207.6	207.8	207.0	207.0	207.8	0.40%
33	207.5	207.4	207.6	207.9	207.0	207.0	207.9	0.40%
34	207.6	207.4	207.7	207.8	207.0	207.0	207.8	0.36%
35	207.5	207.4	207.6	207.7	207.0	207.0	207.7	0.34%
36	207.5	207.5	207.7	207.8	207.0	207.0	207.8	0.38%
37	207.5	207.5	207.8	207.7	207.0	207.0	207.8	0.38%
38	207.5	207.3	207.7	207.7	207.0	207.0	207.7	0.36%
39	207.5	207.4	207.7	207.8	207.0	207.0	207.8	0.42%
40	207.6	207.5	207.7	207.9	207.0	207.0	207.9	0.42%
41	207.6	207.5	207.8	207.9	207.2	207.2	207.9	0.32%
42	207.6	207.5	207.9	208.0	207.3	207.3	208.0	0.34%
43	207.6	207.5	207.9	207.9	207.2	207.2	207.9	0.34%
44	207.6	207.6	208.0	208.0	207.2	207.2	208.0	0.40%
45	207.7	207.6	208.0	208.0	207.3	207.3	208.0	0.36%
46	207.7	207.5	208.0	208.0	207.3	207.3	208.0	0.38%
47	207.7	207.5	208.0	208.0	207.3	207.3	208.0	0.36%
48	207.6	207.5	208.0	208.0	207.2	207.2	208.0	0.40%
49	207.5	207.5	208.0	208.0	207.3	207.3	208.0	0.34%
50	207.5	207.4	208.0	208.0	207.3	207.3	208.0	0.34%

E.2. Main pipeline length

Table E.2: Convergence of the cumulative mean for the main pipeline length in meters per day

Day	Run 1	Run 2	Run 3	Run 4	Run 5	Min	Max	Convergence
1	5026	5026	5100	4978	5002	4978	5100	2.45%
2	5075	5063	5100	5014	5014	5014	5100	1.72%
3	5091	5083	5059	5043	5018	5018	5091	1.45%
4	4984	4965	4959	4959	4929	4929	4984	1.12%
5	5022	5002	5007	5017	4987	4987	5022	0.70%
6	4929	4917	4908	4921	4888	4888	4929	0.84%
7	4936	4911	4915	4929	4890	4890	4936	0.94%
8	4910	4895	4895	4914	4880	4880	4914	0.70%
9	4913	4904	4910	4913	4872	4872	4913	0.84%
10	4914	4900	4914	4917	4870	4870	4917	0.97%
11	4896	4882	4900	4898	4884	4882	4900	0.37%
12	4900	4890	4910	4913	4874	4874	4913	0.80%
13	4904	4906	4916	4925	4901	4901	4925	0.49%
14	4932	4915	4934	4944	4910	4910	4944	0.69%
15	4942	4930	4945	4953	4924	4924	4953	0.59%
16	4929	4918	4930	4935	4906	4906	4935	0.59%
17	4932	4919	4930	4936	4909	4909	4936	0.55%
18	4914	4902	4918	4925	4896	4896	4925	0.59%
19	4894	4881	4898	4903	4874	4874	4903	0.59%
20	4898	4887	4903	4909	4881	4881	4909	0.57%
21	4894	4877	4892	4903	4877	4877	4903	0.53%
22	4898	4883	4899	4908	4880	4880	4908	0.57%
23	4878	4866	4880	4891	4863	4863	4891	0.58%
24	4892	4879	4894	4900	4879	4879	4900	0.43%
25	4900	4893	4905	4908	4888	4888	4908	0.41%
26	4895	4882	4898	4902	4880	4880	4902	0.45%
27	4902	4895	4906	4910	4886	4886	4910	0.49%
28	4885	4882	4890	4895	4871	4871	4895	0.49%
29	4888	4888	4893	4900	4877	4877	4900	0.47%
30	4896	4892	4900	4903	4883	4883	4903	0.41%
31	4887	4886	4890	4897	4882	4882	4897	0.31%
32	4891	4887	4894	4899	4878	4878	4899	0.43%
33	4891	4890	4895	4901	4881	4881	4901	0.41%
34	4900	4897	4904	4904	4889	4889	4904	0.31%
35	4889	4886	4890	4891	4877	4877	4891	0.29%
36	4898	4894	4900	4902	4885	4885	4902	0.35%
37	4896	4894	4902	4902	4883	4883	4902	0.39%
38	4886	4882	4892	4892	4872	4872	4892	0.41%
39	4889	4886	4893	4894	4876	4876	4894	0.37%
40	4891	4887	4896	4896	4876	4876	4896	0.41%
41	4897	4896	4903	4903	4887	4887	4903	0.33%
42	4900	4900	4907	4910	4891	4891	4910	0.39%
43	4905	4904	4912	4914	4896	4896	4914	0.37%
44	4906	4903	4914	4913	4896	4896	4914	0.37%
45	4902	4900	4910	4909	4892	4892	4910	0.37%
46	4909	4906	4914	4916	4898	4898	4916	0.37%
47	4905	4902	4911	4913	4893	4893	4913	0.41%
48	4905	4901	4913	4914	4895	4895	4914	0.39%
49	4895	4891	4900	4899	4886	4886	4900	0.29%
50	4896	4893	4907	4907	4889	4889	4907	0.37%

E.3. PTC wait time

Table E.3: Convergence of the cumulative mean for the PTC wait time in minutes per day

Day	Run 1	Run 2	Run 3	Run 4	Run 5	Min	Max	Convergence
1	26	39	28	29	28	26	39	50.00%
2	37	37	16	18	17	16	37	131.25%
3	33	36	20	31	21	20	36	80.00%
4	37	41	24	26	23	23	41	78.26%
5	35	39	32	28	26	26	39	50.00%
6	31	38	30	26	23	23	38	65.22%
7	31	36	27	28	21	21	36	71.43%
8	32	33	26	28	22	22	33	50.00%
9	29	38	27	32	22	22	38	72.73%
10	30	39	27	32	23	23	39	69.57%
11	29	37	28	34	23	23	37	60.87%
12	29	36	28	33	22	22	36	63.64%
13	28	37	27	31	23	23	37	60.87%
14	28	35	27	31	23	23	35	52.17%
15	28	35	26	31	26	26	35	34.62%
16	32	34	25	30	27	25	34	36.00%
17	33	35	25	29	26	25	35	40.00%
18	32	34	25	29	28	25	34	36.00%
19	30	34	28	29	27	27	34	25.93%
20	31	35	28	29	27	27	35	29.63%
21	31	34	28	29	26	26	34	30.77%
22	31	33	29	29	26	26	33	26.92%
23	31	32	28	30	27	27	32	18.52%
24	31	32	28	32	27	27	32	18.52%
25	31	32	28	31	27	27	32	18.52%
26	30	32	28	31	26	26	32	23.08%
27	30	31	28	30	26	26	31	19.23%
28	30	31	28	30	26	26	31	19.23%
29	30	31	29	31	26	26	31	19.23%
30	30	31	30	30	26	26	31	19.23%
31	30	31	30	31	26	26	31	19.23%
32	29	30	30	31	26	26	31	19.23%
33	29	30	30	30	26	26	30	15.38%
34	29	31	31	30	26	26	31	19.23%
35	30	31	32	31	26	26	32	23.08%
36	29	31	32	31	26	26	32	23.08%
37	29	32	32	32	25	25	32	28.00%
38	30	31	31	32	25	25	32	28.00%
39	29	30	31	32	25	25	32	28.00%
40	29	30	31	32	25	25	32	28.00%
41	29	29	31	32	25	25	32	28.00%
42	29	29	30	32	25	25	32	28.00%
43	28	29	30	32	25	25	32	28.00%
44	29	29	30	32	24	24	32	33.33%
45	28	29	30	32	25	25	32	28.00%
46	28	29	29	31	26	26	31	19.23%
47	28	29	29	32	26	26	32	23.08%
48	28	29	28	31	26	26	31	19.23%
49	28	30	28	32	26	26	32	23.08%
50	28	30	28	31	26	26	31	19.23%

E.4. DJF wait time

Table E.4: Convergence of the cumulative mean for the DJF wait time in minutes per day

Day	Run 1	Run 2	Run 3	Run 4	Run 5	Min	Max	Convergence
1	47	41	46	61	61	41	61	48.78%
2	52	44	56	53	57	44	57	29.55%
3	48	49	56	52	60	48	60	25.00%
4	46	51	57	51	55	46	57	23.91%
5	51	47	59	53	55	47	59	25.53%
6	49	51	58	53	53	49	58	18.37%
7	52	49	57	51	52	49	57	16.33%
8	52	52	55	51	49	49	55	12.24%
9	50	54	58	50	48	48	58	20.83%
10	51	54	57	52	48	48	57	18.75%
11	50	52	54	53	48	48	54	12.50%
12	48	53	52	51	46	46	53	15.22%
13	50	53	52	53	46	46	53	15.22%
14	52	54	51	52	46	46	54	17.39%
15	50	54	53	53	48	48	54	12.50%
16	49	55	53	53	48	48	55	14.58%
17	51	54	53	52	48	48	54	12.50%
18	52	53	54	54	48	48	54	12.50%
19	51	53	53	54	48	48	54	12.50%
20	51	53	53	55	48	48	55	14.58%
21	51	52	53	55	48	48	55	14.58%
22	50	51	53	54	48	48	54	12.50%
23	50	52	53	53	48	48	53	10.42%
24	51	53	54	54	48	48	54	12.50%
25	50	53	55	53	48	48	55	14.58%
26	50	53	54	54	47	47	54	14.89%
27	49	54	54	53	48	48	54	12.50%
28	49	54	53	53	48	48	54	12.50%
29	49	54	53	53	49	49	54	10.20%
30	50	54	53	54	48	48	54	12.50%
31	50	54	52	53	48	48	54	12.50%
32	50	54	52	53	48	48	54	12.50%
33	51	54	53	53	49	49	54	10.20%
34	51	54	54	53	49	49	54	10.20%
35	51	55	53	52	49	49	55	12.24%
36	51	55	54	52	49	49	55	12.24%
37	51	54	53	51	48	48	54	12.50%
38	51	54	54	52	48	48	54	12.50%
39	51	55	54	52	48	48	55	14.58%
40	51	55	54	52	48	48	55	14.58%
41	50	55	54	51	48	48	55	14.58%
42	50	55	55	52	48	48	55	14.58%
43	50	55	55	52	48	48	55	14.58%
44	50	54	55	52	48	48	55	14.58%
45	50	54	55	52	48	48	55	14.58%
46	51	54	55	52	49	49	55	12.24%
47	51	54	55	52	49	49	55	12.24%
48	50	54	54	51	49	49	54	10.20%
49	50	54	55	51	49	49	55	12.24%
50	50	54	54	51	50	50	54	8.00%

E.5. Waiting area wait time

Table E.5: Convergence of the cumulative mean for the waiting area wait time in minutes per day

Day	Run 1	Run 2	Run 3	Run 4	Run 5	Min	Max	Convergence
1	22	14	11	42	27	11	42	281.82%
2	16	17	18	29	21	16	29	81.25%
3	13	18	26	25	18	13	26	100.00%
4	11	14	20	20	15	11	20	81.82%
5	12	13	17	16	12	12	17	41.67%
6	11	11	17	15	11	11	17	54.55%
7	10	12	17	13	13	10	17	70.00%
8	8	11	15	11	11	8	15	87.50%
9	9	9	13	11	15	9	15	66.67%
10	10	10	12	10	16	10	16	60.00%
11	11	10	11	9	14	9	14	55.56%
12	10	9	10	8	13	8	13	62.50%
13	14	11	13	9	12	9	14	55.56%
14	13	13	13	9	14	9	14	55.56%
15	14	14	14	10	14	10	14	40.00%
16	13	13	14	11	15	11	15	36.36%
17	13	13	14	11	15	11	15	36.36%
18	13	13	14	10	15	10	15	50.00%
19	12	12	13	9	15	9	15	66.67%
20	12	12	12	9	14	9	14	55.56%
21	11	11	13	9	13	9	13	44.44%
22	11	11	12	8	14	8	14	75.00%
23	10	10	12	8	13	8	13	62.50%
24	10	11	11	9	13	9	13	44.44%
25	10	11	11	9	14	9	14	55.56%
26	10	12	11	8	13	8	13	62.50%
27	11	11	11	8	14	8	14	75.00%
28	10	11	11	8	14	8	14	75.00%
29	10	11	11	8	13	8	13	62.50%
30	10	11	11	9	13	9	13	44.44%
31	10	11	11	9	14	9	14	55.56%
32	10	11	11	9	14	9	14	55.56%
33	10	11	11	9	13	9	13	44.44%
34	10	12	11	10	13	10	13	30.00%
35	10	11	10	9	13	9	13	44.44%
36	9	12	10	9	13	9	13	44.44%
37	10	12	10	9	14	9	14	55.56%
38	9	11	10	9	14	9	14	55.56%
39	9	11	9	9	13	9	13	44.44%
40	9	11	9	9	13	9	13	44.44%
41	9	11	9	9	13	9	13	44.44%
42	10	12	10	8	14	8	14	75.00%
43	10	12	10	9	14	9	14	55.56%
44	10	12	10	9	14	9	14	55.56%
45	10	12	10	9	14	9	14	55.56%
46	10	12	10	8	14	8	14	75.00%
47	10	12	10	8	14	8	14	75.00%
48	11	12	10	8	14	8	14	75.00%
49	10	12	10	8	13	8	13	62.50%
50	10	12	10	8	13	8	13	62.50%

F

Experimentation Output Data

This appendix presents all extensive simulation output results for the batch experiments described in Chapter 5.

F.1. Base performance

F.1.1. Route (a)

Table F.1: Output Data - Base Performance for Route PS (a)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	549	0	549	573	0	573
Marked	joints	(PS/SB)	81	351	432	327	221	549	364	209	573
Cleaned	joints	(PS/SB)	177	242	419	209	209	418	209	209	418
Sent to hold	joints	(Normal/Anode)	11	3	13	109	22	130	130	25	155
Sent to hold ratio	-	(Normal/Anode)	0.80	0.20	0.03	0.83	0.17	0.24	0.84	0.16	0.27
DJF production	joints	(PS/SB)	104	104	207	104	104	207	103	104	207
DJF rejects	joints	(Normal/Anode)	1.4	0.5	1.9	1.3	0.6	1.9	1.0	1.0	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	4.6	2.2	6.9
Main pipeline length	m				4886			4889			4880
PTC wait count	-	(PS/SB)	0	0	0	41	0	41	173	0	173
PTC wait time	min	(PS/SB)	0	0	0	69	0	69	291	0	291
DJF wait count	-	(PS/SB)	18	21	39	17	19	36	19	23	42
DJF wait time	min	(PS/SB)	21	32	53	19	27	46	22	34	56
Waiting area wait count	-	(Normal/Anode)	2	1	2	1	1	2	1	1	2
Waiting area wait time	min	(Normal/Anode)	5	6	11	4	8	11	4	8	12
Firing line wait count	-	(Normal/Anode)	7	3	10	7	4	11	7	5	11
Firing line wait time	min	(Normal/Anode)	20	11	30	14	15	29	15	17	32

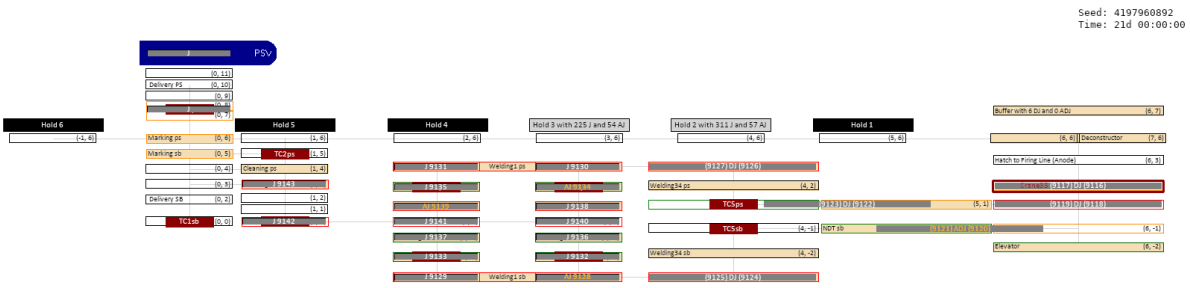


Figure F.1: Snapshot at end of simulation for Base Performance Route PS (a) with slow PTC rate

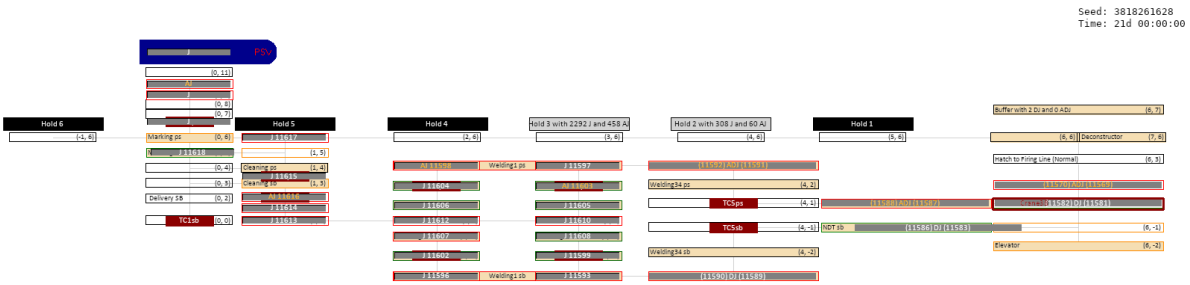


Figure F.2: Snapshot at end of simulation for Base Performance Route PS (a) with avg PTC rate

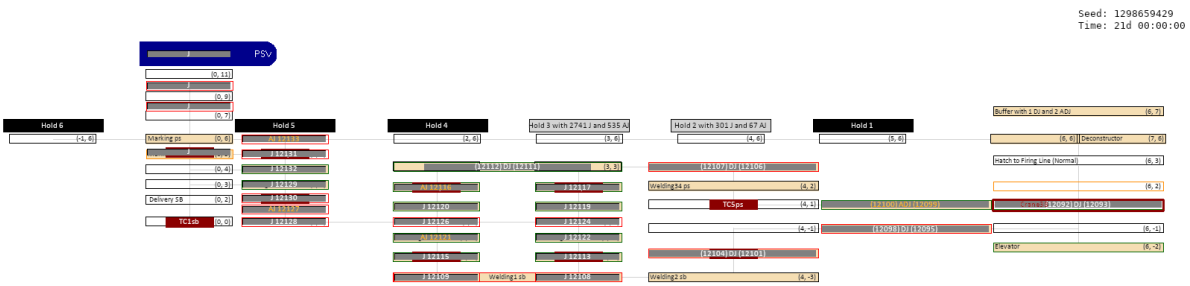


Figure F.3: Snapshot at end of simulation for Base Performance Route PS (a) with fast PTC rate

F.1.2. Route (b)

Table F.2: Output Data - Base Performance for Route PS (b)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	433	0	433	542	0	542	557	0	557
Marked	joints	(PS/SB)	87	346	433	320	221	542	348	209	557
Cleaned	joints	(PS/SB)	197	222	419	210	210	420	209	209	418
Sent to hold	joints	(Normal/Anode)	11	3	14	102	20	122	116	23	139
Sent to hold ratio	-	(Normal/Anode)	0.81	0.19	0.03	0.83	0.17	0.22	0.84	0.16	0.25
DJF production	joints	(PS/SB)	103	104	207	104	104	208	103	104	207
DJF rejects	joints	(Normal/Anode)	1.4	0.5	1.9	1.2	0.7	1.9	1.2	0.7	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	4.6	2.2	6.9
Main pipeline length	m				4892			4903			4888
PTC wait count	-	(PS/SB)	0	0	0	53	0	53	185	0	185
PTC wait time	min	(PS/SB)	0	0	0	90	0	90	323	0	323
DJF wait count	-	(PS/SB)	18	26	44	17	22	39	16	22	38
DJF wait time	min	(PS/SB)	23	40	63	19	32	51	17	29	46
Waiting area wait count	-	(Normal/Anode)	1	2	3	1	1	2	1	1	2
Waiting area wait time	min	(Normal/Anode)	4	7	11	3	7	10	4	4	9
Firing line wait count	-	(Normal/Anode)	6	4	10	7	4	11	7	3	10
Firing line wait time	min	(Normal/Anode)	13	16	30	16	14	29	16	10	26

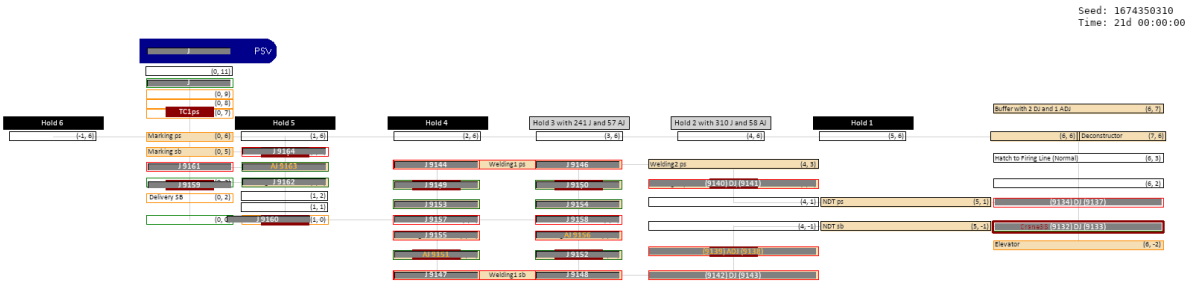


Figure F.4: Snapshot at end of simulation for Base Performance Route PS (b) with slow PTC rate

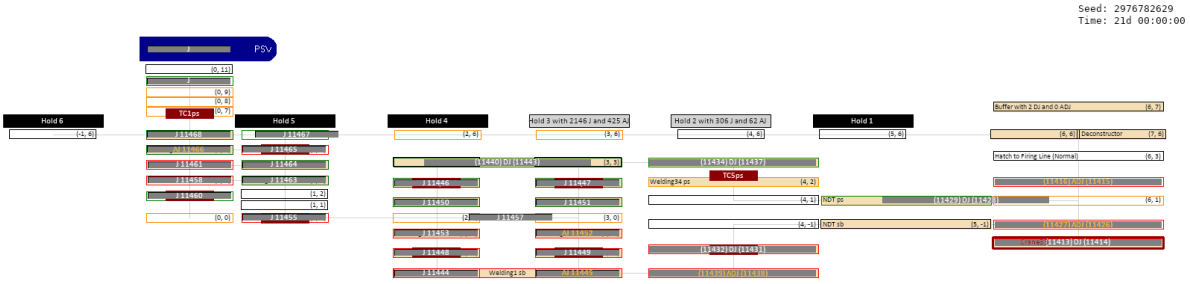


Figure F.5: Snapshot at end of simulation for Base Performance Route PS (b) with avg PTC rate

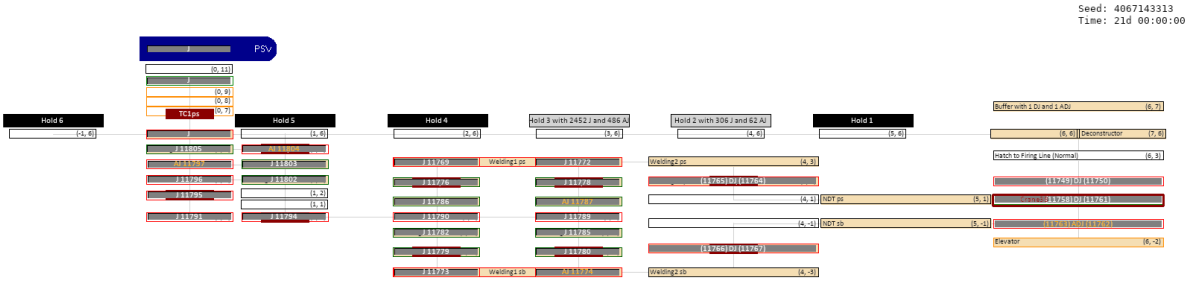


Figure F.6: Snapshot at end of simulation for Base Performance Route PS (b) with fast PTC rate

F.1.3. Route (c)

Table F.3: Output Data - Base Performance for Route PS (c)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	433	0	433	539	0	539	559	0	559
Marked	joints	(PS/SB)	81	352	433	309	230	539	345	214	559
Cleaned	joints	(PS/SB)	247	172	418	257	162	419	257	161	419
Sent to hold	joints	(Normal/Anode)	12	3	15	101	20	121	117	23	140
Sent to hold ratio	-	(Normal/Anode)	0.83	0.17	0.03	0.84	0.16	0.22	0.83	0.17	0.25
DJF production	joints	(PS/SB)	103	104	207	104	104	208	103	104	207
DJF rejects	joints	(Normal/Anode)	1.4	0.5	1.9	1.0	0.9	1.9	1.5	0.4	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	4.6	2.2	6.9
Main pipeline length	m				4880			4894			4886
PTC wait count	-	(PS/SB)	0	0	0	51	0	51	173	0	173
PTC wait time	min	(PS/SB)	0	0	0	91	0	91	324	0	324
DJF wait count	-	(PS/SB)	19	22	41	19	20	39	18	21	39
DJF wait time	min	(PS/SB)	23	33	56	21	27	48	21	30	50
Waiting area wait count	-	(Normal/Anode)	1	1	2	1	1	2	2	1	2
Waiting area wait time	min	(Normal/Anode)	5	8	12	4	4	9	8	3	11
Firing line wait count	-	(Normal/Anode)	5	4	9	7	4	10	8	3	11
Firing line wait time	min	(Normal/Anode)	14	17	31	16	11	27	23	7	30

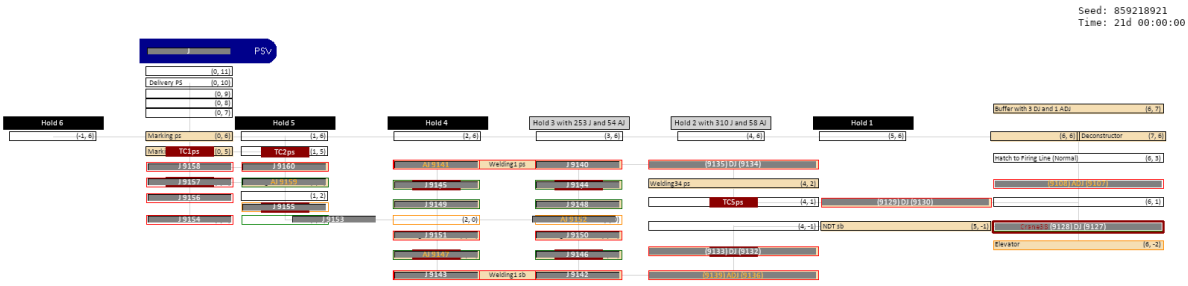


Figure F.7: Snapshot at end of simulation for Base Performance Route PS (c) with slow PTC rate

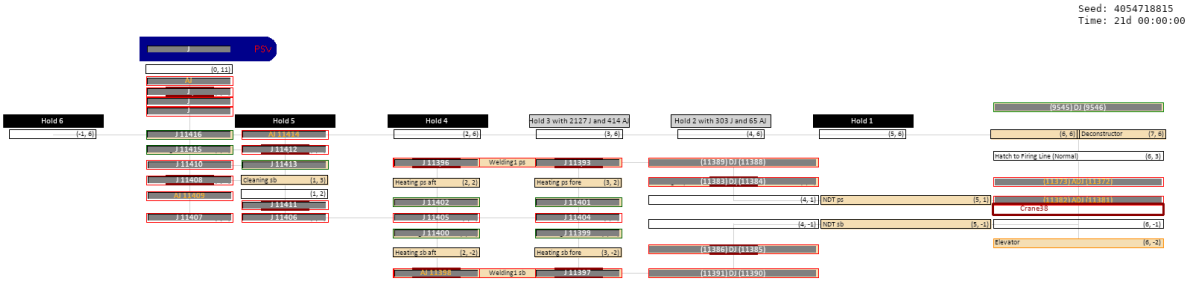


Figure F.8: Snapshot at end of simulation for Base Performance Route PS (c) with avg PTC rate

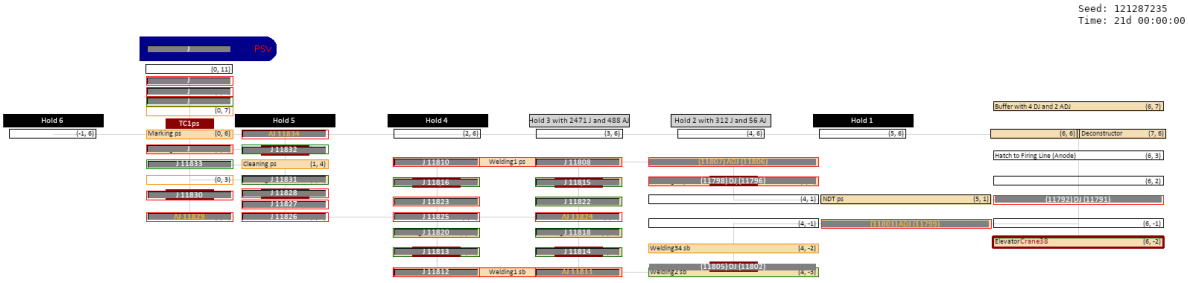


Figure F.9: Snapshot at end of simulation for Base Performance Route PS (c) with fast PTC rate

F.1.4. Route (d)

Table F.4: Output Data - Base Performance for Route PS (d)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	552	0	552	599	0	599
Marked	joints	(PS/SB)	65	367	432	310	242	552	377	222	599
Cleaned	joints	(PS/SB)	207	212	419	267	151	419	268	150	418
Sent to hold	joints	(Normal/Anode)	11	2	13	112	21	133	152	29	181
Sent to hold ratio	-	(Normal/Anode)	0.83	0.17	0.03	0.84	0.16	0.24	0.84	0.16	0.30
DJF production	joints	(PS/SB)	104	104	208	104	103	207	103	104	207
DJF rejects	joints	(Normal/Anode)	1.1	0.8	1.9	1.1	0.8	1.9	1.0	0.9	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	4.6	2.2	6.9
Main pipeline length	m				4888			4885			4880
PTC wait count	-	(PS/SB)	0	0	0	32	0	32	141	0	141
PTC wait time	min	(PS/SB)	1	0	1	62	0	62	240	0	240
DJF wait count	-	(PS/SB)	18	23	41	18	23	41	18	24	41
DJF wait time	min	(PS/SB)	20	34	54	20	32	52	21	33	54
Waiting area wait count	-	(Normal/Anode)	1	1	3	1	2	2	1	1	2
Waiting area wait time	min	(Normal/Anode)	4	7	11	3	10	13	3	13	16
Firing line wait count	-	(Normal/Anode)	7	4	11	6	4	10	7	4	11
Firing line wait time	min	(Normal/Anode)	16	16	32	11	18	29	13	22	34

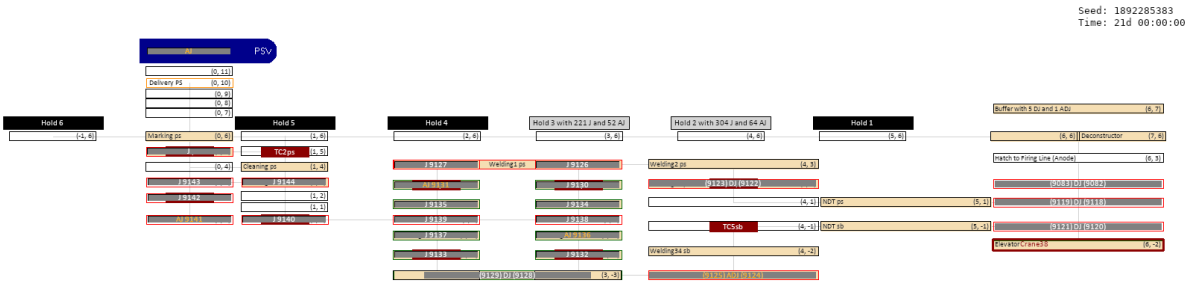


Figure F.10: Snapshot at end of simulation for Base Performance Route PS (d) with slow PTC rate

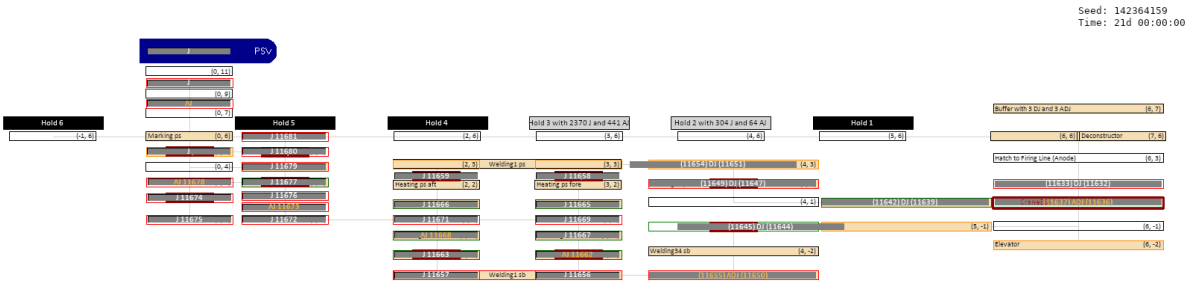


Figure F.11: Snapshot at end of simulation for Base Performance Route PS (d) with avg PTC rate

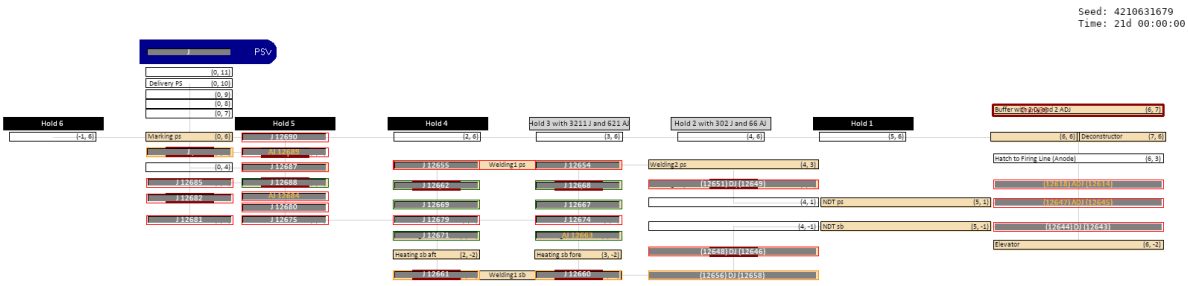


Figure F.12: Snapshot at end of simulation for Base Performance Route PS (d) with fast PTC rate

F.1.5. Route (e)

Table F.5: Output Data - Base Performance for Route PS (e)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	561	0	561	610	0	610
Marked	joints	(PS/SB)	183	249	432	351	209	561	401	209	610
Cleaned	joints	(PS/SB)	170	249	418	209	209	419	209	209	418
Sent to hold	joints	(Normal/Anode)	11	2	13	118	24	142	160	32	192
Sent to hold ratio	-	(Normal/Anode)	0.77	0.18	0.03	0.83	0.17	0.25	0.83	0.17	0.31
DJF production	joints	(PS/SB)	103	104	207	103	104	208	103	104	207
DJF rejects	joints	(Normal/Anode)	1.4	0.5	1.9	1.3	0.6	1.9	1.4	0.5	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	4.6	2.2	6.9
Main pipeline length	m				4889			4896			4880
PTC wait count	-	(PS/SB)	0	0	0	25	0	25	152	0	152
PTC wait time	min	(PS/SB)	1	0	1	38	0	38	221	0	221
DJF wait count	-	(PS/SB)	18	23	41	19	22	41	15	22	37
DJF wait time	min	(PS/SB)	21	32	53	23	32	54	18	30	48
Waiting area wait count	-	(Normal/Anode)	1	1	2	1	1	2	1	1	2
Waiting area wait time	min	(Normal/Anode)	5	5	10	2	9	11	3	10	13
Firing line wait count	-	(Normal/Anode)	7	4	11	5	4	10	6	5	10
Firing line wait time	min	(Normal/Anode)	18	13	30	10	18	28	13	19	32

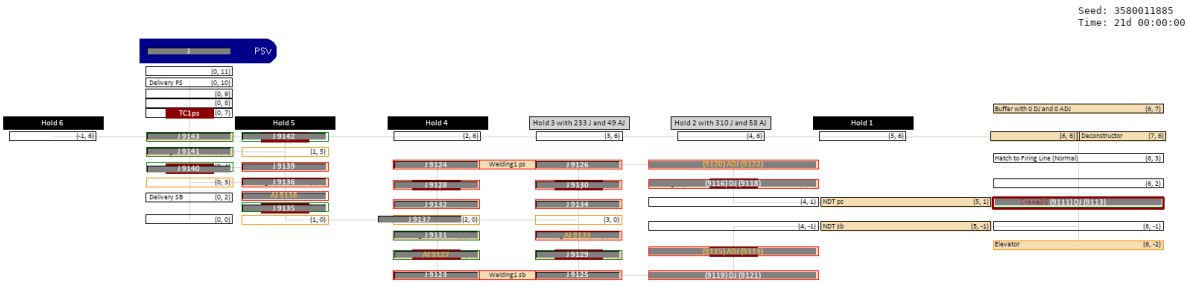


Figure F.13: Snapshot at end of simulation for Base Performance Route PS (e) with slow PTC rate

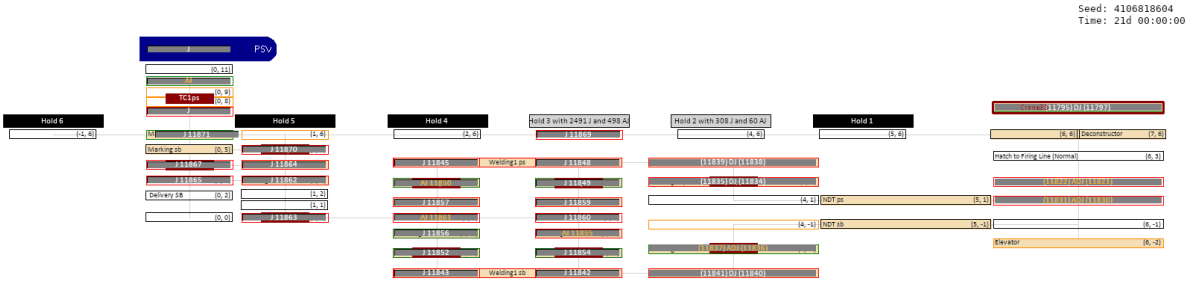


Figure F.14: Snapshot at end of simulation for Base Performance Route PS (e) with avg PTC rate

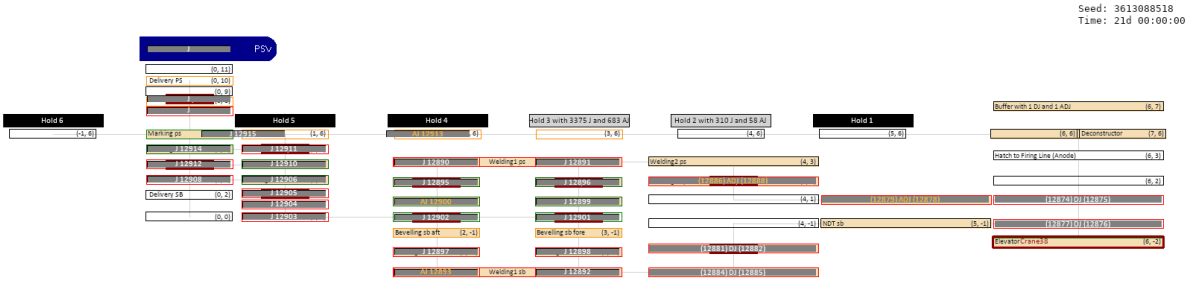


Figure F.15: Snapshot at end of simulation for Base Performance Route PS (e) with fast PTC rate

F.1.6. Route (f)

Table F.6: Output Data - Base Performance for Route PS (f)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	431	0	431	570	0	570	624	0	624
Marked	joints	(PS/SB)	32	399	431	253	317	570	389	237	626
Cleaned	joints	(PS/SB)	146	272	418	209	209	419	209	209	418
Sent to hold	joints	(Normal/Anode)	11	2	13	125	26	151	173	36	209
Sent to hold ratio	-	(Normal/Anode)	0.79	0.21	0.03	0.83	0.17	0.27	0.83	0.17	0.33
DJF production	joints	(PS/SB)	104	103	207	104	104	208	104	105	209
DJF rejects	joints	(Normal/Anode)	1.3	0.6	1.9	1.4	0.5	1.9	0.0	1.0	1.0
Firing line rejects	joints	(Normal/Anode)	4.6	2.2	6.9	4.6	2.2	6.9	8.0	4.0	12.0
Main pipeline length	m				4880			4893			4733.6
PTC wait count	-	(PS/SB)	0	0	0	13	0	13	124	0	124
PTC wait time	min	(PS/SB)	0	0	0	20	0	20	190	0	190
DJF wait count	-	(PS/SB)	20	25	45	16	23	39	11	23	34
DJF wait time	min	(PS/SB)	24	37	61	18	32	50	9	30	39
Waiting area wait count	-	(Normal/Anode)	1	2	2	1	1	2	0	2	2
Waiting area wait time	min	(Normal/Anode)	4	10	14	4	6	10	0	19	19
Firing line wait count	-	(Normal/Anode)	6	4	10	6	4	10	4	7	11
Firing line wait time	min	(Normal/Anode)	13	19	32	13	13	26	3	32	35

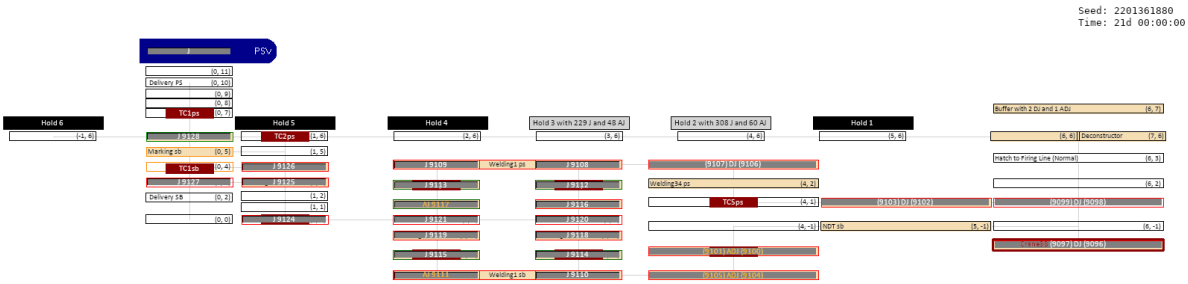


Figure F.16: Snapshot at end of simulation for Base Performance Route PS (f) with slow PTC rate

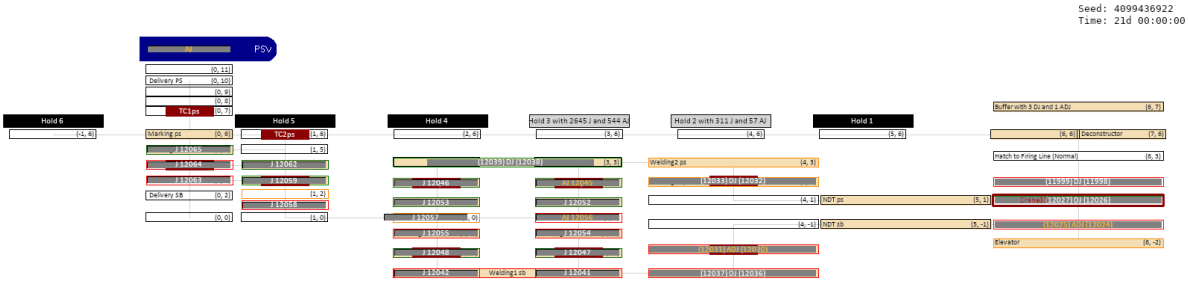


Figure F.17: Snapshot at end of simulation for Base Performance Route PS (f) with avg PTC rate

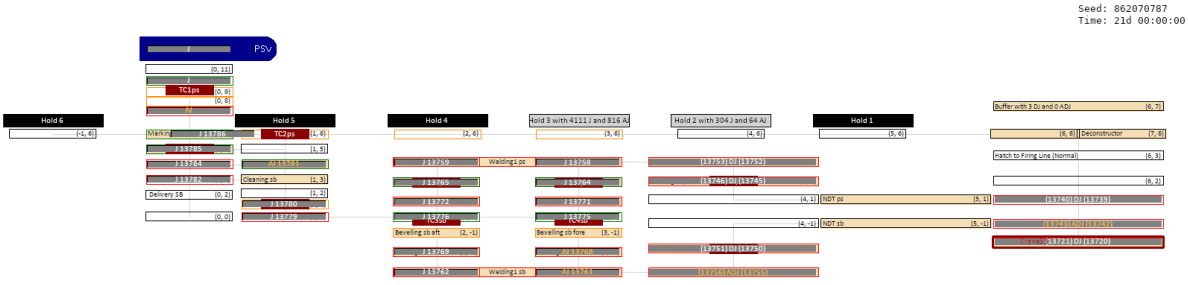


Figure F.18: Snapshot at end of simulation for Base Performance Route PS (f) with fast PTC rate

F.2. Conclusive batch experimentation

F.2.1. Route (a)

Table F.7: Output Data - Final Performance for Route PS (a)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	577	0	577	716	0	716
Marked	joints	(PS/SB)	12	420	432	172	405	577	329	387	716
Cleaned	joints	(PS/SB)	96	324	419	209	210	419	209	209	419
Sent to hold	joints	(Normal/Anode)	10	3	13	131	27	158	247	50	297
Sent to hold ratio	-	(Normal/Anode)	0.66	0.34	0.03	0.83	0.17	0.27	0.83	0.17	0.42
DJF production	joints	(PS/SB)	104	104	208	104	104	208	104	104	207
DJF rejects	joints	(Normal/Anode)	1.3	0.6	1.9	1.4	0.5	1.9	1.3	0.6	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4896			4894			4890
PTC wait count	-	(PS/SB)	0	0	0	0	0	0	11	0	11
PTC wait time	min	(PS/SB)	0	0	0	0	0	0	7	0	7
DJF wait count	-	(PS/SB)	23	25	48	13	18	31	13	17	30
DJF wait time	min	(PS/SB)	41	55	96	12	21	33	12	20	32
Waiting area wait count	-	(Normal/Anode)	2	0	2	2	0	2	2	0	2
Waiting area wait time	min	(Normal/Anode)	7	1	7	7	0	7	8	0	8
Firing line wait count	-	(Normal/Anode)	9	2	11	9	2	11	10	2	12
Firing line wait time	min	(Normal/Anode)	24	3	27	23	2	25	28	1	29

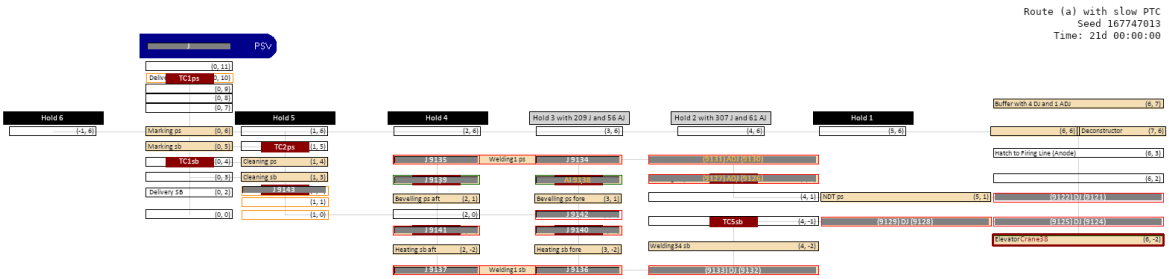


Figure F.19: Snapshot at end of simulation for Route PS (a) with slow PTC rate

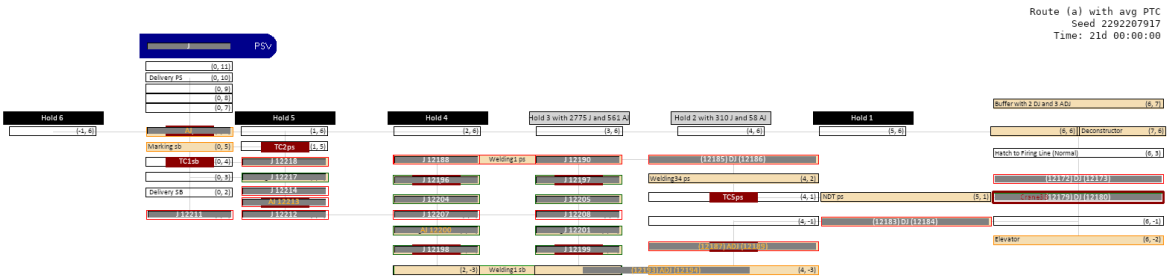


Figure F.20: Snapshot at end of simulation for Route PS (a) with avg PTC rate

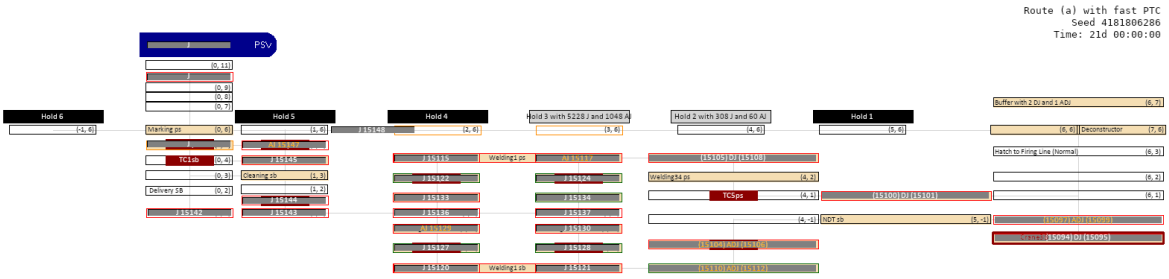


Figure F.21: Snapshot at end of simulation for Route PS (a) with fast PTC rate

F.2.2. Route (b)

Table F.8: Output Data - Final Performance for Route PS (b)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	430	0	430	575	0	575	711	0	711
Marked	joints	(PS/SB)	10	420	430	177	397	575	346	365	711
Cleaned	joints	(PS/SB)	119	300	419	209	209	418	208	208	417
Sent to hold	joints	(Normal/Anode)	10	2	11	130	26	156	245	49	294
Sent to hold ratio	-	(Normal/Anode)	0.83	0.17	0.03	0.83	0.17	0.27	0.83	0.17	0.41
DJF production	joints	(PS/SB)	104	103	207	104	104	207	103	103	207
DJF rejects	joints	(Normal/Anode)	1.1	0.8	1.9	1.0	0.9	1.9	1.3	0.6	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4883			4886			4868
PTC wait count	-	(PS/SB)	0	0	0	1	0	1	20	0	20
PTC wait time	min	(PS/SB)	0	0	0	0	0	0	19	0	19
DJF wait count	-	(PS/SB)	23	25	47	13	20	34	15	18	32
DJF wait time	min	(PS/SB)	36	45	81	12	23	35	12	20	32
Waiting area wait count	-	(Normal/Anode)	1	1	3	2	0	3	3	0	3
Waiting area wait time	min	(Normal/Anode)	6	7	13	9	1	10	9	0	9
Firing line wait count	-	(Normal/Anode)	6	4	10	9	2	11	12	1	14
Firing line wait time	min	(Normal/Anode)	17	14	31	27	3	30	34	1	36

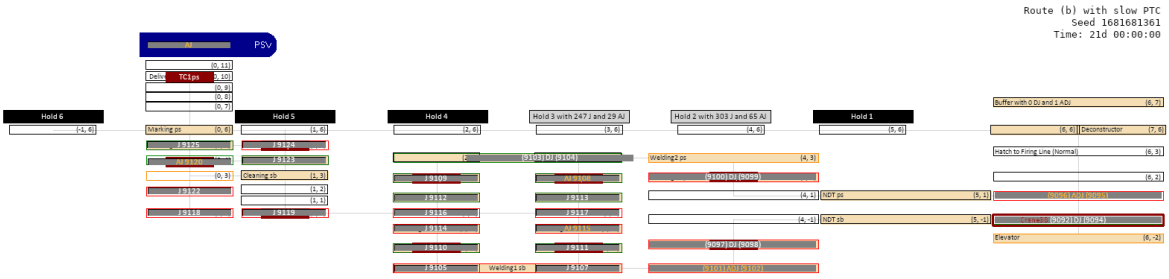


Figure F.22: Snapshot at end of simulation for Route PS (b) with slow PTC rate

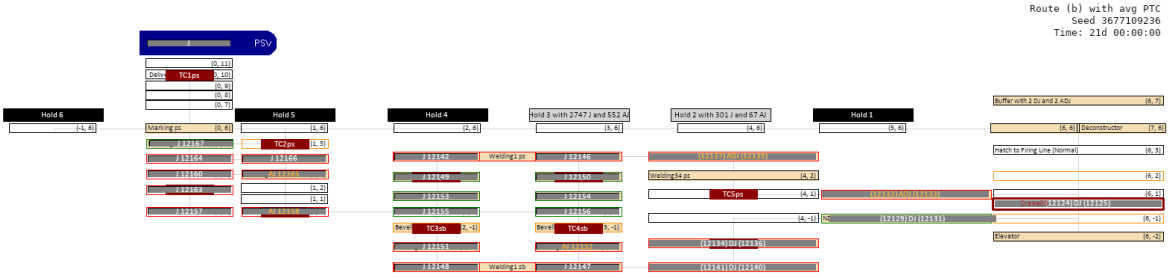


Figure F.23: Snapshot at end of simulation for Route PS (b) with avg PTC rate

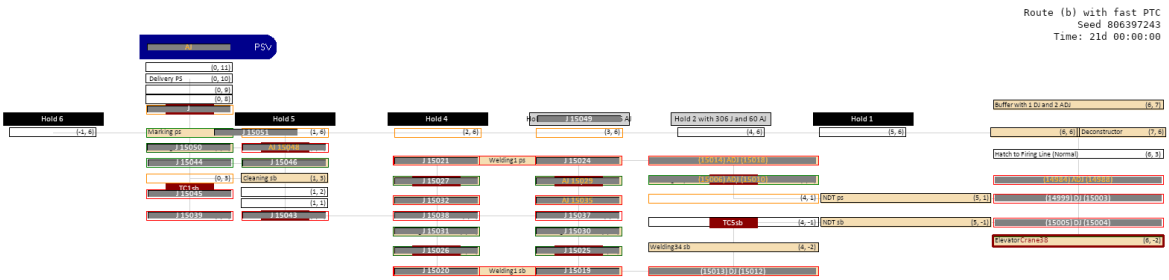


Figure F.24: Snapshot at end of simulation for Route PS (b) with fast PTC rate

F.2.3. Route (c)

Table F.9: Output Data - Final Performance for Route PS (c)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	577	0	577	713	0	713
Marked	joints	(PS/SB)	10	422	432	180	397	577	343	370	713
Cleaned	joints	(PS/SB)	184	236	420	209	209	417	209	209	418
Sent to hold	joints	(Normal/Anode)	10	2	12	134	26	160	246	49	295
Sent to hold ratio	-	(Normal/Anode)	0.83	0.17	0.03	0.84	0.16	0.28	0.83	0.17	0.41
DJF production	joints	(PS/SB)	104	104	208	104	103	207	103	104	207
DJF rejects	joints	(Normal/Anode)	1.2	0.7	1.9	1.2	0.7	1.9	1.1	0.8	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4901			4873			4881
PTC wait count	-	(PS/SB)	0	0	0	1	0	1	20	0	20
PTC wait time	min	(PS/SB)	0	0	0	1	0	1	15	0	15
DJF wait count	-	(PS/SB)	18	22	39	14	19	33	15	19	34
DJF wait time	min	(PS/SB)	20	33	53	13	22	35	13	21	34
Waiting area wait count	-	(Normal/Anode)	1	1	3	2	0	2	3	0	3
Waiting area wait time	min	(Normal/Anode)	4	5	10	9	0	9	10	0	10
Firing line wait count	-	(Normal/Anode)	8	3	11	10	2	12	10	2	12
Firing line wait time	min	(Normal/Anode)	18	11	28	31	2	33	30	2	32

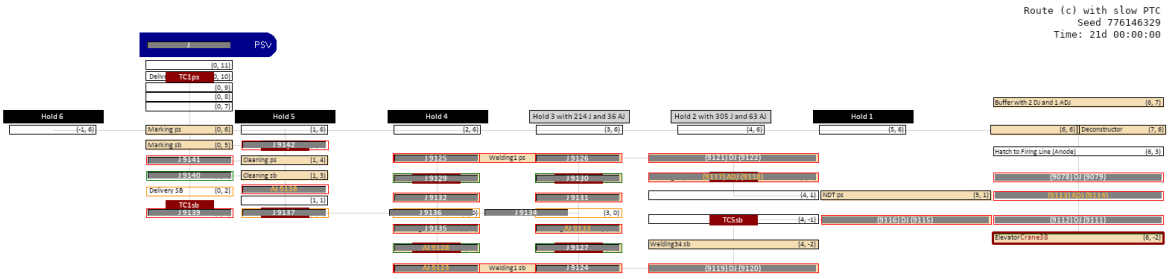


Figure F.25: Snapshot at end of simulation for Route PS (c) with slow PTC rate

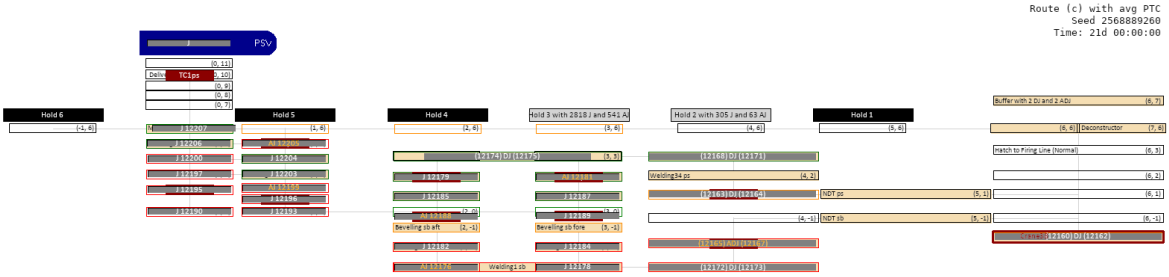


Figure F.26: Snapshot at end of simulation for Route PS (c) with avg PTC rate

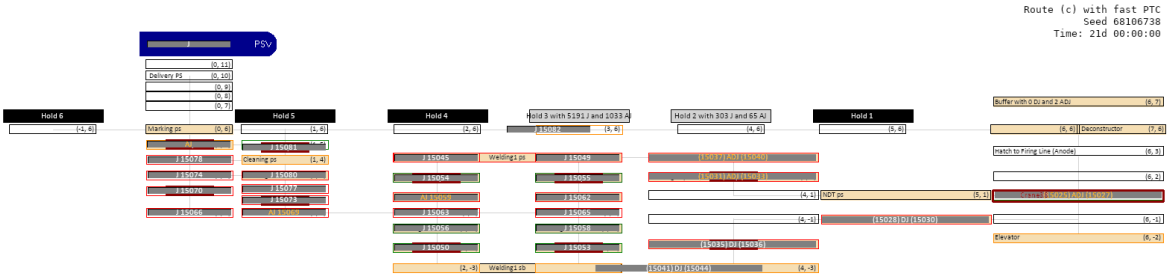


Figure F.27: Snapshot at end of simulation for Route PS (c) with fast PTC rate

F.2.4. Route (d)

Table F.10: Output Data - Final Performance for Route PS (d)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	576	0	576	713	0	713
Marked	joints	(PS/SB)	12	421	432	173	403	576	328	385	713
Cleaned	joints	(PS/SB)	111	308	419	249	171	419	263	156	420
Sent to hold	joints	(Normal/Anode)	11	2	13	131	26	157	244	49	293
Sent to hold ratio	-	(Normal/Anode)	0.82	0.18	0.03	0.84	0.16	0.27	0.83	0.17	0.41
DJF production	joints	(PS/SB)	104	103	208	104	104	208	104	104	208
DJF rejects	joints	(Normal/Anode)	1.5	0.4	1.9	1.5	0.4	1.9	1.2	0.7	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4899			4899			4901
PTC wait count	-	(PS/SB)	0	0	0	1	0	1	22	0	22
PTC wait time	min	(PS/SB)	0	0	0	1	0	1	16	0	16
DJF wait count	-	(PS/SB)	25	27	52	18	22	40	18	23	40
DJF wait time	min	(PS/SB)	39	53	92	21	31	52	20	31	51
Waiting area wait count	-	(Normal/Anode)	2	0	2	2	0	2	1	0	1
Waiting area wait time	min	(Normal/Anode)	6	2	7	9	1	10	6	1	6
Firing line wait count	-	(Normal/Anode)	8	3	11	10	2	12	8	2	10
Firing line wait time	min	(Normal/Anode)	23	5	28	27	3	30	21	3	24

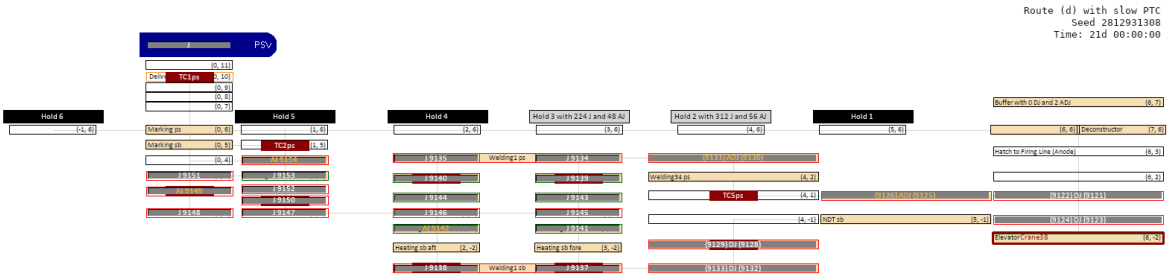


Figure F.28: Snapshot at end of simulation for Route PS (d) with slow PTC rate

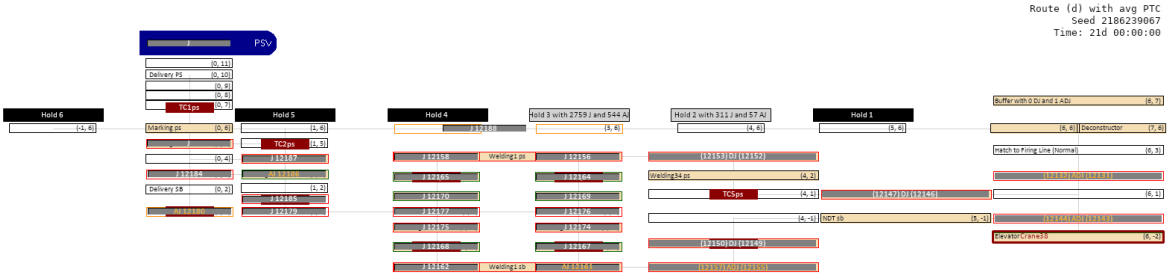


Figure F.29: Snapshot at end of simulation for Route PS (d) with avg PTC rate

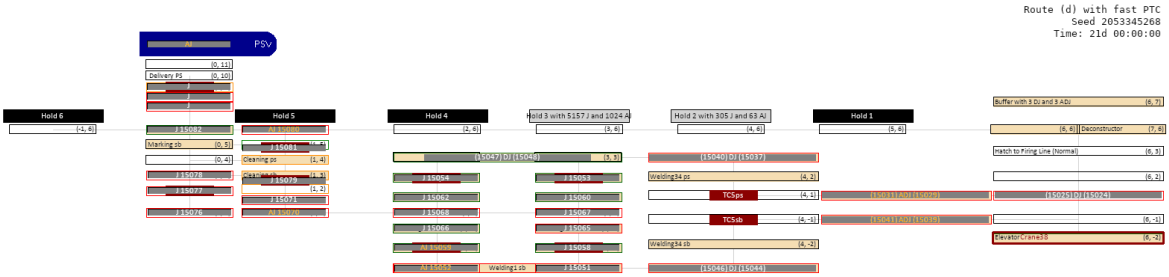


Figure F.30: Snapshot at end of simulation for Route PS (d) with fast PTC rate

F.2.5. Route (e)

Table F.11: Output Data - Final Performance for Route PS (e)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	433	0	433	577	0	577	710	0	710
Marked	joints	(PS/SB)	134	299	433	180	397	577	492	218	710
Cleaned	joints	(PS/SB)	122	297	419	209	209	417	209	209	418
Sent to hold	joints	(Normal/Anode)	11	2	14	134	26	160	243	48	292
Sent to hold ratio	-	(Normal/Anode)	0.82	0.18	0.03	0.84	0.16	0.28	0.83	0.17	0.41
DJF production	joints	(PS/SB)	104	104	208	104	103	207	104	104	207
DJF rejects	joints	(Normal/Anode)	1.3	0.6	1.9	1.2	0.7	1.9	1.3	0.6	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4897			4873			4887
PTC wait count	-	(PS/SB)	0	0	0	1	0	1	23	0	23
PTC wait time	min	(PS/SB)	0	0	0	1	0	1	18	0	18
DJF wait count	-	(PS/SB)	15	19	35	14	19	33	15	17	32
DJF wait time	min	(PS/SB)	17	28	45	13	22	35	14	20	34
Waiting area wait count	-	(Normal/Anode)	2	0	2	2	0	2	2	0	2
Waiting area wait time	min	(Normal/Anode)	8	1	9	9	0	9	9	0	9
Firing line wait count	-	(Normal/Anode)	8	2	10	10	2	12	11	2	13
Firing line wait time	min	(Normal/Anode)	23	5	28	31	2	33	29	1	31

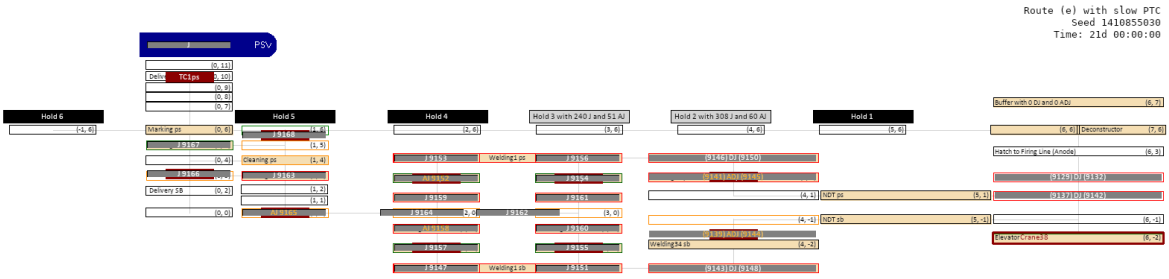


Figure F.31: Snapshot at end of simulation for Route PS (e) with slow PTC rate

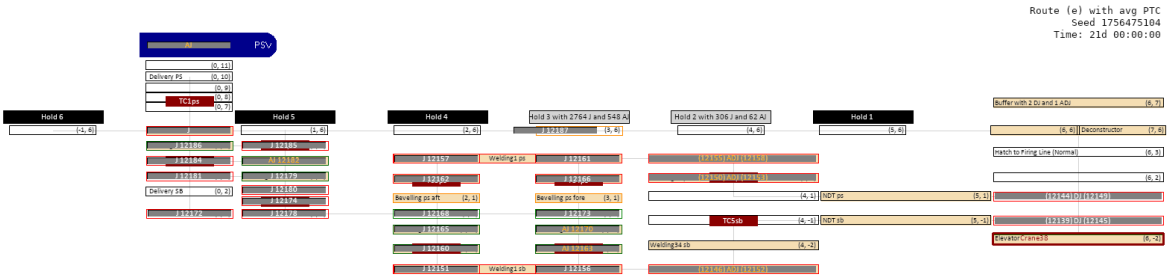


Figure F.32: Snapshot at end of simulation for Route PS (e) with avg PTC rate

Figure F.33: Snapshot at end of simulation for Route PS (e) with fast PTC rate

F.2.6. Route (f)

Table F.12: Output Data - Final Performance for Route PS (f)

			slow PTC rate (18 SJ/h)			avg PTC rate (24 SJ/h)			fast PTC rate (30 SJ/h)		
Data	**Unit**	**(1/2)**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**	**(1)**	**(2)**	**Total**
PTC delivery	joints	(PS/SB)	432	0	432	576	0	576	722	0	722
Marked	joints	(PS/SB)	0	431	432	16	560	576	44	677	722
Cleaned	joints	(PS/SB)	97	322	419	207	212	419	209	209	419
Sent to hold	joints	(Normal/Anode)	11	2	13	131	26	157	252	51	303
Sent to hold ratio	-	(Normal/Anode)	0.84	0.16	0.03	0.83	0.17	0.27	0.83	0.17	0.42
DJF production	joints	(PS/SB)	104	104	207	104	103	207	104	104	208
DJF rejects	joints	(Normal/Anode)	1.1	0.8	1.9	1.2	0.7	1.9	1.5	0.4	1.9
Firing line rejects	joints	(Normal/Anode)	4.6	2.3	6.9	4.6	2.3	6.9	4.6	2.3	6.9
Main pipeline length	m				4889			4890			4893
PTC wait count	-	(PS/SB)	0	0	0	0	0	0	0	0	0
PTC wait time	min	(PS/SB)	0	0	0	0	0	0	0	0	0
DJF wait count	-	(PS/SB)	15	19	35	15	18	33	14	18	32
DJF wait time	min	(PS/SB)	16	24	40	14	21	34	13	20	33
Waiting area wait count	-	(Normal/Anode)	1	1	2	2	0	2	2	0	2
Waiting area wait time	min	(Normal/Anode)	3	6	9	8	0	8	8	0	8
Firing line wait count	-	(Normal/Anode)	7	3	9	11	2	13	9	2	11
Firing line wait time	min	(Normal/Anode)	16	10	27	31	1	32	26	1	27

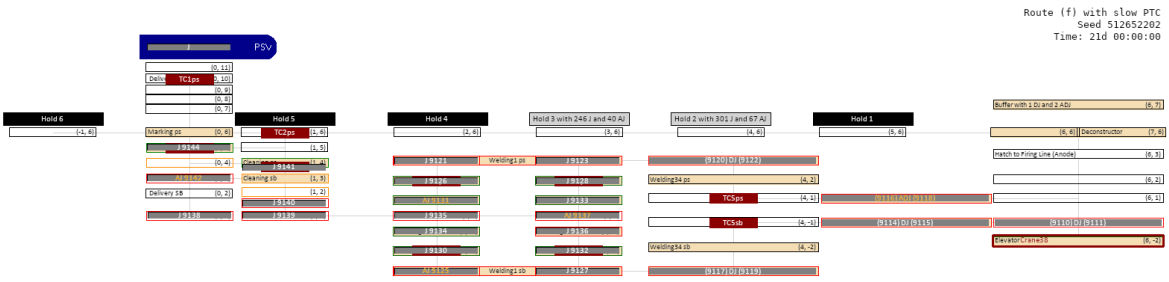


Figure F.34: Snapshot at end of simulation for Route PS (f) with slow PTC rate

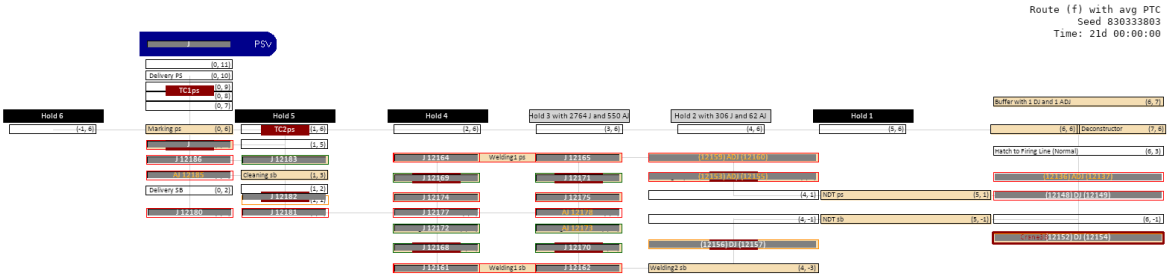


Figure F.35: Snapshot at end of simulation for Route PS (f) with avg PTC rate

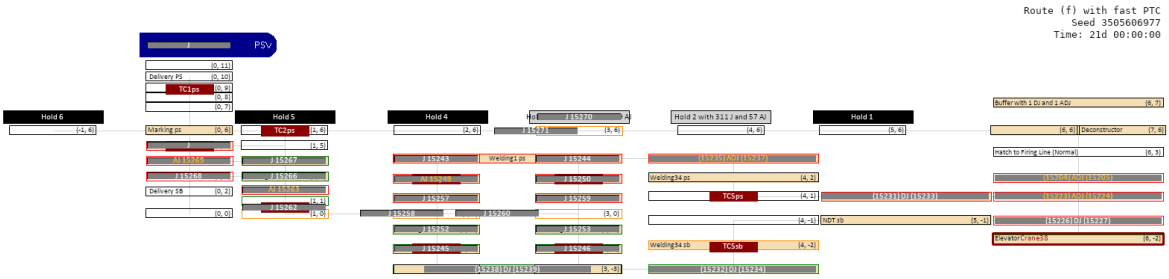


Figure F.36: Snapshot at end of simulation for Route PS (f) with fast PTC rate

G

Python Code

This appendix presents the Python code for the simulation model with implementation of the final routing strategies applied for the conclusive batch experiments presented in Section 5.6. The Python code consists of four scripts: parameters, map, main and a run script. The parameters script loads all input parameters defined in an Excel sheet. The map script generates the environment of the simulation model including all positional coordinates and loads the move map of the routing strategy that is defined in another Excel sheet. The main script generates the simulation components and applies the parameters script and map script to run the simulation. The run script is used to define the pipe supply scenario and route configuration, and run the main script accordingly.

G.1. Parameters

Listing G.1: Parameters Python Script

```python
import pandas as pd
import random

import os
psv = os.environ.get('PSV', 'PS')
ptc_rate = os.environ.get('PTC_RATE', 'avg')
live_animation = os.environ.get('LIVE_ANIMATION', 'False') == 'True'
video = os.environ.get('VIDEO', 'False') == 'True'

scale = 7  # Scale for the animation
stepsize = 0.02 # Step size for the animation

# Load datafiles
parameters_df = pd.read_excel('input/parameters.xlsx', sheet_name=f'Parameters {psv} {ptc_rate} PTC')
parameters = {}
for _, row in parameters_df.iterrows():
    component = row["Component"]
    variable = row["Variable"]
    value = row["Value"]
    if component not in parameters:
        parameters[component] = {}
    parameters[component][variable] = value

if live_animation or video:
    parameters['PSVps']['joint_amount'] = 200

#Joint parameters
joint_length = parameters["Joint"]["length"] * scale
joint_width = parameters["Joint"]["diameter"] * scale

#LCO parameters
LCO_v = parameters["LCO"]["velocity"]
LCO_a_time = parameters["LCO"]["acceleration_time"]
LCO_a_distance = parameters["LCO"]["acceleration_distance"]

#TC parameters
tc_length = parameters["TC"]["length"] * scale
tc_width = parameters["TC"]["width"] * scale
TC_v = parameters["TC"]["velocity"]
TC_a_time = parameters["TC"]["acceleration_time"]
TC_a_distance = parameters["TC"]["acceleration_distance"]

#Crane parameters
crane_length = parameters["Crane38"]["length"] * scale
crane_width = parameters["Crane38"]["width"] * scale
crane_v = parameters["Crane38"]["velocity"]
crane_a_time = parameters["Crane38"]["acceleration_time"]
crane_a_distance = parameters["Crane38"]["acceleration_distance"]

#Empirical station duration parameters
SEE_data = pd.read_excel('Input/parameters.xlsx', sheet_name='SEE data')

def sample_duration(station):
    data = SEE_data[station].dropna()
    data = data[data < data.quantile(0.8)]
    sample = random.choice(data)
    return sample
```

G.2. Map

Listing G.2: Map Python Script

```python
import salabim as sim
import pandas as pd
```

```
from input.parameters import *

import os
psv = os.environ.get('PSV', 'PS')
ptc_rate = os.environ.get('PTC_RATE', 'avg')
route = os.environ.get('ROUTE', 'a')
djf_supply_line = os.environ.get('DJF_SUPPLY_LINE', 'False') == 'True'

position_map_df = pd.read_excel('input/map.xlsx', sheet_name='Position Map')
position_map = {
    eval(row["Position"]) if isinstance(row["Position"], str) and row["Position"].startswith("(") else row["Position"]:
    (row["X"], row["Y"])
    for _, row in position_map_df.iterrows()
}

pos = {
    row["Name"]: row["Position"] if isinstance(row["Position"], tuple) else eval(row["Position"])
    for _, row in position_map_df.iterrows()
    if pd.notna(row["Name"]) and row["Name"] != ""
}

posRouting = [eval(row["Position"]) for _, row in position_map_df.iterrows() if row["posRouting"]]
posSingleJoints = [eval(row["Position"]) for _, row in position_map_df.iterrows() if row["posSingleJoint"]]
posDoubleJoints = [eval(row["Position"]) for _, row in position_map_df.iterrows() if row["posDoubleJoint"]]
posMarking = [pos["Marking_sb"], pos["Marking_ps"]]
posCleaning = [pos["Cleaning_sb"], pos["Cleaning_ps"]]
posBevelling = [pos["Bevelling_sb_aft"], pos["Bevelling_sb_fore"], pos["Bevelling_ps_aft"], pos["Bevelling_ps_fore"]]
posHeating = [pos["Heating_sb_aft"], pos["Heating_sb_fore"], pos["Heating_ps_aft"], pos["Heating_ps_fore"]]
posWelding1 = [pos["Welding1_sb_aft"], pos["Welding1_sb_fore"], pos["Welding1_ps_aft"], pos["Welding1_ps_fore"]]
posWelding2 = [pos["Welding2_sb"], pos["Welding2_ps"]]
posWelding34 = [pos["Welding34_sb"], pos["Welding34_ps"]]
posNDT = [pos["NDT_sb"], pos["NDT_ps"]]
posWaitingArea = [pos["waiting_sb"], pos["waiting_ps"]]
posProcessingStations = [pos for sublist in [posMarking, posCleaning, posBevelling, posHeating, posWelding1, posWelding2, posWelding34, posNDT] for pos in sublist]


if psv == 'PS+SB':
    move_sheet = 'Move Map PS+SB'
elif psv == 'PS':
    move_sheet = f'Move Map PS ({route})'
elif psv == 'SB':
    move_sheet = 'Move Map SB'

move_map_df = pd.read_excel('input/map.xlsx', sheet_name=move_sheet)

move_map_marking = {
    eval(row["Position"]): (eval(row["Marking"]), None if row["Marking Resource"] == "LCO" else row["Marking Resource"])
    for _, row in move_map_df.iterrows()
    if pd.notna(row["Marking"])
}

move_map_hold = {
    eval(row["Position"]): (eval(row["Hold"]), None if row["Hold Resource"] == "LCO" else row["Hold Resource"])
    for _, row in move_map_df.iterrows()
    if pd.notna(row["Hold"])
}

move_map_cleaning = {
    eval(row["Position"]): (eval(row["Cleaning"]), None if row["Cleaning Resource"] == "LCO" else row["Cleaning Resource"])
```

```
    for _, row in move_map_df.iterrows()
    if pd.notna(row["Cleaning"])
}

move_map_djf = {
    eval(row["Position"]): (eval(row["DJF"]), None if row["DJF Resource"] == "LCO" else row["
        DJF Resource"])
    for _, row in move_map_df.iterrows()
    if pd.notna(row["DJF"])
}
if djf_supply_line:
    move_map_djf[(0, 0)] = None, None
    move_map_djf[(1, 0)] = None, None


env = sim.Environment()
occupancy_map = {
    position: sim.State(name=str(position), value=0)
    for position in position_map
    if not isinstance(position, str)
}

background = {}

def create_background():
    for position in posSingleJoints:
        x, y = position_map[position]
        rectangle = sim.AnimateRectangle(
            spec=((-0.52*joint_length), (-0.7*joint_width), (0.52*joint_length), (0.7*
                joint_width)),
            fillcolor="",
            linewidth=0.5,
            linecolor="black",
            textcolor="black",
            text=f"{position}",
            text_anchor="e",
            fontsize=1.2*joint_width,
            layer=10,
            x=x * scale,
            y=y * scale,
        )
        background[position] = rectangle

    for position in posDoubleJoints:
        x, y = position_map[position]
        rectangle = sim.AnimateRectangle(
            spec=((-1.02*joint_length), (-0.7*joint_width), (1.02*joint_length), (0.7*
                joint_width)),
            fillcolor="",
            linewidth=0.5,
            linecolor="black",
            textcolor="black",
            text=f"{position}",
            text_anchor="e",
            fontsize=1.2*joint_width,
            layer=11,
            x=x * scale,
            y=y * scale,
        )

        if position in [(2.5, -3), (2.5, 3), (6.5, 6)]:
            rectangle.text = None
            if position in [(2.5, -3), (2.5, 3)]:
                rectangle.spec = ((-1.37*joint_length), (-0.7*joint_width), (1.37*
                    joint_length), (0.7*joint_width))
            elif position == (6.5, 6):
                rectangle.spec = ((-1.05*joint_length), (-0.7*joint_width), (1.05*
                    joint_length), (0.7*joint_width))
        background[position] = rectangle


```

```
128
129     line_positions = [
130         ((0, 0), (0, 10)),
131         ((1, 0), (1, 6)),
132         ((0, 3), (1, 3)),
133         ((0, 4), (1, 4)),
134         ((0, 5), (1, 5)),
135         ((-1, 6), (7, 6)),
136         ((1, 0), (3, 0)),
137         ((2, -3), (2, 3)),
138         ((3, -3), (3, 3)),
139         ((2.5, -3), (4, -3)),
140         ((2.5, 3), (4, 3)),
141         ((4, -3), (4, -1)),
142         ((4, 3), (4, 1)),
143         ((4, -1), (6, -1)),
144         ((4, 1), (6, 1)),
145         ((6,-2), (6.5,6))
146     ]
147
148     for pos1, pos2 in line_positions:
149         x1, y1 = position_map[pos1]
150         x2, y2 = position_map[pos2]
151         sim.AnimateLine(
152             spec=(x1 * scale, y1 * scale, x2 * scale, y2 * scale),
153             linecolor="lightgray",
154             linewidth=1,
155             layer=90,
156         )
```

G.3. Main

Listing G.3: Main Python Script

```
1  import salabim as sim
2  import random
3  import numpy as np
4  import time
5  from input.parameters import *
6  from input.map import *
7  from statistics import mean
8  import openpyxl
9
10 import os
11 psv_side = os.environ.get('PSV')
12 route = os.environ.get('ROUTE')
13 scenario = f'Route ({route})'
14 ptc_rate = os.environ.get('PTC_RATE')
15 seed = int(os.environ.get('SEED'))
16
17 # region Simulation Settings
18
19 title = f"{scenario} with {ptc_rate} PTC"
20 warm_up_period = 4 * 3600
21 period_duration = 24 * 3600
22 periods = 21
23 simulation_duration = periods * period_duration + warm_up_period
24 animation = os.environ.get('ANIMATION', 'False') == 'True'
25 blind_animation = True
26 animation_speed = 100
27
28 # end region
29
30 random.seed(seed)
31 rng_djf = random.Random(44)
32 rng_firingline = random.Random(69)
33
34
35 # region Components
```

```
class PSV(sim.Component):
    def setup(self, position, joint_amount, anode_ratio):
        self.position = position
        self.joint_count = int(joint_amount)
        self.anode_ratio = anode_ratio
        self.queue = []

        self.x, self.y = position_map[self.position]
        psv_length = 1.5 * joint_length
        psv_width = 4 * joint_width
        self.rectangle = sim.AnimateRectangle(
            spec=(-0.5*psv_length, -0.5*psv_width,
                  0.5*psv_length, 0.5*psv_width),
            fillcolor='darkblue',
            linewidth=0,
            layer=100,
            x=(self.x * scale) + 0.1*psv_length,
            y=self.y * scale,
        )
        self.circle = sim.AnimateCircle(
            radius=(0.5*psv_width),
            fillcolor='darkblue',
            linewidth=0,
            text='PSV',
            fontsize=1.8*joint_width,
            text_anchor='e',
            layer=100,
            x=(self.x * scale) + 0.6*psv_length,
            y=self.y * scale,
        )


    def process(self):
        anode_count = int(self.joint_count * self.anode_ratio)
        regular_count = self.joint_count - anode_count

        regular_joints = [
            Joint(
                name=f'Joint',
                position=self.position,
                anode=False,
            )
            for i in range(regular_count)
        ]
        anode_joints = [
            Joint(
                name=f'AnodeJoint',
                position=self.position,
                anode=True,
            )
            for i in range(anode_count)
        ]

        anode_interval = int(1/self.anode_ratio)
        regulars = [anode_interval-2, anode_interval-1, anode_interval]
        while regular_joints or anode_joints:
            if anode_joints:
                aj = anode_joints.pop(0)
                self.queue.append(aj)
            for i in range(random.choice(regulars)):
                if regular_joints:
                    j = regular_joints.pop(0)
                    self.queue.append(j)

        for joint in self.queue:
            joint.rectangle.visible = False
        self.passivate()

class PTC(sim.Component):
    def setup(self, position, psv):
        self.position = position
```

```
107         self.x, self.y = position_map[self.position]
108         self.psv = psv
109         self.joint_counter = 0
110         self.lift_time_min = parameters['PTC']['lift_time_min']
111         self.lift_time_mode = parameters['PTC']['lift_time_mode']
112         self.lift_time_max = parameters['PTC']['lift_time_max']
113         self.lift_time_mean = parameters['PTC']['lift_time_mean']
114         self.lift_time_constant = parameters['PTC']['lift_time_constant']
115         self.lift_time = sim.Triangular(self.lift_time_min, self.lift_time_max, self.
                lift_time_mode)
116         self.drop_time = parameters['PTC']['drop_time']
117         self.wait_memory = []
118         self.process_memory = []
119 
120         self.rectangle = sim.AnimateRectangle(
121             spec=(-0.5*joint_length, -0.5*joint_width, 0.5*joint_length, 0.5*joint_width),
122             fillcolor='',
123             linewidth=0,
124             text=f'Delivery PS' if self.position == pos['Delivery_ps'] else f'Delivery SB',
125             textcolor='black',
126             text_anchor='w',
127             fontsize=1.2*joint_width,
128             layer=100,
129             x=self.x * scale,
130             y=self.y * scale,
131         )
132 
133 
134     def process(self):
135         while self.psv.queue:
136             joint = self.psv.queue.pop(0)
137             self.joint_counter += 1
138             joint.id = self.joint_counter
139             joint.name = f'Joint({joint.id})'
140             joint.rectangle.visible = True
141             process_time = self.lift_time.sample()
142             self.hold(process_time)
143             process_time = process_time + self.drop_time
144             self.process_memory.append(process_time)
145 
146             if occupancy(self.position) != 0:
147                 start_wait = env.now()
148                 self.psv.circle.textcolor = 'red'
149                 self.wait((env.occupancy_map[self.position], 0))
150                 self.psv.circle.textcolor = 'white'
151                 wait_time = int(env.now() - start_wait)
152                 self.wait_memory.append(wait_time)
153 
154             set_occupancy(self.position, 1)
155             self.hold(self.drop_time)
156 
157             if self.position == pos['Delivery_ps']:
158                 env.data['PTC delivery']['1'] += 1
159             elif self.position == pos['Delivery_sb']:
160                 env.data['PTC delivery']['2'] += 1
161             env.data['PTC delivery']['Total'] += 1
162 
163             set_position(joint, self.position)
164             joint.rectangle.layer = 1
165 
166             pipe_routing_system.joints.append(joint)
167             joint.waiting_for_move = True
168             set_occupancy(self.position, 2)
169 
170         self.psv.rectangle.visible = False
171         self.psv.circle.visible = False
172         self.passivate()
173 
174 class PipeRoutingSystem(sim.Component):
175     def setup(self):
176         self.joints = []
```

```
177         self.joints_with_remainders = []
178         self.anode_count = 0
179         self.normal_count = 0
180         self.joint_0_5_to_hold = False
181         self.djf_supply_line = [None, None, None, None, None] if route=='b' else [None, None,
                None, None]
182         self.djf_supply_line_positions = [(1, 0)]
183         self.djf_supply_line_positions.append((0, 0)) if route in ['b', 'c', 'd'] else None
184         self.djf_supply_line_positions.append((1, 1)) if route == 'b' else None
185 
186     def process(self):
187         while True:
188             move_assigned = False
189             for joint in self.joints:
190                 if joint.ispassive() and joint.waiting_for_move:
191 
192                     if not joint.marked:
193                         joint.destination, joint.resource = move_map_marking.get(joint.
                                position, (None, None))
194                     elif joint.to_hold:
195                         joint.destination, joint.resource = move_map_hold.get(joint.position,
                                (None, None))
196                     elif joint.marked and not joint.cleaned:
197                         joint.destination, joint.resource = move_map_cleaning.get(joint.
                                position, (None, None))
198                     elif joint.marked and joint.cleaned:
199                         if joint.position in self.djf_supply_line_positions:
200                             if route in ['a', 'e', 'f']:
201                                 move_assigned = self.djf_supply_aef(joint)
202                             elif route in ['c', 'd']:
203                                 move_assigned = self.djf_supply_cd(joint)
204                             elif route in ['b']:
205                                 move_assigned = self.djf_supply_b(joint)
206                             if move_assigned:
207                                 break
208                             else:
209                                 continue
210                         else:
211                             joint.destination, joint.resource = move_map_djf.get(joint.
                                    position, (None, None))
212 
213                     if joint.destination is None:
214                         continue
215                     elif joint.destination == (pos['Cleaning_sb']) and occupancy(pos['
                            Cleaning_ps']) == 2.2 and (route == 'e' or route == 'f'):
216                         continue
217 
218                     move, remainder = self.path(joint.position, joint.destination)
219 
220                     if move and (joint.resource is None or self.resource_available(joint.
                            resource)):
221                         joint.move = move
222                         joint.remainder = remainder
223 
224                         for position in move:
225                             set_occupancy(position, 1)
226                         joint.activate()
227                         move_assigned = True
228 
229                         if remainder:
230                             self.joints_with_remainders.append(joint)
231 
232                         self.hold(0.1)
233                         break
234 
235             if not move_assigned:
236                 tc_1ps = resources['TC1ps']['component']
237                 if tc_1ps.ispassive():
238                     new_position = (0, 10)
239                     for position in [(0, 7), (0, 8), (0, 9), (0, 10)]:
240                         if occupancy(position) >= 1:
```

```
                        new_position = position
                        break
            tc_1ps.new_position = new_position
            tc_1ps.activate()

        tc_2ps = resources['TC2ps']['component']
        if tc_2ps.ispassive() and tc_2ps.position in posCleaning:
            new_position = (1, 5)
            tc_2ps.new_position = new_position
            tc_2ps.activate()

        if route in ['a', 'e', 'f']:
            tc_2sb = resources['TC2sb']['component']
            if tc_2sb.ispassive():
                new_position = (1, 2)
                for position in [(1, 1), (1, 2), (1, 3), (1, 4)]:
                    if occupancy(position) >= 2:
                        new_position = position
                        break
                tc_2sb.new_position = new_position
                tc_2sb.activate()

            tc_1sb = resources['TC1sb']['component']
            if tc_1sb.ispassive():
                if route == 'a' and tc_1sb.position in posMarking:
                    tc_1sb.new_position = (0, 4)
                    tc_1sb.activate()
                elif route == 'e':
                    new_position = (0, 4)
                    for position in [(0, 4), (0, 5)]:
                        if occupancy(position) >= 2:
                            new_position = position
                            break
                    tc_1sb.new_position = new_position
                    tc_1sb.activate()
                elif route == 'f':
                    new_position = (0, 4)
                    for position in [(0, 5), (0, 6)]:
                        if occupancy(position) >= 2:
                            new_position = position
                            break
                    tc_1sb.new_position = new_position
                    tc_1sb.activate()

        elif route == 'b':
            tc_1sb = resources['TC1sb']['component']
            if tc_1sb.ispassive():
                new_position = (0, 4) if occupancy((1, 4)) > occupancy((1, 3)) and
                    occupancy((1, 3)) >= 0 else (0, 3)
                for position in [(0, 2), (0, 3), (0, 4)]:
                    if occupancy(position) >= 1:
                        new_position = position
                        break
                tc_1sb.new_position = new_position
                tc_1sb.activate()

        elif route == 'c':
            tc_1sb = resources['TC1sb']['component']
            if tc_1sb.ispassive():
                new_position = (0, 4)
                for position in [(0, 2), (0, 3), (0, 4)]:
                    if occupancy(position) >= 1:
                        new_position = position
                        break
                tc_1sb.new_position = new_position
                tc_1sb.activate()

            tc_2sb = resources['TC2sb']['component']
            if tc_2sb.ispassive():
                new_position = (1, 2)
                for position in [(1, 1), (1, 2), (1, 3)]:
```

```
                            if occupancy(position) >= 2:
                                new_position = position
                                break
                        tc_2sb.new_position = new_position
                        tc_2sb.activate()

                elif route == 'd':
                    tc_1sb = resources['TC1sb']['component']
                    if tc_1sb.ispassive():
                        new_position = (0, 3)
                        for position in [(0, 2), (0, 3)]:
                            if occupancy(position) >= 1:
                                new_position = position
                                break
                        tc_1sb.new_position = new_position
                        tc_1sb.activate()
                    tc_2sb = resources['TC2sb']['component']
                    if tc_2sb.ispassive():
                        new_position = (1, 2)
                        for position in [(1, 1), (1, 2), (1, 4)]:
                            if occupancy(position) >= 2:
                                new_position = position
                                break
                        tc_2sb.new_position = new_position
                        tc_2sb.activate()

                self.passivate()
                self.hold(0.1)


    def path(self, start, end):
        start_x, start_y = start
        end_x, end_y = end
        move = []
        remainder = []

        if start_x != end_x:
            step = 1 if start_x < end_x else -1
            for x in range(start_x + step, end_x + step, step):
                position = (x, start_y)
                if occupancy(position) != 0:
                    remainder.append(position)
                    break
                move.append(position)

        elif start_y != end_y:
            step = 1 if start_y < end_y else -1
            for y in range(start_y + step, end_y + step, step):
                position = (start_x, y)
                if position not in env.occupancy_map:
                    continue
                if occupancy(position) != 0:
                    remainder.append(position)
                    break
                move.append(position)
        return move, remainder

    def path_update(self):
        self.hold(0.1)
        for joint in self.joints_with_remainders:
            if joint.path_updateable:
                move_extend, new_remainder = self.path(joint.move[-1], joint.destination)
                if (0, 6) in move_extend and self.joint_0_5_to_hold:
                    move_extend.remove((0, 6))
                    new_remainder.append((0, 6))

                if move_extend:
                    joint.move.extend(move_extend)
                    joint.remainder = new_remainder
                    for pos in move_extend:
```

```
                        set_occupancy(pos, 1)

                    if not new_remainder:
                        self.joints_with_remainders.remove(joint)

    def joint_to_hold(self, marking_position, marking_joint):
        if marking_joint.to_hold:
            return
        joint_0_5 = next((j for j in self.joints if j.position == (0, 5)), None)

        anode_ratio = 1 / (2 * parameters['FiringLine']['anode_interval'])
        total_count = self.normal_count + self.anode_count
        if total_count < 20:
            return

        system_minus6normal_ratio = (self.anode_count) / (total_count - 6)
        system_minus2anode_ratio = (self.anode_count - 2) / (total_count - 2)
        normal_to_hold = True if (system_minus6normal_ratio < anode_ratio) else False
        anode_to_hold = True if system_minus2anode_ratio > anode_ratio else False

        if not marking_joint.anode:
            if normal_to_hold:
                marking_joint.to_hold = True
            elif route == 'f':
                if marking_position == (0, 5):
                    marking_joint.to_hold = True if occupancy((0, 4)) != 0 else False
                elif marking_position == (0, 6):
                    marking_joint.to_hold = True if occupancy((0, 4)) != 0 or (joint_0_5 and
                        not joint_0_5.to_hold) else False
            else:
                if marking_position == (0, 6):
                    marking_joint.to_hold = True if occupancy((1, 5)) != 0 else False
            if marking_joint.to_hold:
                self.normal_count -= 1
                if route != 'f' and marking_position == (0, 5):
                    self.joint_0_5_to_hold = True
                env.data['Sent to hold']['1'] += 1
                env.data['Sent to hold']['Total'] += 1

        elif marking_joint.anode:
            marking_joint.to_hold = True if anode_to_hold else False
            if marking_joint.to_hold:
                self.anode_count -= 1
                if route != 'f' and marking_position == (0, 5):
                    self.joint_0_5_to_hold = True
                env.data['Sent to hold']['2'] += 1
                env.data['Sent to hold']['Total'] += 1


    def djf_supply_aef(self, joint):
        move_assigned = False
        self.djf_supply_line[1] = joint
        joint_0, joint_1, joint_aft, joint_fore = self.djf_supply_line
        if joint_1 and not joint_0 and not joint_aft and not joint_fore:
            if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore']) ==
                0:
                joint_1.destination = pos['DJFsupply_fore']
                joint_1.move = [pos['DJFsupply_aft'], pos['DJFsupply_fore']]
                for position in joint_1.move:
                    set_occupancy(position, 1)
                self.djf_supply_line[1] = None
                self.djf_supply_line[3] = joint_1
                joint_1.activate()
                move_assigned = True
        elif joint_1 and joint_fore and not joint_aft:
            if not joint_fore.anode or not joint_1.anode:
                joint_1.destination = pos['DJFsupply_aft']
                joint_1.move = [pos['DJFsupply_aft']]
                for position in joint_1.move:
                    set_occupancy(position, 1)
                self.djf_supply_line[1] = None
```

```
451                 self.djf_supply_line[2] = joint_1
452                 if djf.start_real_wait_ps or djf.start_real_wait_sb:
453                     start_wait = min(djf.start_real_wait_ps, djf.start_real_wait_sb) if djf.
                            start_real_wait_ps and djf.start_real_wait_sb else djf.
                            start_real_wait_ps if djf.start_real_wait_ps else djf.
                            start_real_wait_sb
454                     wait_time = round(env.now() - start_wait)
455                     if start_wait == djf.start_real_wait_ps:
456                         djf.real_wait_memory_ps.append(wait_time)
457                         djf.start_real_wait_ps = None
458                     elif start_wait == djf.start_real_wait_sb:
459                         djf.real_wait_memory_sb.append(wait_time)
460                         djf.start_real_wait_sb = None
461
462                 joint_1.activate()
463                 move_assigned = True
464             elif not joint_0:
465                 joint_1.destination = (0, 0)
466                 joint_1.move = [(0, 0)]
467                 for position in joint_1.move:
468                     set_occupancy(position, 1)
469                 self.djf_supply_line[1] = None
470                 self.djf_supply_line[0] = joint_1
471                 joint_1.activate()
472                 move_assigned = True
473             else:
474                 joint_1.destination = pos['DJFsupply_aft']
475                 joint_1.move = [pos['DJFsupply_aft']]
476                 for position in joint_1.move:
477                     set_occupancy(position, 1)
478                 self.djf_supply_line[1] = None
479                 self.djf_supply_line[2] = joint_1
480                 if djf.start_real_wait_ps or djf.start_real_wait_sb:
481                     start_wait = min(djf.start_real_wait_ps, djf.start_real_wait_sb) if djf.
                            start_real_wait_ps and djf.start_real_wait_sb else djf.
                            start_real_wait_ps if djf.start_real_wait_ps else djf.
                            start_real_wait_sb
482                     wait_time = round(env.now() - start_wait)
483                     if start_wait == djf.start_real_wait_ps:
484                         djf.real_wait_memory_ps.append(wait_time)
485                         djf.start_real_wait_ps = None
486                     elif start_wait == djf.start_real_wait_sb:
487                         djf.real_wait_memory_sb.append(wait_time)
488                         djf.start_real_wait_sb = None
489                 joint_1.activate()
490                 move_assigned = True
491         elif not joint_0 and joint_1 and joint_aft and joint_fore:
492             joint_1.destination = (0, 0)
493             joint_1.move = [(0, 0)]
494             for position in joint_1.move:
495                 set_occupancy(position, 1)
496             self.djf_supply_line[1] = None
497             self.djf_supply_line [0] = joint_1
498             joint_1.activate()
499             move_assigned = True
500         elif joint_0 and joint_1 and not joint_aft and not joint_fore:
501             if joint_0.anode and joint_1.anode:
502                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
503                     joint_1.destination = pos['DJFsupply_fore']
504                     joint_1.move = [pos['DJFsupply_aft'], pos['DJFsupply_fore']]
505                     for position in joint_1.move:
506                         set_occupancy(position, 1)
507                     self.djf_supply_line[1] = None
508                     self.djf_supply_line[3] = joint_1
509                     joint_1.activate()
510                     move_assigned = True
511             else:
512                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
513                     joint_1.destination = pos['DJFsupply_fore']
```

```
                    joint_1.move = [pos['DJFsupply_fore']]
                    joint_0.destination = pos['DJFsupply_aft']
                    joint_0.move = [(1, 0), pos['DJFsupply_aft']]
                    for position in joint_1.move + joint_0.move:
                        set_occupancy(position, 1)
                    self.djf_supply_line[3] = joint_1
                    self.djf_supply_line[2] = joint_0
                    self.djf_supply_line[1] = None
                    self.djf_supply_line[0] = None
                    if djf.start_real_wait_ps:
                        wait_time = round(env.now() - djf.start_real_wait_ps)
                        djf.real_wait_memory_ps.append(wait_time)
                        djf.start_real_wait_ps = None
                    elif djf.start_real_wait_sb:
                        wait_time = round(env.now() - djf.start_real_wait_sb)
                        djf.real_wait_memory_sb.append(wait_time)
                        djf.start_real_wait_sb = None
                    joint_1.free_pos = False
                    joint_1.activate()
                    joint_0.activate()
                    move_assigned = True
        else:
            move_assigned = False

        return move_assigned


    def djf_supply_b(self, joint):
        move_assigned = False
        if joint.position == (0,0):
            self.djf_supply_line[0] = joint

        joint_0, joint_1, joint_aft, joint_fore, joint_reserve = self.djf_supply_line

        if joint == joint_reserve:
            if not joint_aft and joint_fore and joint_fore.anode:
                move_assigned = False
            elif joint_0 and joint_0.anode:
                move_assigned = False
            else:
                if occupancy((1, 0)) == 0:
                    joint.destination = (1, 0)
                    joint.move = [(1, 0)]
                    joint.resource = 'TC2sb'
                    for position in joint.move:
                        set_occupancy(position, 1)
                    self.djf_supply_line[4] = None
                    self.djf_supply_line[1] = joint
                    joint.activate()
                    move_assigned = True

        elif joint == joint_1:
            if joint.anode and ((not joint_aft and joint_fore and joint_fore.anode) or (
                joint_0 and joint_0.anode)):
                if occupancy((1, 1)) == 0:
                    joint.destination = (1, 1)
                    joint.move = [(1, 1)]
                    joint.resource = 'TC2sb'
                    for position in joint.move:
                        set_occupancy(position, 1)
                    self.djf_supply_line[1] = None
                    self.djf_supply_line[4] = joint
                    joint.activate()
                    move_assigned = True
            elif not joint_0:
                if not joint_aft and not joint_fore:
                    if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore
                        ']) == 0:
                        joint.destination = pos['DJFsupply_fore']
                        joint.move = [pos['DJFsupply_aft'], pos['DJFsupply_fore']]
                        for position in joint.move:
```

```
583                             set_occupancy(position, 1)
584                         self.djf_supply_line[1] = None
585                         self.djf_supply_line[3] = joint
586                         joint.activate()
587                         move_assigned = True
588                 elif not joint_aft and joint_fore:
589                     if occupancy(pos['DJFsupply_aft']) == 0:
590                         joint.destination = pos['DJFsupply_aft']
591                         joint.move = [pos['DJFsupply_aft']]
592                         for position in joint.move:
593                             set_occupancy(position, 1)
594                         self.djf_supply_line[1] = None
595                         self.djf_supply_line[2] = joint
596                         joint.activate()
597                         move_assigned = True
598 
599         elif joint == joint_0:
600             if not joint_1 and not joint_aft and not joint_fore:
601                 if occupancy((1, 0)) == 0 and occupancy(pos['DJFsupply_aft']) == 0 and
                        occupancy(pos['DJFsupply_fore']) == 0:
602                     joint.destination = pos['DJFsupply_fore']
603                     joint.move = [(1, 0), pos['DJFsupply_aft'], pos['DJFsupply_fore']]
604                     for position in joint.move:
605                         set_occupancy(position, 1)
606                     self.djf_supply_line[0] = None
607                     self.djf_supply_line[3] = joint
608                     joint.activate()
609                     move_assigned = True
610             elif not joint_1 and not joint_aft and joint_fore:
611                 if (not joint.anode or not joint_fore.anode) or joint_reserve:
612                     if occupancy((1, 0)) == 0 and occupancy(pos['DJFsupply_aft']) == 0:
613                         joint.destination = pos['DJFsupply_aft']
614                         joint.move = [(1, 0), pos['DJFsupply_aft']]
615                         for position in joint.move:
616                             set_occupancy(position, 1)
617                         self.djf_supply_line[0] = None
618                         self.djf_supply_line[2] = joint
619                         joint.activate()
620                         move_assigned = True
621                 else:
622                     if not joint_reserve:
623                         if occupancy((1, 0)) == 0:
624                             joint.destination = (1, 0)
625                             joint.move = [(1, 0)]
626                             for position in joint.move:
627                                 set_occupancy(position, 1)
628                             self.djf_supply_line[0] = None
629                             self.djf_supply_line[1] = joint
630                             joint.activate()
631                             move_assigned = True
632             elif not joint_1 and joint_aft and joint_fore:
633                 if occupancy((1, 0)) == 0:
634                     joint.destination = (1, 0)
635                     joint.move = [(1, 0)]
636                     for position in joint.move:
637                         set_occupancy(position, 1)
638                     self.djf_supply_line[0] = None
639                     self.djf_supply_line[1] = joint
640                     joint.activate()
641                     move_assigned = True
642             elif joint_1 and not joint_aft and not joint_fore:
643                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
644                     joint_1.destination = pos['DJFsupply_fore']
645                     joint_1.move = [pos['DJFsupply_fore']]
646                     joint.destination = pos['DJFsupply_aft']
647                     joint.move = [(1, 0), pos['DJFsupply_aft']]
648                     for position in joint_1.move + joint_0.move:
649                         set_occupancy(position, 1)
650                     self.djf_supply_line [3] = joint_1
651                     self.djf_supply_line [2] = joint
```

```
652                     self.djf_supply_line [1] = None
653                     self.djf_supply_line [0] = None
654                     joint_1.free_pos = False
655                     joint_1.activate()
656                     joint.activate()
657                     move_assigned = True
658         else:
659             move_assigned = False
660 
661         return move_assigned
662 
663 
664     def djf_supply_cd(self, joint):
665         move_assigned = False
666         if joint.position == (0,0):
667             self.djf_supply_line[0] = joint
668         elif joint.position == (1,0):
669             self.djf_supply_line[1] = joint
670         joint_0, joint_1, joint_aft, joint_fore = self.djf_supply_line
671 
672         if joint_aft and joint_fore:
673             move_assigned = False
674         elif not joint_aft and not joint_fore:
675             if joint_0 and not joint_1:
676                 if occupancy((1, 0)) == 0 and occupancy(pos['DJFsupply_aft']) == 0 and
                        occupancy(pos['DJFsupply_fore']) == 0:
677                     joint_0.destination = pos['DJFsupply_fore']
678                     joint_0.move = [(1, 0), pos['DJFsupply_aft'], pos['DJFsupply_fore']]
679                     for position in joint_0.move:
680                         set_occupancy(position, 1)
681                     self.djf_supply_line [0] = None
682                     self.djf_supply_line [3] = joint_0
683                     joint_0.activate()
684                     move_assigned = True
685             elif not joint_0 and joint_1:
686                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
687                     joint_1.destination = pos['DJFsupply_fore']
688                     joint_1.move = [pos['DJFsupply_aft'], pos['DJFsupply_fore']]
689                     for position in joint_1.move:
690                         set_occupancy(position, 1)
691                     self.djf_supply_line [1] = None
692                     self.djf_supply_line [3] = joint_1
693                     joint_1.activate()
694                     move_assigned = True
695             elif joint_0.anode and joint_1.anode:
696                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
697                     joint_1.destination = pos['DJFsupply_fore']
698                     joint_1.move = [pos['DJFsupply_aft'], pos['DJFsupply_fore']]
699                     for position in joint_1.move:
700                         set_occupancy(position, 1)
701                     self.djf_supply_line [1] = None
702                     self.djf_supply_line [3] = joint_1
703                     joint_1.activate()
704                     move_assigned = True
705             else:
706                 if occupancy(pos['DJFsupply_aft']) == 0 and occupancy(pos['DJFsupply_fore'])
                        == 0:
707                     joint_1.destination = pos['DJFsupply_fore']
708                     joint_1.move = [pos['DJFsupply_fore']]
709                     joint_0.destination = pos['DJFsupply_aft']
710                     joint_0.move = [(1, 0), pos['DJFsupply_aft']]
711                     for position in joint_1.move + joint_0.move:
712                         set_occupancy(position, 1)
713                     self.djf_supply_line [0] = None
714                     self.djf_supply_line [1] = None
715                     self.djf_supply_line [2] = joint_0
716                     self.djf_supply_line [3] = joint_1
717 
718                     joint_1.free_pos = False
```

```
                    joint_1.activate()
                    joint_0.activate()
                    move_assigned = True
        elif not joint_aft and joint_fore:
            if joint_1:
                if occupancy(pos['DJFsupply_aft']) == 0:
                    joint_1.destination = pos['DJFsupply_aft']
                    joint_1.move = [pos['DJFsupply_aft']]
                    for position in joint_1.move:
                        set_occupancy(position, 1)
                    self.djf_supply_line [1] = None
                    self.djf_supply_line [2] = joint_1
                    joint_1.activate()
                    move_assigned = True
            elif joint_0 and not joint_1:
                if not joint_0.anode or not joint_fore.anode:
                    if occupancy((1, 0)) == 0 and occupancy(pos['DJFsupply_aft']) == 0:
                        joint_0.destination = pos['DJFsupply_aft']
                        joint_0.move = [(1, 0), pos['DJFsupply_aft']]
                        for position in joint_0.move:
                            set_occupancy(position, 1)
                        self.djf_supply_line [0] = None
                        self.djf_supply_line [2] = joint_0
                        joint_0.activate()
                        move_assigned = True

        else:
            move_assigned = False

        return move_assigned


    def resource_available(self, resource):
        '''Check if a resource is available.'''
        tc = resources[resource]['component']
        tc_available = resources[resource]['resource'].available_quantity() > 0
        return tc.ispassive() and tc_available


class Joint(sim.Component):
    def setup(self, position, anode):
        self.id = None
        self.length = joint_length
        self.anode = anode
        self.position = position
        self.x, self.y = position_map[position]
        self.destination = None
        self.waiting_for_move = False
        self.path_updateable = False
        self.marked = False
        self.cleaned = False
        self.to_hold = False
        self.free_pos = True

        self.rectangle = sim.AnimateRectangle(
            spec=(-0.5*joint_length, -0.5*joint_width, 0.5*joint_length, 0.5*joint_width),
            fillcolor='gray',
            linewidth=0.5,
            linecolor='black',
            text=f'J' if not self.anode else 'AJ',
            fontsize=1.4*joint_width,
            textcolor='white' if not self.anode else 'orange',
            layer=1,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        self.passivate()
        while True:
            self.waiting_for_move = False
```

```
            self.path_updateable = True
            new_position =  self.move[-1]
            new_x, new_y = position_map[new_position]

            if new_x != self.x:
                set_occupancy(self.position, -1) if self.free_pos else None
                joint_lco_move(self, self.move)
                set_occupancy(self.position, 2)

            if new_y != self.y:
                resource_name = self.resource
                resource = resources[resource_name]['component']
                self.request(resources[resource_name]['resource'])
                set_occupancy(self.position, -1)
                resource.joint = self
                resource.new_position = self.position
                resource.activate()
                self.passivate()

                component = 'TC'
                joint_y_move(self, component, self.move)
                self.passivate()

                self.release(resources[resource_name]['resource'])
                set_occupancy(self.position, 2)

            self.destination = None
            self.move = []

            if self.position in posProcessingStations:
                self.passivate()

            if not self.move:
                self.waiting_for_move = True
                pipe_routing_system.activate()
                self.passivate()

class DoubleJoint(sim.Component):
    def setup(self, position, aft, fore, anode):
        self.id = (aft.id, fore.id)
        self.position = position
        self.x, self.y = position_map[self.position]
        self.aft = aft
        self.fore = fore
        self.anode = anode
        self.waiting_for_move = False
        self.marked = True
        self.cleaned = True
        self.reject = False
        self.to_hold = False
        self.length = 2 * joint_length
        self.free_pos = True

        self.rectangle = sim.AnimateRectangle(
            spec=(-joint_length, -0.5*joint_width, joint_length, 0.5*joint_width),
            fillcolor='gray',
            linewidth=0.5,
            text=f'({aft.id}) DJ ({fore.id})' if not self.anode else f'({aft.id}) ADJ ({fore.
                id})',
            fontsize=1.4*joint_width,
            textcolor='white' if not self.anode else 'orange',
            layer=1,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        self.passivate()
        while True:
            self.waiting_for_move = False
            new_x, new_y = position_map[self.destination]
```

```
860
861             if new_x != self.x:
862                 set_occupancy(self.position, -1)
863                 joint_lco_move(self, self.move)
864                 set_occupancy(self.position, 2)
865
866             if new_y != self.y:
867                 resource_name = self.resource
868                 resource = resources[resource_name]['component']
869                 self.request(resources[resource_name]['resource'])
870                 set_occupancy(self.position, -1)
871                 resource.joint = self
872                 resource.new_position = self.position
873                 resource.activate()
874                 self.passivate()
875                 if resource_name == 'Crane38':
876                     component = 'Crane38'
877                 else:
878                     component = 'TC'
879                 joint_y_move(self, component, self.move)
880                 self.passivate()
881                 self.release(resources[resource_name]['resource'])
882                 set_occupancy(self.position, 2)
883
884             self.destination = None
885             self.move = []
886
887             if self.position in posProcessingStations:
888                 self.passivate()
889
890             if not self.move:
891                 self.waiting_for_move = True
892                 if self in pipe_routing_system.joints:
893                     pipe_routing_system.activate()
894                 elif self in waitingarea.doublejoints:
895                     waitingarea.activate()
896                 self.passivate()
897
898 class TC(sim.Component):
899     def setup(self, position):
900         self.position = position
901         self.x, self.y = position_map[position]
902         self.lift_time = parameters['TC']['lift_time']
903         self.drop_time = parameters['TC']['drop_time']
904         self.new_position = self.position
905         self.joint = None
906
907
908         self.rectangle = sim.AnimateRectangle(
909             spec=(-0.5*tc_length, -0.5*tc_width, 0.5*tc_length, 0.5*tc_width),
910             fillcolor='darkred',
911             text=f'{self.name()}',
912             textcolor='white',
913             fontsize=1.4*joint_width,
914             layer=2,
915             x=self.x * scale,
916             y=self.y * scale,
917         )
918
919     def process(self):
920         while True:
921             self.passivate()
922             component = 'TC'
923             move = [self.new_position]
924             y_move(self, component, move)
925
926             if self.joint:
927                 self.hold(self.lift_time)
928                 self.joint.activate()
929                 y_move(self, component, self.joint.move)
930                 self.hold(self.drop_time)
```

```
                self.joint.activate()
                self.joint = None

class MarkingStation(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.marking_time = parameters['MarkingStation']['process_time']

        self.rectangle = sim.AnimateRectangle(
            spec=(-0.52*joint_length, -0.7*joint_width, 0.52*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))

            set_occupancy(self.position, 2.1)

            joint = next((j for j in pipe_routing_system.joints if j.position == self.
                position), None)
            if not joint.marked:
                self.hold(self.marking_time)
                joint.rectangle.text = f'J {joint.id}' if not joint.anode else f'AJ {joint.id
                    }'
                joint.marked = True
                if joint.anode:
                    pipe_routing_system.anode_count += 1
                else:
                    pipe_routing_system.normal_count += 1
                if self.position == pos['Marking_ps']:
                    env.data['Marked']['1'] += 1
                elif self.position == pos['Marking_sb']:
                    env.data['Marked']['2'] += 1
                env.data['Marked']['Total'] += 1
            elif self.position == pos['Marking_ps'] and joint.to_hold:
                pipe_routing_system.joint_0_5_to_hold = False
            set_occupancy(self.position, 2.2)
            pipe_routing_system.joint_to_hold(self.position, joint)
            joint.activate()

class CleaningStation(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.cleaning_time = parameters['CleaningStation']['process_time']

        self.rectangle = sim.AnimateRectangle(
            spec=(-0.52*joint_length, -0.7*joint_width, 0.52*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))
            set_occupancy(self.position, 2.1)
```

```
1000
1001            joint = next((j for j in pipe_routing_system.joints if j.position == self.
                    position), None)
1002            if not joint.cleaned:
1003                self.hold(self.cleaning_time)
1004                joint.cleaned = True
1005                if self.position == pos['Cleaning_ps']:
1006                    env.data['Cleaned']['1'] += 1
1007                elif self.position == pos['Cleaning_sb']:
1008                    env.data['Cleaned']['2'] += 1
1009                env.data['Cleaned']['Total'] += 1
1010            set_occupancy(self.position, 2.2)
1011            joint.activate()
1012
1013 class Hold(sim.Component):
1014     def setup(self, position, location, storage_active, retrieval_active):
1015         self.position = position
1016         self.storage_location = location
1017         self.storage_active = storage_active
1018         self.retrieval_active = retrieval_active
1019         self.x, self.y = position_map[self.storage_location]
1020         self.lift_time = parameters['Hold']['lift_time']
1021         self.storage_time = parameters['Hold']['storage_time']
1022         self.retrieval_time = parameters['Hold']['retrieval_time']
1023         self.normal_storage = []
1024         self.anode_storage = []
1025
1026         self.rectangle = sim.AnimateRectangle(
1027             spec=(-0.6*joint_length, -joint_width, 0.6*joint_length, joint_width),
1028             fillcolor=f'black' if not self.storage_active and not self.retrieval_active else
                    'lightgray',
1029             linewidth=1,
1030             linecolor='black',
1031             text=f'{self.name()}' if not self.storage_active and not self.retrieval_active
                    else f'{self.name()} with {len(self.normal_storage)} J and {len(self.
                    normal_storage)} AJ',
1032             fontsize=1.4*joint_width,
1033             textcolor='white' if not self.storage_active and not self.retrieval_active else '
                    black',
1034             layer=10,
1035             x=self.x * scale,
1036             y=self.y * scale,
1037         )
1038
1039     def process(self):
1040         while self.storage_active:
1041             self.wait((env.occupancy_map[self.position], 2))
1042             joint = next((j for j in pipe_routing_system.joints if j.position == self.
                    position), None)
1043             if joint.to_hold:
1044                 self.hold(self.lift_time)
1045                 set_position(joint, self.storage_location)
1046                 set_occupancy(self.position, 0)
1047                 self.hold(self.storage_time)
1048
1049                 joint.rectangle.visible = False
1050                 self.anode_storage.append(joint) if joint.anode else self.normal_storage.
                        append(joint)
1051                 self.rectangle.text = f'{self.name()} with {len(self.normal_storage)} J and {
                        len(self.anode_storage)} AJ'
1052                 pipe_routing_system.joints.remove(joint)
1053         self.passivate()
1054
1055 class DJF(sim.Component):
1056     def setup(self):
1057         self.posAft = pos['DJFsupply_aft']
1058         self.posFore = pos['DJFsupply_fore']
1059         self.start_wait_ps = None
1060         self.start_wait_sb = None
1061         self.start_real_wait_ps = None
1062         self.start_real_wait_sb = None
```

```
1063            self.wait_memory_ps = []
1064            self.wait_memory_sb = []
1065            self.real_wait_memory_ps = []
1066            self.real_wait_memory_sb = []
1067            self.doublejoints = []
1068
1069        def process(self):
1070            while True:
1071                self.wait((env.occupancy_map[self.posAft], 2), (env.occupancy_map[self.posFore],
                        2), all=True)
1072
1073                if occupancy(self.posAft) == 2 and occupancy(self.posFore) == 2:
1074                    joint_aft = next((j for j in pipe_routing_system.joints if j.position == self
                            .posAft), None)
1075                    joint_fore = next((j for j in pipe_routing_system.joints if j.position ==
                            self.posFore), None)
1076
1077                    if occupancy(bevelling_psa.position) == 0 and occupancy(bevelling_psf.
                            position) == 0:
1078                        joint_aft.destination, joint_aft.resource = bevelling_psa.position, '
                                TC3ps'
1079                        bevelling_psa.joint = joint_aft
1080                        joint_fore.destination, joint_fore.resource = bevelling_psf.position, '
                                TC4ps'
1081                        bevelling_psf.joint = joint_fore
1082                        for joint in [joint_aft, joint_fore]:
1083                            set_occupancy(joint.position, 2.1)
1084                            joint.move = [joint.destination]
1085                            set_occupancy(joint.destination, 1)
1086                            joint.activate()
1087                        pipe_routing_system.djf_supply_line[2] = None
1088                        pipe_routing_system.djf_supply_line[3] = None
1089                        if self.start_wait_ps is not None:
1090                            wait_time = round(env.now() - self.start_wait_ps)
1091                            self.wait_memory_ps.append(wait_time)
1092                            self.start_wait_ps = None
1093
1094
1095                    elif occupancy(bevelling_sba.position) == 0 and occupancy(bevelling_sbf.
                            position) == 0:
1096                        joint_aft.destination, joint_aft.resource = bevelling_sba.position, '
                                TC3sb'
1097                        joint_fore.destination, joint_fore.resource = bevelling_sbf.position, '
                                TC4sb'
1098                        for joint in [joint_aft, joint_fore]:
1099                            set_occupancy(joint.position, 2.1)
1100                            joint.move = [joint.destination]
1101                            set_occupancy(joint.destination, 1)
1102                            joint.activate()
1103                        pipe_routing_system.djf_supply_line[2] = None
1104                        pipe_routing_system.djf_supply_line[3] = None
1105                        if self.start_wait_sb is not None:
1106                            wait_time = round(env.now() - self.start_wait_sb)
1107                            self.wait_memory_sb.append(wait_time)
1108                            self.start_wait_sb = None
1109
1110                    else:
1111                        self.passivate()
1112
1113                else:
1114                    if occupancy(bevelling_psa.position) == 0 and occupancy(bevelling_psf.
                            position) == 0:
1115                        if self.start_wait_ps is None:
1116                            self.start_wait_ps = env.now()
1117                            if self.start_real_wait_ps is None:
1118                                if (pipe_routing_system.djf_supply_line[2] is None or
                                        pipe_routing_system.djf_supply_line[3] is None) and (
                                        pipe_routing_system.djf_supply_line[0] is None or
                                        pipe_routing_system.djf_supply_line[1] is None):
1119                                    self.start_real_wait_ps = env.now()
```

```
1120                elif occupancy(bevelling_sba.position) == 0 and occupancy(bevelling_sbf.
                        position) == 0:
1121                    if self.start_wait_sb is None:
1122                        self.start_wait_sb = env.now()
1123                        if self.start_real_wait_sb is None:
1124                            if (pipe_routing_system.djf_supply_line[2] is None or
                                    pipe_routing_system.djf_supply_line[3] is None) and (
                                    pipe_routing_system.djf_supply_line[0] is None or
                                    pipe_routing_system.djf_supply_line[1] is None):
1125                                self.start_real_wait_sb = env.now()
1126
1127
1128 class BevelStation(sim.Component):
1129     def setup(self, position, next_position, tc):
1130         self.position = position
1131         self.x, self.y = position_map[self.position]
1132         self.bevel_time_min = parameters['BevelStation']['process_time_min']
1133         self.bevel_time_ppd = parameters['BevelStation']['process_time_ppd']
1134         self.bevel_time_max = parameters['BevelStation']['process_time_max']
1135         self.beta1 = sim.Beta(parameters['BevelStation']['alpha1'], parameters['BevelStation'
                ]['beta1'])
1136         self.beta2 = sim.Beta(parameters['BevelStation']['alpha2'], parameters['BevelStation'
                ]['beta2'])
1137         self.w = parameters['BevelStation']['weight_factor']
1138         self.data_name = 'Bevel_aft' if self.position[0] == 2 else 'Bevel_fore'
1139         self.process_memory = []
1140         self.next_position = next_position
1141         self.tc = tc
1142
1143         self.rectangle = sim.AnimateRectangle(
1144             spec=(-0.52*joint_length, -0.7*joint_width, 0.52*joint_length, 0.7*joint_width),
1145             fillcolor='wheat',
1146             text=f'{self.name()}',
1147             fontsize=1.2*joint_width,
1148             textcolor='black',
1149             text_anchor='w',
1150             layer=80,
1151             x=self.x * scale,
1152             y=self.y * scale,
1153         )
1154
1155     def process(self):
1156         while True:
1157             self.wait((env.occupancy_map[self.position], 2))
1158             set_occupancy(self.position, 2.1)
1159             if occupancy(self.next_position) >= 2:
1160                 self.tc.new_position = self.next_position
1161                 self.tc.activate()
1162
1163             joint = next((j for j in pipe_routing_system.joints if j.position == self.
                    position), None)
1164             if random.random() < self.w:
1165                 process_time = self.beta1.sample() * (self.bevel_time_max - self.
                        bevel_time_min) + self.bevel_time_min
1166             else:
1167                 process_time = self.beta2.sample() * (self.bevel_time_max - self.
                        bevel_time_min) + self.bevel_time_min
1168             self.hold(process_time)
1169             self.process_memory.append(process_time)
1170             set_occupancy(self.position, 2.2)
1171             joint.activate()
1172
1173 class HeatingStation(sim.Component):
1174     def setup(self, position, next_station):
1175         self.position = position
1176         self.x, self.y = position_map[self.position]
1177         self.heating_time = parameters['HeatingStation']['process_time']
1178         self.next_station = next_station
1179
1180         self.rectangle = sim.AnimateRectangle(
1181             spec=(-0.52*joint_length, -0.7*joint_width, 0.52*joint_length, 0.7*joint_width),
```

```
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))
            set_occupancy(self.position, 2.1)
            djf.activate()
            joint = next((j for j in pipe_routing_system.joints if j.position == self.
                position), None)
            self.hold(self.heating_time)

            set_occupancy(self.position, 2.2)
            self.wait((env.occupancy_map[self.next_station.position], 0))
            joint.activate()


class WeldingStation1(sim.Component):
    def setup(self, position, posAft, posFore, next_station):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.welding1_time_min = parameters['WeldingStation1']['process_time_min']
        self.welding1_time_ppd = parameters['WeldingStation1']['process_time_ppd']
        self.welding1_time_max = parameters['WeldingStation1']['process_time_max']
        self.beta = sim.Beta(parameters['WeldingStation1']['alpha'], parameters['
            WeldingStation1']['beta'])

        self.data_name = 'Welding1_ps' if self.position[1] == 3 else 'Welding1_sb'
        self.posAft = posAft
        self.posFore = posFore
        self.next_station = next_station
        self.doublejoint_count = 0
        self.wait_memory = []
        self.process_memory = []

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.37*joint_length, -0.7*joint_width, 1.37*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='c',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.posAft], 2), (env.occupancy_map[self.posFore],
                2), all=True)

            set_occupancy(self.posAft, 2.1)
            set_occupancy(self.posFore, 2.1)
            joint_aft = next((j for j in pipe_routing_system.joints if j.position == self.
                posAft), None)
            joint_fore = next((j for j in pipe_routing_system.joints if j.position == self.
                posFore), None)

            set_occupancy(self.position, 2.1)
            process_time = self.beta.sample() * (self.welding1_time_max - self.
                welding1_time_min) + self.welding1_time_min
            self.hold(process_time)
            self.process_memory.append(process_time)
```

```
            start_wait = env.now()

            pipe_routing_system.joints.remove(joint_aft)
            pipe_routing_system.joints.remove(joint_fore)
            joint_aft.rectangle.visible = False
            joint_fore.rectangle.visible = False
            set_occupancy(self.posAft, 0)
            set_occupancy(self.posFore, 0)
            if not joint_aft.anode and not joint_fore.anode:
                doublejoint = DoubleJoint(name=f'DoubleJoint({joint_aft.id},{joint_fore.id})'
                    , position=self.position, aft=joint_aft, fore=joint_fore, anode=False)
            elif joint_aft.anode or joint_fore.anode:
                doublejoint = DoubleJoint(name=f'DoubleJoint({joint_aft.id},{joint_fore.id})'
                    , position=self.position, aft=joint_aft, fore=joint_fore, anode=True)
            set_occupancy(self.position, 2.2)
            self.hold(0.1)
            pipe_routing_system.joints.append(doublejoint)
            self.doublejoint_count += 1

            while occupancy(self.next_station.position) != 0:
                self.wait((env.occupancy_map[self.next_station.position], 0))
                wait_time = int(env.now() - start_wait)
                self.wait_memory.append(wait_time)

            doublejoint.destination = self.next_station.position
            doublejoint.move = [doublejoint.destination]
            set_occupancy(doublejoint.destination, 1)
            doublejoint.activate()

class Weldingstation2(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.welding2_time_min = parameters['WeldingStation2']['process_time_min']
        self.welding2_time_ppd = parameters['WeldingStation2']['process_time_ppd']
        self.welding2_time_max = parameters['WeldingStation2']['process_time_max']
        self.beta = sim.Beta(parameters['WeldingStation2']['alpha'], parameters['
            WeldingStation2']['beta'])
        self.data_name = 'Welding2_ps' if self.position[1] == 3 else 'Welding2_sb'
        self.process_memory = []

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))
            set_occupancy(self.position, 2.1)

            doublejoint = next((dj for dj in pipe_routing_system.joints if dj.position ==
                self.position), None)
            process_time = self.beta.sample() * (self.welding2_time_max - self.
                welding2_time_min) + self.welding2_time_min
            self.hold(process_time)
            self.process_memory.append(process_time)

            set_occupancy(self.position, 2.2)
            doublejoint.activate()

class WeldingStation34(sim.Component):
    def setup(self, position):
        self.position = position
```

```
1313            self.x, self.y = position_map[self.position]
1314            self.welding34_time_min = parameters['WeldingStation34']['process_time_min']
1315            self.welding34_time_ppd = parameters['WeldingStation34']['process_time_ppd']
1316            self.welding34_time_max = parameters['WeldingStation34']['process_time_max']
1317            self.beta = sim.Beta(parameters['WeldingStation34']['alpha'], parameters['
                    WeldingStation34']['beta'])
1318            self.data_name = 'Welding34_ps' if self.position[1] == 2 else 'Welding34_sb'
1319            self.process_memory = []
1320
1321            self.rectangle = sim.AnimateRectangle(
1322                spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
1323                fillcolor='wheat',
1324                text=f'{self.name()}',
1325                fontsize=1.2*joint_width,
1326                textcolor='black',
1327                text_anchor='w',
1328                layer=80,
1329                x=self.x * scale,
1330                y=self.y * scale,
1331            )
1332
1333        def process(self):
1334            while True:
1335                self.wait((env.occupancy_map[self.position], 2))
1336                set_occupancy(self.position, 2.1)
1337
1338                doublejoint = next((dj for dj in pipe_routing_system.joints if dj.position ==
                        self.position), None)
1339                process_time = self.beta.sample() * (self.welding34_time_max - self.
                        welding34_time_min) + self.welding34_time_min
1340                self.hold(process_time)
1341                self.process_memory.append(process_time)
1342
1343                set_occupancy(self.position, 2.2)
1344                doublejoint.activate()
1345
1346    class NDT(sim.Component):
1347        def setup(self, position, next_position):
1348            self.position = position
1349            self.x, self.y = position_map[self.position]
1350            self.testing_time = parameters['NDT']['process_time']
1351            self.reject_probability = parameters['NDT']['reject_probability']
1352            self.next_position = next_position
1353
1354            self.rectangle = sim.AnimateRectangle(
1355                spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
1356                fillcolor='wheat',
1357                text=f'{self.name()}',
1358                fontsize=1.2*joint_width,
1359                textcolor='black',
1360                text_anchor='w',
1361                layer=80,
1362                x=self.x * scale,
1363                y=self.y * scale,
1364            )
1365
1366        def process(self):
1367            while True:
1368                self.wait((env.occupancy_map[self.position], 2))
1369                set_occupancy(self.position, 2.1)
1370
1371                doublejoint = next((dj for dj in pipe_routing_system.joints if dj.position ==
                        self.position), None)
1372                self.hold(self.testing_time)
1373                if rng_djf.random() < self.reject_probability:
1374                    doublejoint.reject = True
1375                    doublejoint.rectangle.textcolor = 'red'
1376                    if not doublejoint.anode:
1377                        pipe_routing_system.normal_count -= 2
1378                        env.data['DJF rejects']['1'] += 1
1379                    elif doublejoint.anode:
```

```
                    for joint in [doublejoint.aft, doublejoint.fore]:
                        if not joint.anode:
                            pipe_routing_system.normal_count -= 1
                        elif joint.anode:
                            pipe_routing_system.anode_count -= 1
                    env.data['DJF rejects']['2'] += 1
                env.data['DJF rejects']['Total'] += 1
            else:
                doublejoint.reject = False
                if self.position == pos['NDT_ps']:
                    env.data['DJF production']['1'] += 1
                elif self.position == pos['NDT_sb']:
                    env.data['DJF production']['2'] += 1
                env.data['DJF production']['Total'] += 1
            set_occupancy(self.position, 2.2)
            pipe_routing_system.joints.remove(doublejoint)
            waitingarea.doublejoints.append(doublejoint)
            waitingarea.activate()
            self.wait((env.occupancy_map[self.next_position], 0))
            doublejoint.destination = self.next_position
            doublejoint.move = [doublejoint.destination]
            set_occupancy(doublejoint.destination, 1)
            doublejoint.activate()


class Crane(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[position]
        self.hoist_time = parameters['Crane38']['hoist_time']
        self.hoist_time_firingline = parameters['Crane38']['hoist_time_firingline']
        self.new_position = self.position
        self.joint = None
        self.idle = sim.State(name='Crane38_Idle', value=True)

        self.rectangle = sim.AnimateRectangle(
            spec=(-0.5*crane_length, -0.5*crane_width, 0.5*crane_length, 0.5*crane_width),
            fillcolor='',
            linewidth=2,
            linecolor='darkred',
            text=f'                {self.name()}',
            textcolor='darkred',
            text_anchor='w',
            fontsize=1.4*joint_width,
            layer=0,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.idle.set(True)
            self.passivate()
            self.idle.set(False)
            component = 'Crane38'
            move = [self.new_position]
            y_move(self, component, move)

            if self.joint and self.joint.position is firingline.position:
                lift_time = 2 * self.hoist_time_firingline
                self.hold(lift_time)
                while occupancy(deconstructor.position) != 0:
                    self.wait((env.occupancy_map[deconstructor.position], 0))
                set_occupancy(self.joint.destination, 1)
            elif self.joint:
                lift_time = 2 * self.hoist_time
                self.hold(lift_time)
            else:
                return

            self.joint.activate()
```

```
            y_move(self, component, self.joint.move)

            drop_time = self.hoist_time
            self.hold(drop_time)
            self.joint.activate()

            self.hold(self.hoist_time)
            self.joint = None

class Buffer(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.retrieval = False
        self.capacity = 6
        self.DJbuffer = []
        self.ADJbuffer = []

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()} with {len(self.DJbuffer)} DJ and {len(self.ADJbuffer)} ADJ',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))

            doublejoint = next((dj for dj in waitingarea.doublejoints if dj.position == self.
                position), None)
            if doublejoint:
                doublejoint.rectangle.visible = False
                doublejoint.position = 'Buffer'
                set_occupancy(self.position, 0)
                if not doublejoint.anode:
                    self.DJbuffer.append(doublejoint)
                elif doublejoint.anode:
                    self.ADJbuffer.append(doublejoint)
                self.rectangle.text = f'{self.name()} with {len(self.DJbuffer)} DJ and {len(
                    self.ADJbuffer)} ADJ'
                doublejoint = None

            elif self.retrieval:
                if not firingline.anode:
                    doublejoint = self.DJbuffer.pop()
                elif firingline.anode:
                    doublejoint = self.ADJbuffer.pop()
                self.rectangle.text = f'{self.name()} with {len(self.DJbuffer)} DJ and {len(
                    self.ADJbuffer)} ADJ'
                doublejoint.position = self.position
                doublejoint.rectangle.visible = True
                set_occupancy(self.position, 2.1)
                doublejoint.destination = elevator.position
                doublejoint.move = [doublejoint.destination]
                doublejoint.resource = 'Crane38'
                set_occupancy(doublejoint.destination, 1)
                doublejoint.activate()
                self.retrieval = False
                doublejoint = None

class WaitingArea(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.crane = resources['Crane38']['component']
```

```
1519            self.doublejoints = []
1520            self.wait_memory_normal = []
1521            self.wait_memory_anode = []
1522
1523        def process(self):
1524            start_wait_normal = None
1525            start_wait_anode = None
1526            while True:
1527                self.hold(1)
1528                doublejoint_sb = next((dj for dj in self.doublejoints if (dj.position == pos['
                        waiting_sb'] and dj.waiting_for_move == True)), None)
1529                doublejoint_ps = next((dj for dj in self.doublejoints if (dj.position == pos['
                        waiting_ps'] and dj.waiting_for_move == True)), None)
1530                doublejoint_reserve = next((dj for dj in self.doublejoints if (dj.position == pos
                        ['waiting_reserve'] and dj.waiting_for_move == True)), None)
1531
1532                doublejoints = [dj for dj in [doublejoint_sb, doublejoint_ps, doublejoint_reserve
                        ] if dj is not None]
1533                normal_doublejoints = len([dj for dj in doublejoints if not dj.anode]) + len(
                        buffer.DJbuffer)
1534                anode_doublejoints = len([dj for dj in doublejoints if dj.anode]) + len(buffer.
                        ADJbuffer)
1535
1536                if start_wait_normal is None:
1537                    if firingline.waiting and occupancy(elevator.position) == 0 and not
                            firingline.anode and normal_doublejoints == 0:
1538                        start_wait_normal = env.now()
1539                elif start_wait_normal is not None:
1540                    if normal_doublejoints > 0:
1541                        wait_time = round(env.now() - start_wait_normal)
1542                        self.wait_memory_normal.append(wait_time)
1543                        start_wait_normal = None
1544                if start_wait_anode is None:
1545                    if firingline.waiting and occupancy(elevator.position) == 0 and firingline.
                            anode and anode_doublejoints == 0:
1546                        start_wait_anode = env.now()
1547                elif start_wait_anode is not None:
1548                    if anode_doublejoints > 0:
1549                        wait_time = round(env.now() - start_wait_anode)
1550                        self.wait_memory_anode.append(wait_time)
1551                        start_wait_anode = None
1552
1553                if firingline.rejects:
1554                    for doublejoint in list(firingline.rejects):
1555                        while not self.crane.idle():
1556                            self.wait((self.crane.idle, True))
1557                        doublejoint.destination = deconstructor.position
1558                        doublejoint.move = [doublejoint.destination]
1559                        doublejoint.resource = 'Crane38'
1560                        doublejoint.activate()
1561                        firingline.rejects.remove(doublejoint)
1562                        self.hold(1)
1563                    firingline.activate()
1564                elif occupancy(pos['Elevator']) == 0 and doublejoint_sb and not doublejoint_sb.
                        reject and doublejoint_sb.anode == firingline.anode:
1565                    self.move(doublejoint_sb, elevator.position)
1566                elif occupancy(pos['Elevator']) == 0 and doublejoint_ps and not doublejoint_ps.
                        reject and doublejoint_ps.anode == firingline.anode:
1567                    self.move(doublejoint_ps, elevator.position)
1568                elif occupancy(pos['Elevator']) == 0 and doublejoint_reserve and
                        doublejoint_reserve.anode == firingline.anode:
1569                    self.move(doublejoint_reserve, elevator.position)
1570                elif occupancy(pos['Elevator']) == 0 and not firingline.anode and buffer.DJbuffer
                        :
1571                    buffer.retrieval = True
1572                    buffer.activate()
1573                elif occupancy(pos['Elevator']) == 0 and firingline.anode and buffer.ADJbuffer:
1574                    buffer.retrieval = True
1575                    buffer.activate()
1576                elif doublejoint_ps and doublejoint_ps.reject and occupancy(deconstructor.
                        position) == 0:
```

```
                self.move(doublejoint_ps, deconstructor.position)
            elif doublejoint_sb and doublejoint_sb.reject and occupancy(deconstructor.
                position) == 0:
                self.move(doublejoint_sb, deconstructor.position)
            elif doublejoint_ps and not doublejoint_ps.reject and occupancy(ndt_ps.position)
                == 2.2 and (occupancy(pos['waiting_reserve']) == 0 or len(buffer.DJbuffer)+
                len(buffer.ADJbuffer) < buffer.capacity):
                if occupancy(pos['waiting_reserve']) == 0:
                    self.move(doublejoint_ps, pos['waiting_reserve'])
                else:
                    self.move(doublejoint_ps, buffer.position)
            elif doublejoint_sb and not doublejoint_sb.reject and occupancy(ndt_sb.position)
                == 2.2 and (occupancy(pos['waiting_reserve']) == 0 or len(buffer.DJbuffer)+
                len(buffer.ADJbuffer) < buffer.capacity):
                if occupancy(pos['waiting_reserve']) == 0:
                    self.move(doublejoint_sb, pos['waiting_reserve'])
                else:
                    self.move(doublejoint_sb, buffer.position)
            else:
                self.passivate()
            self.hold(1)

            while not self.crane.idle():
                self.wait((self.crane.idle, True))

    def move(self, doublejoint, destination):
        doublejoint.destination = destination
        doublejoint.move = [doublejoint.destination]
        doublejoint.resource = 'Crane38'
        set_occupancy(doublejoint.position, -1)
        set_occupancy(doublejoint.destination, 1)
        doublejoint.activate()


class Elevator(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.elevator_time_empty = parameters['Elevator']['lift_time_empty']
        self.elevator_time_loaded = parameters['Elevator']['lift_time_loaded']
        self.unloading_time = parameters['Elevator']['unloading_time']

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'{self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))
            set_occupancy(self.position, 2.1)
            doublejoint = next((dj for dj in waitingarea.doublejoints if dj.position == self.
                position), None)
            self.hold(self.elevator_time_loaded)
            self.hold(self.unloading_time)
            set_occupancy(self.position, 2.2)

            doublejoint.rectangle.visible = False
            self.wait((env.occupancy_map[self.position], 2.3))

            doublejoint = None
            self.hold(self.elevator_time_empty)

            set_occupancy(self.position, 0)
```

```
            waitingarea.activate()

class FiringLine(sim.Component):
    def setup(self, position):
        self.position = position
        self.x, self.y = position_map[self.position]
        self.processing_time_mean = parameters['FiringLine']['process_time_mean']
        self.processing_time_ppd = parameters['FiringLine']['process_time_ppd']
        self.processing_time_pull = parameters['FiringLine']['process_time_pull']
        self.processing_time_min = parameters['FiringLine']['process_time_min']
        self.processing_time_max = parameters['FiringLine']['process_time_max']
        self.reject_length = parameters['FiringLine']['reject_length']
        self.reject_probability = parameters['FiringLine']['reject_probability']
        self.anode_interval = parameters['FiringLine']['anode_interval']
        self.anode = False
        self.counter = 1
        self.pipeline = []
        self.rejects = []
        self.wait_memory_normal = []
        self.wait_memory_anode = []
        self.process_memory = []
        self.waiting = False

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.02*joint_length, -0.7*joint_width, 1.02*joint_length, 0.7*joint_width),
            fillcolor='white',
            text=f'Hatch to {self.name()} (Normal)',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='w',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            if occupancy(pos['Elevator']) != 2.2:
                self.waiting = True
                waitingarea.activate()
                start_wait = env.now()
                self.wait((env.occupancy_map[pos['Elevator']], 2.2))
                self.waiting = False
                wait_time = int(env.now() - start_wait)
                if not self.anode:
                    self.wait_memory_normal.append(wait_time)
                else:
                    self.wait_memory_anode.append(wait_time)

            set_occupancy(pos['Elevator'], 2.3)
            doublejoint = next((dj for dj in waitingarea.doublejoints if dj.position == pos['
                Elevator']), None)
            doublejoint.position = self.position
            self.counter += 1
            if self.counter % self.anode_interval == 0:
                self.anode = True
                self.rectangle.text = f'Hatch to {self.name()} (Anode)'
            else:
                self.anode = False
                self.rectangle.text = f'Hatch to {self.name()} (Normal)'
            self.pipeline.append(doublejoint)
            env.data['Main pipeline length']['Total'] += 2 * parameters["Joint"]["length"]
            if len(self.pipeline) > self.reject_length:
                ndt_doublejoint = self.pipeline.pop(0)
                waitingarea.doublejoints.remove(ndt_doublejoint)
                if not ndt_doublejoint.anode:
                    pipe_routing_system.normal_count -= 2
                elif ndt_doublejoint.anode:
                    for joint in [ndt_doublejoint.aft, ndt_doublejoint.fore]:
                        if not joint.anode:
                            pipe_routing_system.normal_count -= 1
```

```
                        elif joint.anode:
                            pipe_routing_system.anode_count -= 1

            process_time = sample_duration('FiringLine')
            self.hold(process_time - self.processing_time_pull)
            self.process_memory.append(process_time)

            if rng_firingline.random() < self.reject_probability:
                self.rejects = self.pipeline[::-1]
                self.counter = self.counter - len(self.rejects)
                for doublejoint in self.rejects:
                    self.pipeline.remove(doublejoint)
                    doublejoint.reject = True
                    doublejoint.rectangle.textcolor = 'red'
                    set_position(doublejoint, self.position)
                    if not doublejoint.anode:
                        pipe_routing_system.normal_count -= 2
                        env.data['Firing line rejects']['1'] += 1
                    elif doublejoint.anode:
                        for joint in [doublejoint.aft, doublejoint.fore]:
                            if not joint.anode:
                                pipe_routing_system.normal_count -= 1
                            elif joint.anode:
                                pipe_routing_system.anode_count -= 1
                        env.data['Firing line rejects']['2'] += 1
                env.data['Main pipeline length']['Total'] -= len(self.rejects) * 2 *
                    parameters["Joint"]["length"]
                env.data['Firing line rejects']['Total'] += len(self.rejects)
                waitingarea.activate()
                while self.rejects:
                    self.passivate()
            else:
                self.hold(self.processing_time_pull)

            doublejoint = None

class Deconstructor(sim.Component):
    def setup(self):
        self.position = pos['Deconstruction']
        self.position_aft = pos['Deconstruction_aft']
        self.position_fore = pos['Deconstruction_fore']
        self.x, self.y = position_map[self.position]
        self.deconstruction_time = parameters['Deconstructor']['process_time']

        self.rectangle = sim.AnimateRectangle(
            spec=(-1.05*joint_length, -0.7*joint_width, 1.05*joint_length, 0.7*joint_width),
            fillcolor='wheat',
            text=f'                                {self.name()}',
            fontsize=1.2*joint_width,
            textcolor='black',
            text_anchor='c',
            layer=80,
            x=self.x * scale,
            y=self.y * scale,
        )

    def process(self):
        while True:
            self.wait((env.occupancy_map[self.position], 2))
            set_occupancy(self.position, 2.1)

            doublejoint = next((dj for dj in waitingarea.doublejoints if dj.position == self.
                position), None)
            aft_joint = doublejoint.aft
            fore_joint = doublejoint.fore
            self.hold(self.deconstruction_time)

            doublejoint.rectangle.visible = False
            waitingarea.doublejoints.remove(doublejoint)
            doublejoint.position = None
            set_position(aft_joint, self.position_aft)
```

```
            set_position(fore_joint, self.position_fore)
            for joint in [fore_joint, aft_joint]:
                set_occupancy(joint.position, 2)
                joint.cleaned = False
                joint.to_hold = True
                pipe_routing_system.joints.insert(0, joint)
                joint.activate()

            doublejoint = None
            self.wait((env.occupancy_map[self.position_aft], 0), (env.occupancy_map[self.
                position_fore], 0), all=True)
            set_occupancy(self.position, 0)
            waitingarea.activate()
# endregion


# Create an environment
env = sim.Environment(blind_animation=blind_animation)
env.random_seed(seed)

env.occupancy_map = {
    position: sim.State(name=str(position), value=0)
    for position in position_map
    if not isinstance(position, str)
}

def set_occupancy(position, value):
    '''Set the occupancy state of a position.'''
    env.occupancy_map[position].set(value)
    rectangle = background.get(position, None)

    if value == 0:
        rectangle.linecolor = 'black'
        rectangle.linewidth = 0.5
        if position in posRouting:
            pipe_routing_system.activate()
            pipe_routing_system.path_update()
    elif value == 1:
        rectangle.linecolor = 'darkorange'
        rectangle.linewidth = 1
    elif value == 2 or value == 2.1:
        rectangle.linecolor = 'red'
        rectangle.linewidth = 1
        if position in posRouting:
            pipe_routing_system.activate()
    elif value == -1:
        rectangle.linecolor = 'green'
        rectangle.linewidth = 1
    else:
        rectangle.linecolor = 'darkgreen'
        rectangle.linewidth = 1

def occupancy(position):
    '''Get the occupancy state of a position.'''
    return env.occupancy_map[position]()

def set_position(joint, position):
    '''Animate the position of a joint.'''
    joint.position = position
    joint.x, joint.y = position_map[joint.position]
    joint.rectangle.x = joint.x * scale
    joint.rectangle.y = joint.y * scale
    joint.rectangle.visible = True

def joint_lco_move(self, move):
    # acceleration
    next_x = position_map[move[0]][0]
    direction = 1 if (next_x - self.x) > 0 else -1
    dx_acc = direction * LCO_a_distance
    steps = max(int(abs(dx_acc) / stepsize), 1)
    for step in range(steps):
```

```
1851            t = step / steps
1852            interpolated_x = self.x + t * dx_acc
1853            self.rectangle.x = interpolated_x * scale
1854            self.hold(LCO_a_time / steps)
1855        self.x = interpolated_x
1856
1857        # constant speed
1858        for i, position in enumerate(move):
1859            old_position = self.position
1860            old_x = position_map[old_position][0]
1861            next_x = position_map[position][0]
1862            dx = next_x - self.x
1863            final_position = True if i == len(move) - 1 else False
1864            if final_position:
1865                dx -= direction * LCO_a_distance
1866            movement_time = abs(dx) / LCO_v
1867            steps = max(int(abs(dx) / stepsize), 1)
1868            for step in range(steps):
1869                t = step / steps
1870                interpolated_x = self.x + t * dx
1871                self.rectangle.x = interpolated_x * scale
1872
1873                # Free the old position
1874                x_space = self.length / scale
1875                if abs(interpolated_x - old_x) > x_space and old_position:
1876                    set_occupancy(old_position, 0)  if self.free_pos else None
1877                    self.free_pos = True
1878                    old_position = None
1879
1880                self.hold(movement_time / steps)
1881
1882            self.x = interpolated_x
1883            self.position = position
1884        self.path_updateable = False
1885
1886        # deceleration
1887        dx_dec = direction * LCO_a_distance
1888        steps = max(int(abs(dx_dec) / stepsize), 1)
1889        for step in range(steps):
1890            t = step / steps
1891            interpolated_x = self.x + t * dx_dec
1892            self.rectangle.x = interpolated_x * scale
1893            if abs(interpolated_x - old_x) > x_space and old_position:
1894                set_occupancy(old_position, 0)
1895                old_position = None
1896            self.hold(LCO_a_time / steps)
1897        self.x = position_map[self.position][0]
1898
1899    def y_acceleration(self, component, direction):
1900        dy_acc = direction * parameters[component]['acceleration_distance']
1901        steps = max(int(abs(dy_acc) / stepsize), 1)
1902        for step in range(steps):
1903            t = step / steps
1904            interpolated_y = self.y + t * dy_acc
1905            self.rectangle.y = interpolated_y * scale
1906            self.hold(parameters[component]['acceleration_time'] / steps)
1907        self.y = interpolated_y
1908
1909    def y_deceleration(self, component, direction):
1910        dy_dec = direction * parameters[component]['acceleration_distance']
1911        steps = max(int(abs(dy_dec) / stepsize), 1)
1912        for step in range(steps):
1913            t = step / steps
1914            interpolated_y = self.y + t * dy_dec
1915            self.rectangle.y = interpolated_y * scale
1916            self.hold(parameters[component]['acceleration_time'] / steps)
1917        self.y = position_map[self.position][1]
1918
1919    def y_move(self, component, move):
1920        # direction
1921        next_y = position_map[move[0]][1]
```

```
    if next_y == self.y:
        return
    direction = 1 if (next_y - self.y) > 0 else -1

    # acceleration
    y_acceleration(self, component, direction)

    # constant speed
    for i, position in enumerate(move):
        next_y = position_map[position][1]
        dy = next_y - self.y
        final_position = True if i == len(move) - 1 else False
        if final_position:
            dy -= direction * parameters[component]['acceleration_distance']
        movement_time = abs(dy) / parameters[component]['velocity']
        steps = max(int(abs(dy) / stepsize), 1)
        for step in range(steps):
            t = step / steps
            interpolated_y = self.y + t * dy
            self.rectangle.y = interpolated_y * scale
            self.hold(movement_time / steps)
        self.y = interpolated_y
        self.position = position

    # deceleration
    y_deceleration(self, component, direction)

def joint_y_move(self, component, move):
    # direction
    next_y = position_map[move[0]][1]
    direction = 1 if (next_y - self.y) > 0 else -1

    # acceleration
    y_acceleration(self, component, direction)

    # constant speed
    for i, position in enumerate(move):
        old_position = self.position
        old_y = position_map[old_position][1]
        next_y = position_map[position][1]
        dy = next_y - self.y
        final_position = True if i == len(move) - 1 else False
        if final_position:
            dy -= direction * parameters[component]['acceleration_distance']
        movement_time = abs(dy) / parameters[component]['velocity']
        steps = max(int(abs(dy) / stepsize), 1)
        for step in range(steps):
            t = step / steps
            interpolated_y = self.y + t * dy
            self.rectangle.y = interpolated_y * scale

            # Free the old position
            y_space = joint_width / scale
            if abs(interpolated_y - old_y) > y_space and old_position:
                set_occupancy(old_position, 0)
                old_position = None

            self.hold(movement_time / steps)

        self.y = interpolated_y
        self.position = position
    self.path_updateable = False

    # deceleration
    y_deceleration(self, component, direction)

# region Generate components
resources = {
    'TC1ps': {
        'resource': sim.Resource(name='TC1ps', capacity=1),
        'component': TC(name='TC1ps', position=(0,11)),
```

```
    },
    'TC1sb': {
        'resource': sim.Resource(name='TC1sb', capacity=1),
        'component': TC(name='TC1sb', position=(0,0)),
    },
    'TC2ps': {
        'resource': sim.Resource(name='TC2ps', capacity=1),
        'component': TC(name='TC2ps', position=(1,6)),
    },
    'TC2sb': {
        'resource': sim.Resource(name='TC2sb', capacity=1),
        'component': TC(name='TC2sb', position=(1,0)),
    },
    'TC3ps': {
        'resource': sim.Resource(name='TC3ps', capacity=1),
        'component': TC(name='TC3ps', position=(2,3)),
    },
    'TC3sb': {
        'resource': sim.Resource(name='TC3sb', capacity=1),
        'component': TC(name='TC3sb', position=(2,-3)),
    },
    'TC4ps': {
        'resource': sim.Resource(name='TC4ps', capacity=1),
        'component': TC(name='TC4ps', position=(3,3)),
    },
    'TC4sb': {
        'resource': sim.Resource(name='TC4sb', capacity=1),
        'component': TC(name='TC4sb', position=(3,-3)),
    },
    'TC5ps': {
        'resource': sim.Resource(name='TC5ps', capacity=1),
        'component': TC(name='TC5ps', position=(4,1)),
    },
    'TC5sb': {
        'resource': sim.Resource(name='TC5sb', capacity=1),
        'component': TC(name='TC5sb', position=(4,-1)),
    },
    'Crane38': {
        'resource': sim.Resource(name='Crane38', capacity=1),
        'component': Crane(name='Crane38', position=(6, 2)),
    },
}

deconstructor = Deconstructor(name='Deconstructor')
firingline = FiringLine(name='Firing Line', position=pos['FiringLine'])
elevator = Elevator(name='Elevator', position=pos['Elevator'])
waitingarea = WaitingArea(name='Waiting Area', position='WaitingArea')
buffer = Buffer(name='Buffer', position=pos['Buffer'])

ndt_ps = NDT(name='NDT ps', position=pos['NDT_ps'], next_position=pos['waiting_ps'])
ndt_sb = NDT(name='NDT sb', position=pos['NDT_sb'], next_position=pos['waiting_sb'])

welding34_ps = WeldingStation34(name='Welding34 ps', position=pos['Welding34_ps'])
welding34_sb = WeldingStation34(name='Welding34 sb', position=pos['Welding34_sb'])

welding2_ps = Weldingstation2(name='Welding2 ps', position=pos['Welding2_ps'])
welding2_sb = Weldingstation2(name='Welding2 sb', position=pos['Welding2_sb'])

welding1_ps = WeldingStation1(name='Welding1 ps', position=pos['Welding1_ps'], posAft=pos['
    Welding1_ps_aft'], posFore=pos['Welding1_ps_fore'], next_station=welding2_ps)
welding1_sb = WeldingStation1(name='Welding1 sb', position=pos['Welding1_sb'], posAft=pos['
    Welding1_sb_aft'], posFore=pos['Welding1_sb_fore'], next_station=welding2_sb)

heating_psa = HeatingStation(name='Heating ps aft', position=pos['Heating_ps_aft'],
    next_station=welding1_ps)
heating_psf = HeatingStation(name='Heating ps fore', position=pos['Heating_ps_fore'],
    next_station=welding1_ps)
heating_sba = HeatingStation(name='Heating sb aft', position=pos['Heating_sb_aft'],
    next_station=welding1_sb)
heating_sbf = HeatingStation(name='Heating sb fore', position=pos['Heating_sb_fore'],
    next_station=welding1_sb)
```

```
2058
2059 bevelling_psa = BevelStation(name='Bevelling ps aft', position=pos['Bevelling_ps_aft'],
         next_position=pos['Heating_ps_aft'], tc=resources['TC3ps']['component'])
2060 bevelling_psf = BevelStation(name='Bevelling ps fore', position=pos['Bevelling_ps_fore'],
         next_position=pos['Heating_ps_fore'], tc=resources['TC4ps']['component'])
2061 bevelling_sba = BevelStation(name='Bevelling sb aft', position=pos['Bevelling_sb_aft'],
         next_position=pos['Heating_sb_aft'], tc=resources['TC3sb']['component'])
2062 bevelling_sbf = BevelStation(name='Bevelling sb fore', position=pos['Bevelling_sb_fore'],
         next_position=pos['Heating_sb_fore'], tc=resources['TC4sb']['component'])
2063
2064 djf = DJF(name='DJF')
2065
2066 hold1 = Hold(name='Hold 1', position=(5, 6), location='Hold1', storage_active=False,
         retrieval_active=False)
2067 hold2 = Hold(name='Hold 2', position=(4, 6), location='Hold2', storage_active=True,
         retrieval_active=False)
2068 hold3 = Hold(name='Hold 3', position=(3, 6), location='Hold3', storage_active=True,
         retrieval_active=False)
2069 hold4 = Hold(name='Hold 4', position=(2, 6), location='Hold4', storage_active=False,
         retrieval_active=False)
2070 hold5 = Hold(name='Hold 5', position=(1, 6), location='Hold5', storage_active=False,
         retrieval_active=False)
2071 hold6 = Hold(name='Hold 6', position=(-1, 6), location='Hold6', storage_active=False,
         retrieval_active=False)
2072
2073 marking_ps = MarkingStation(name='Marking ps', position=pos['Marking_ps'])
2074 marking_sb = MarkingStation(name='Marking sb', position=pos['Marking_sb'])
2075
2076 cleaning_ps = CleaningStation(name='Cleaning ps', position=pos['Cleaning_ps'])
2077 cleaning_sb = CleaningStation(name='Cleaning sb', position=pos['Cleaning_sb'])
2078
2079 psv_ps = PSV(name='PSV', position='PSVps', joint_amount=parameters['PSVps']['joint_amount'],
         anode_ratio=parameters['PSVps']['anode_ratio'])
2080 ptc_ps = PTC(name='PTC', position=pos['Delivery_ps'], psv=psv_ps)
2081
2082 psv_sb = PSV(name='PSV', position='PSVsb', joint_amount=parameters['PSVsb']['joint_amount'],
         anode_ratio=parameters['PSVsb']['anode_ratio'])
2083 ptc_sb = PTC(name='PTC', position=pos['Delivery_sb'], psv=psv_sb)
2084
2085 pipe_routing_system = PipeRoutingSystem(name='PipeRoutingSystem')
2086 # endregion
2087
2088 create_background()
2089 def format_sim_time():
2090     now = int(env.now()) - warm_up_period
2091     days = now // 86400
2092     hours = (now % 86400) // 3600
2093     minutes = (now % 3600) // 60
2094     seconds = now % 60
2095     return f"{days:02d}d {hours:02d}:{minutes:02d}:{seconds:02d}"
2096
2097 def right_align_lines(text, width=30):
2098     lines = text.split('\n')
2099     return '\n'.join(line.rjust(width) for line in lines)
2100
2101 sim.AnimateText(
2102     text=lambda: right_align_lines(f"{title}\nSeed {seed}\nTime: {format_sim_time()}"),
2103     x=deconstructor.x * scale + joint_length,
2104     y=2.3 * deconstructor.y * scale,
2105     fontsize=10,
2106     textcolor="black",
2107     text_anchor="e",
2108     layer=200
2109 )
2110
2111 # region Output data
2112 env.data = {
2113     'PTC delivery': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2114     'Marked': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2115     'Cleaned': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2116     'Sent to hold': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
```

```
    'Sent to hold ratio': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': '-'},
    'DJF production': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
    'DJF rejects': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
    'Firing line rejects': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
    'Main pipeline length': {'(1/2)': None,'1': None, '2': None, 'Total': 0, 'Unit': 'm'},
    }

def data_report():
    env.data['Sent to hold ratio']['1'] = round(env.data['Sent to hold']['1'] / env.data['Sent to hold']['Total'], 2) if env.data['Sent to hold']['Total'] > 0 else 0
    env.data['Sent to hold ratio']['2'] = round(env.data['Sent to hold']['2'] / env.data['Sent to hold']['Total'], 2) if env.data['Sent to hold']['Total'] > 0 else 0
    env.data['Sent to hold ratio']['Total'] = round(env.data['Sent to hold']['Total'] / env.data['PTC delivery']['Total'], 2) if env.data['PTC delivery']['Total'] > 0 else 0
    df_data = pd.DataFrame(env.data).T
    return df_data

def wait_report():
    env.wait_report = {
        'PTC wait count': {
            '(1/2)': '(PS/SB)',
            '1': len(ptc_ps.wait_memory),
            '2': len(ptc_sb.wait_memory),
            'Total': len(ptc_ps.wait_memory) + len(ptc_sb.wait_memory),
            'Unit': '-'
        },
        'PTC wait time': {
            '(1/2)': '(PS/SB)',
            '1': round(sum(ptc_ps.wait_memory) / 60),
            '2': round(sum(ptc_sb.wait_memory) / 60),
            'Total': round((sum(ptc_ps.wait_memory) + sum(ptc_sb.wait_memory)) / 60),
            'Unit': 'min'
        },
        'DJF wait count': {
            '(1/2)': '(PS/SB)',
            '1': len(djf.wait_memory_ps),
            '2': len(djf.wait_memory_sb),
            'Total': len(djf.wait_memory_ps) + len(djf.wait_memory_sb),
            'Unit': '-'
        },
        'DJF wait time': {
            '(1/2)': '(PS/SB)',
            '1': round(sum(djf.wait_memory_ps) / 60),
            '2': round(sum(djf.wait_memory_sb) / 60),
            'Total': round((sum(djf.wait_memory_ps) + sum(djf.wait_memory_sb)) / 60),
            'Unit': 'min'
        },
        'DJF real wait count': {
            '(1/2)': '(PS/SB)',
            '1': len(djf.real_wait_memory_ps),
            '2': len(djf.real_wait_memory_sb),
            'Total': len(djf.real_wait_memory_ps) + len(djf.real_wait_memory_sb),
            'Unit': '-'
        },
        'DJF real wait time': {
            '(1/2)': '(PS/SB)',
            '1': round(sum(djf.real_wait_memory_ps) / 60),
            '2': round(sum(djf.real_wait_memory_sb) / 60),
            'Total': round((sum(djf.real_wait_memory_ps) + sum(djf.real_wait_memory_sb)) / 60),
            'Unit': 'min'
        },
        'Waiting area wait count': {
            '(1/2)': '(Normal/Anode)',
            '1': len(waitingarea.wait_memory_normal),
            '2': len(waitingarea.wait_memory_anode),
            'Total': len(waitingarea.wait_memory_normal) + len(waitingarea.wait_memory_anode),
            'Unit': '-'
```

```
2181            },
2182            'Waiting area wait time': {
2183                '(1/2)': '(Normal/Anode)',
2184                '1': round(sum(waitingarea.wait_memory_normal) / 60),
2185                '2': round(sum(waitingarea.wait_memory_anode) / 60),
2186                'Total': round((sum(waitingarea.wait_memory_normal) + sum(waitingarea.
                        wait_memory_anode)) / 60),
2187                'Unit': 'min'
2188            },
2189            'Firing line wait count': {
2190                '(1/2)': '(Normal/Anode)',
2191                '1': len(firingline.wait_memory_normal),
2192                '2': len(firingline.wait_memory_anode),
2193                'Total': len(firingline.wait_memory_normal) + len(firingline.wait_memory_anode),
2194                'Unit': '-'
2195            },
2196            'Firing line wait time': {
2197                '(1/2)': '(Normal/Anode)',
2198                '1': round(sum(firingline.wait_memory_normal) / 60),
2199                '2': round(sum(firingline.wait_memory_anode) / 60),
2200                'Total': round((sum(firingline.wait_memory_normal) + sum(firingline.
                        wait_memory_anode)) / 60),
2201                'Unit': 'min'
2202            }
2203        }
2204
2205        df_wait_report = pd.DataFrame(env.wait_report).T
2206        return df_wait_report
2207
2208    def station_report():
2209        env.station_report = {
2210            'PTC process duration': {
2211                '(1/2)': '(PS/SB)',
2212                '1': round(mean(ptc_ps.process_memory)) if ptc_ps.process_memory else 0,
2213                '2': round(mean(ptc_sb.process_memory)) if ptc_sb.process_memory else 0,
2214                'Total': round(mean(ptc_ps.process_memory + ptc_sb.process_memory)) if (ptc_ps.
                        process_memory or ptc_sb.process_memory) else 0,
2215                'Unit': 's'
2216            },
2217            'Bevel process duration': {
2218                '(1/2)': '(PS/SB)',
2219                '1': round(mean(bevelling_psa.process_memory + bevelling_psf.process_memory)) if
                        (bevelling_psa.process_memory or bevelling_psf.process_memory) else 0,
2220                '2': round(mean(bevelling_sba.process_memory + bevelling_sbf.process_memory)) if
                        (bevelling_sba.process_memory or bevelling_sbf.process_memory) else 0,
2221                'Total': round(mean(bevelling_psa.process_memory + bevelling_psf.process_memory +
                         bevelling_sba.process_memory + bevelling_sbf.process_memory)) if (
                        bevelling_psa.process_memory or bevelling_psf.process_memory or bevelling_sba
                        .process_memory or bevelling_sbf.process_memory) else 0,
2222                'Unit': 's'
2223            },
2224            'Welding 1 process duration': {
2225                '(1/2)': '(PS/SB)',
2226                '1': round(mean(welding1_ps.process_memory)) if welding1_ps.process_memory else
                        0,
2227                '2': round(mean(welding1_sb.process_memory)) if welding1_sb.process_memory else
                        0,
2228                'Total': round(mean(welding1_ps.process_memory + welding1_sb.process_memory)) if
                        (welding1_ps.process_memory or welding1_sb.process_memory) else 0,
2229                'Unit': 's'
2230            },
2231            'Welding 2 process duration': {
2232                '(1/2)': '(PS/SB)',
2233                '1': round(mean(welding2_ps.process_memory)) if welding2_ps.process_memory else
                        0,
2234                '2': round(mean(welding2_sb.process_memory)) if welding2_sb.process_memory else
                        0,
2235                'Total': round(mean(welding2_ps.process_memory + welding2_sb.process_memory)) if
                        (welding2_ps.process_memory or welding2_sb.process_memory) else 0,
2236                'Unit': 's'
2237            },
```

```
2238            'Welding 3/4 process duration': {
2239                '(1/2)': '(PS/SB)',
2240                '1': round(mean(welding34_ps.process_memory)) if welding34_ps.process_memory else
                        0,
2241                '2': round(mean(welding34_sb.process_memory)) if welding34_sb.process_memory else
                        0,
2242                'Total': round(mean(welding34_ps.process_memory + welding34_sb.process_memory))
                        if (welding34_ps.process_memory or welding34_sb.process_memory) else 0,
2243                'Unit': 's'
2244            },
2245            'Firing line process duration': {
2246                '(1/2)': None,
2247                '1': None,
2248                '2': None,
2249                'Total': round(mean(firingline.process_memory)) if firingline.process_memory else
                        0,
2250                'Unit': 's'
2251            }
2252        }
2253        df_station_report = pd.DataFrame(env.station_report).T
2254        return df_station_report
2255
2256    def reset_data():
2257        env.data = {
2258            'PTC delivery': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2259            'Marked': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2260            'Cleaned': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2261            'Sent to hold': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': '
                    joints'},
2262            'Sent to hold ratio': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit':
                    '-'},
2263            'DJF production': {'(1/2)': '(PS/SB)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints'},
2264            'DJF rejects': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit': 'joints
                    '},
2265            'Firing line rejects': {'(1/2)': '(Normal/Anode)','1': 0, '2': 0, 'Total': 0, 'Unit':
                     'joints'},
2266            'Main pipeline length': {'(1/2)': None,'1': None, '2': None, 'Total': 0, 'Unit': 'm'
                    },
2267        }
2268
2269        for component in [ptc_ps, ptc_sb]:
2270            component.wait_memory = []
2271        for component in [djf]:
2272            component.wait_memory_ps = []
2273            component.wait_memory_sb = []
2274            component.real_wait_memory_ps = []
2275            component.real_wait_memory_sb = []
2276        for component in [waitingarea, firingline]:
2277            component.wait_memory_normal = []
2278            component.wait_memory_anode = []
2279
2280        for station in [ptc_ps, ptc_sb, bevelling_psa, bevelling_psf, bevelling_sba,
                bevelling_sbf, welding1_ps, welding1_sb, welding2_ps, welding2_sb, welding34_ps,
                welding34_sb, firingline]:
2281            station.process_memory = []
2282
2283    data_reports = []
2284    wait_reports = []
2285    station_reports = []
2286
2287    def save_data(datafile, sheet_name):
2288        file_path = f'output/Output data - {title}.xlsx'
2289
2290        if os.path.exists(file_path):
2291            with pd.ExcelWriter(file_path, engine='openpyxl', mode='a', if_sheet_exists='replace'
                    ) as writer:
2292                datafile.to_excel(writer, sheet_name=sheet_name, index=True)
2293        else:
2294            with pd.ExcelWriter(file_path, engine='openpyxl') as writer:
2295                datafile.to_excel(writer, sheet_name=sheet_name, index=True)
2296
```

```
    if sheet_name == 'Result' or sheet_name == 'KPI':
        wb = openpyxl.load_workbook(file_path)
        result_sheet = wb[sheet_name]
        result_sheet.title = f'{sheet_name} Seed{seed}'
        wb._sheets.remove(result_sheet)
        wb._sheets.insert(0, result_sheet)
        wb.save(file_path)

    print(f'Data saved to {file_path} in sheet {sheet_name}.')

def mean_data(reports):
    reports_clean = [df.replace({None: np.nan}) for df in reports]
    concat_df = pd.concat(reports_clean)
    numeric_cols = concat_df.select_dtypes(include=[np.number]).columns
    mean_df = concat_df[numeric_cols].groupby(concat_df.index).mean()
    for col in reports_clean[0].columns:
        if col not in mean_df.columns:
            mean_df[col] = reports_clean[0][col]
    mean_df = mean_df[reports_clean[0].columns]
    mean_df = mean_df.reindex(reports_clean[0].index)
    return mean_df

def round_data(df):
    rows_to_round_1 = ['DJF rejects', 'Firing line rejects']
    rows_to_round_2 = ['Sent to hold ratio']
    rounded_0 = df.drop(rows_to_round_1 + rows_to_round_2).round()
    rounded_1 = df.loc[rows_to_round_1].round(1)
    rounded_2 = df.loc[rows_to_round_2].round(2)
    df_rounded = pd.concat([rounded_0, rounded_1, rounded_2]).reindex(df.index)
    return df_rounded

# endregion

# region Simulation setup

start_time = time.time()
print("Simulating warm-up period...")
env.run(till=warm_up_period)
reset_data()
data_report_start = data_report()
wait_report_start = wait_report()
station_report_start = station_report()
df_start = pd.concat([data_report_start, wait_report_start, station_report_start])
save_data(df_start, 'Start')

if animation is True:
    env.animate(True)
    env.show_fps(False)
    env.animation_parameters(
        width=1600,
        height=400,
        speed=animation_speed,
        x0=-200,
        y0=-80,
        background_color='white',
        show_time=False
        )

    snapshot_folder = f'output/snapshots/{title}'
    os.makedirs(snapshot_folder, exist_ok=True)
    env.snapshot_folder = snapshot_folder
    snapshot_path = os.path.join(env.snapshot_folder, f'{title} - Start.png')
    env.snapshot(snapshot_path)

for period in range(periods):
    print(f'Simulating period {period+1}...')
    start_period = time.time()
    env.run(till=(period+1)*period_duration + warm_up_period)
    data_report_period = data_report()
    wait_report_period = wait_report()
    station_report_period = station_report()
```

```
2368        data_reports.append(data_report_period)
2369        wait_reports.append(wait_report_period)
2370        station_reports.append(station_report_period)
2371
2372        print(f'Period {period+1} finished in {round(time.time() - start_period)} seconds.')
2373        print("DATA REPORT")
2374        print(data_report_period)
2375        df_period = pd.concat([data_report_period, wait_report_period, station_report_period])
2376        sheet_name_period = f'Period {period+1}'
2377        save_data(df_period, sheet_name_period)
2378        reset_data()
2379
2380        if animation is True:
2381            snapshot_path = os.path.join(env.snapshot_folder, f'{title} - Day {period+1}.png')
2382            env.snapshot(snapshot_path)
2383
2384    end_time = time.time()
2385
2386    simulation_time = int(end_time - start_time)
2387    print(f'Simulation finished in {simulation_time} seconds.')
2388    print(f'Using seed: {seed}')
2389
2390    data_report_result = round_data(mean_data(data_reports))
2391    wait_report_result = mean_data(wait_reports).round()
2392    station_report_result = mean_data(station_reports).round()
2393    df_result = pd.concat([data_report_result, wait_report_result, station_report_result])
2394
2395    kpi_data = data_report_result.loc[['DJF production', 'Main pipeline length']]
2396    kpi_wait = wait_report_result.loc[wait_report_result.index.str.contains('wait time', case=
         False)]
2397    kpi_report = pd.concat([kpi_data, kpi_wait])
2398
2399    print(f'KPI report for seed {seed}:')
2400    print(kpi_report)
2401
2402    save_data(kpi_report, 'KPI')
2403    save_data(df_result, 'Result')
2404
2405    # endregion
```

G.4. Run script

Listing G.4: Run Python Script

```
1  import os
2  import secrets
3
4  script = 'main.py'
5
6  os.environ['PSV'] = 'PS'
7  routes = ['a', 'b', 'c', 'd', 'e', 'f']
8  ptc_rates = ['slow','avg', 'fast']
9  os.environ['DJF_SUPPLY_LINE'] = 'True'
10 os.environ['ANIMATION'] = 'True'
11 os.environ['LIVE_ANIMATION'] = 'False'
12 os.environ['VIDEO'] = 'False'
13
14 for route in routes:
15     os.environ['ROUTE'] = route
16     for ptc_rate in ptc_rates:
17         seed = secrets.randbits(32)
18         os.environ['SEED'] = str(seed)
19         os.environ['PTC_RATE'] = ptc_rate
20         print(f'Running {script} for Route ({route}) with {ptc_rate} PTC rate ...')
21         os.system(f'python {script}')
22
23 print('ALL RUNS COMPLETED')
```