Master Thesis

Design and Implementation of Standard Diagnostics Interface for Railway Control-Command and Signalling: OPC UA Server Approach Based on EULYNX Field Elements

Author:	Peeranut Noonurak
Matriculation No.:	7023582
Course of Studies:	M.Eng. in Industrial Informatics
Academic Supervisor:	Prof. Dr.-Ing. Armando Walter Colombo
First Industry Supervisor:	Ralph R. Müller
Second Industry Supervisor:	Ibtihel Cherif, M.Eng.
Third Industry Supervisor:	Prof. Dr.-Ing. Karl-Albrecht Klinge
Submission date:	29.11.2024

University of Applied Sciences Emden/Leer · Faculty of Technology ·
Department of Electrical Engineering and Computer Science
Master Degree in Industrial Informatics
Constantiaplatz 4 · 26723 Emden · http://www.hs-emden-leer.de

&

DB InfraGO AG · Adam-Riese-Str. 11-13
60327 Frankfurt am Main · https://www.dbinfrago.com/

Declaration of authorship

I hereby declare that I, the undersigned, am the sole author of this document, All sources consulted for this document have been listed: all quotations from and references to these sources have been properly cited and included in chapter notes and in the list of references. No version of this document, either in whole or any section of it, has been used to achieve an academic degree or any other examination.

I understand that any false statements made in this declaration may be punishable by law.

_____

Date Signature

Abstracts

This master thesis describes the design and implementation of a standardized diagnostic interface for railway infrastructure, specifically for the Command Control and Signaling (CCS) system. This project, developed as a proof of concept for the EU-Rail and EULYNX Interface Specification's Standard Diagnostics Interface (SDI) in collaboration with Europe's Rail System Pillar, illustrates the potential of standardising diagnostic models across railway components. The project integrates OPC UA technology, translating UML-based diagnostic structures into an OPC UA Information Model conforming to SDI Specification. Real-time diagnostics for field elements are supported by this implementation as a practical example of moving theoretical standardization to actual application.

The system is containerized using Docker and orchestrated with Kubernetes and Helm resulting in scalability, high availability as well as efficient use of resources. Multiple OPC UA server instances were deployed and demonstrate interoperability through a shared namespace (Namespace 1) and manufacture specific customization (Namespace 2). Deployment is automated and kept up to date with a CI/CD pipeline. Validation testing demonstrated that the system met functional, nonfunctional, technical, and operational requirements for real time data collection, connectivity, and simulation based scenario testing. Connectivity with a Maintenance and Data Management (MDM) system developed in a parallel thesis project validated interoperability, EULYNX diagnostic framework alignment, and dynamic updates through namespace aggregation.

This thesis was part of the demonstration "End-to-End Data Process for a Digital Railway" at the Europe's Rail Joint Undertaking System Pillar booth at InnoTrans 2024 (September 23-27, Messe Berlin). In particular, the thesis was contributed to the Operation & Diagnosis phase and demonstrated the possibility of using OPC UA server features for real-time monitoring and diagnostics compatible with the EU-Rail and EULYNX Interface Specification. Through showing critical and non critical failure scenarios the thesis has shown the practical applicability of the SDI approach towards reliable diagnostics of railway infrastructure.

Due to these limitations the system sets the foundation for further improvements, incuding restricted external access, basic security features, lack of backup and disaster recovery plan. The improvements proposed are cloud hosted deployment, enhanced security, integration of live data sourced from trackside equipment, as well as frequent updates to stay in sync with current SDI specifications. This thesis offers stakeholders a scalable, standardized diagnostic framework and is relevant in the broader context of a Single European Railway Area (SERA) through its contribution to the ongoing digitalization of railway operations.

Furthermore, the most recent version of this document and other resources necessary for this project, such as the code and configuration files that were developed, can also be found on GitHub, and information about the repository structure and its contents are described in details in Section 5.5 - Repository Structure and Setup Configuration in Chapter 5 - Implementation.

Contents

List of Figures

List of Tables

Acronyms

BL4R3 Baseline 4 Release 3

CCS Control Command and Signalling

DB Deutsche Bahn

DICPS Digitalization of Industrial Cyber-Physical Systems

DSD Digital Schiene Deutschland

DSTW Digital Interlocking

ERA European Union Agency for Railways

ERTMS European Rail Traffic Management System

ETCS European Train Control System

EU-Rail Europe's Rail Joint Undertaking

GHCR GitHub Container Registry

ICPS Industrial Cyber-Physical Systems

IDTT Industrial Data Transport Technologie

K8s Kubernetes

MDM Maintenance and Data Management

OPC UA Open Platform Communications Unified Architecture

SD2 Sub-domain 2

SDI Standard Diagnostics Interface

SERA Single European Railway Area

TCCS Transversal Command and Control Systems

UML Unified Modeling Language

UNISIG Union Industry of Signalling

1. Introduction

1.1. Problem Statement

The railway industry is under pressure in several areas related to interoperability and diagnosis of the control-command and signaling systems. Given proprietary interfaces and disconnected systems cause inefficiencies, high maintenance and repairs take time to be done. The European Union Agency for Railways (ERA) notes that there is no harmonization when it comes to railway signaling systems; it deserves to be mentioned that this is one of the reasons for increased operational complications and lower efficiency. ERA reference-based accounts for Europe reveal that the efficiency of cross-border rail services hinges on enhancing interoperability, which calls for standardisation of solutions aimed at advancing the reliability of railway construction[1].

Besides, challenges that include infrastructure as well as costs are realised during implementation of modern signaling systems like the European Rail Traffic Management System (ERTMS). Some of the issues include realization of interfaces between the different national systems, conversion of existing installations and keeping operationally ready during migration from traditional signaling systems to digital systems. Details have been put forward by ERA regarding such issues and the emerging need to enhance the reliability of railway systems by formulating standards. Solving these problems with a standard diagnostics interface can greatly improve the productivity and stability of railway systems [2].

1.2. Objectives of the Thesis

The primary objective of this master's thesis is to investigate, design, and implement a Standard Diagnostics Interface (SDI) for Control Command and Signalling (CCS) based on field elements. Consistent with the Europe's Rail Joint Undertaking (EU-Rail) and its System Pillar Project objectives, this effort is a joint activity with Deutsche Bahn InfraGO. The objective is to improve interoperability, efficiency and the reliability of railway infrastructure. [3].

The System Pillar serves as the "generic system integrator" for EU-Rail and the architect of the future EU railway system. It provides governance, resources, and outputs to support a coherent and coordinated approach to evolving the rail system and developing a system view based on a formal functional system architecture [4]. By leveraging OPC UA-based server technology and integrating with EULYNX field elements, this thesis aims to overcome the challenges posed by proprietary diagnostics interfaces and disparate systems.

This master's thesis also aligns with the focus areas of Industrial Cyber-Physical Systems (ICPS), Industrial Data Transport Technologie (IDTT), and Digitalization of Industrial Cyber-Physical Systems (DICPS) by examining the integration of these disciplines into railway control-command and signaling systems. The project exemplifies the principles of ICPS, which involve combining physical processes with computational and communication systems to improve system efficiency, reliability, and safety [5]. By adopting a principle of DICPS approaching to railway diagnostics, this thesis contributes to the ongoing digital transformation of the transportation sector, supporting interoperability, automation, and data-driven decision-making in railway operations [6]. The ideology of IDTT, which emphasizes the benefits of OPC UA in terms of straightforward client-server model and the capability to create complex information models, is also integrated [7].

The specific goals outlined below are to achieve this thesis. Each has been carefully determined to address distinct aspects of the railway control-command and signaling systems, focusing on improving interoperability, enhancing system efficiency and reliability, and ensuring adherence to critical industry standards:

- Develop a standardized interface for seamless communication across railway systems.
- Optimise diagnostics performance and accuracy with better handling of data.
- Develop robust monitoring system to monitor it's performace continuously.
- Compliance with EU-Rail and EULYNX Interface Specification Standard Diagnosis Interface (SDI).
- To promote interoperability, automation and data driven decisions.
- Design an information model that is based on the existing industry standards.
- Select basic protocol that support client-server architecture.
- Develop a flexible, salable server solution suitable to current and future needs.
- Enable server adaptability and customization of different field elements.

To realize these objectives the project will be implementing a single solution that will partially follow concept of Industry 4.0 (I40), align with the Europe Rail and System Pillar objectives and facilitates communication between different railway components. The project's aim is to support the ongoing development of Transversal CCS (TCCS) Sub-Domain 2 within the European railway, the promotion of innovation, standardization and sustainable development. Thus, as shown in Figure 1.1, can be presented the generic concept of EU-Rail and EULYNX Standard Diagnostics Interface (SDI).

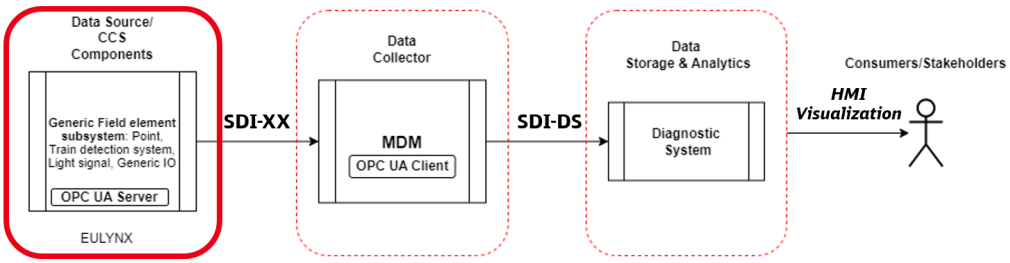


Figure 1.1.: EU-Rail and EULYNX SDI Generic Concept, own illustration

As illustrated in Figure 1.1, the SDI Generic Concept outlines the key components involved:

- **Data Source/CCS Components:** Includes the Generic Field element subsystem, which are points, train detection systems, light signals, generic I/O, and level crossing interfaced through an OPC UA Server.
- **Data Collector:** Represented by the MDM (OPC UA Client), which collects data from the EULYNX Field Elements via the SDI-XX interface.
- **Data Storage & Analytics:** The Diagnostic System stores and analyzes the collected data, connected via the SDI-DS interface.
- **Consumers/Stakeholders:** HMI Visualization to visualize the maintenance team, infrastructure manager, signaling team, etc.

Hence, thesis's scope will focuses on the Data Source/CCS Components, as shown in highlighted red square in Figure 1.1. This includes the Generic Field element subsystem, which consists of points, train detection systems, light signals, generic I/O, and is interfaced through an OPC UA Server.

1.3. Scope of the Thesis

In this thesis, a proof of concept OPC UA based diagnostics interface for EULYNX Interface Specification Standard Diagnostics Interface (SDI) is presented together with a partner EU-Rail System Pillar. Key areas of focus include:

- **Requirements Analysis:** Perform an analysis of the requirements for a standardized diagnostic interface between diagnostic systems and EULYNX field element subsystems. The requirements define object connectivity, key functionalities, communication protocols, and interoperability considerations to match with the EULYNX Interface Specification SDI and EU-Rail System Pillar using UML diagrams.
- **Solution Design:** Design an OPC UA Information Model mapped from UML to conform EU-Rail and EULYNX Interface Specifications SDI, Baseline 4 Release 3 (BL4R3). Develop a schema for modelling diagnostic data for railway field elements by defining nodes, attributes, and relationships.
- **Server Implementation:** Implement an OPC UA server that represent the designed information model, where each node, attribute, and their relationship according to the EU-Rail and EULYNX Interface Specification SDI. Values can also be updated on the server side for testing field element diagnostics with dynamic values. There are two specific Namespaces that represent the generic EULYNX data model and example of manufacturer instances.
- **Testing and Validation:** Carry out experiments for the validation purpose of its features involving the diagnostics interface. The testing includes OPC UA connectivity, information model verification, performance evaluation, and scenario-based simulations using Excel as a data source since there are no physical trackside devices available. Interoperability being proven through integration with the Maintenance and Data Management (MDM) system as an OPC UA client where MDM connects to multiple server instances to prove connectivity and aggregate data.
- **Containerization and Deployment:** Containerize the OPC UA server with Docker and orchestrate it in Kubernetes. Helm charts ease the process of deployment and scaling of applications, enabling multiple server instances with shared generic and custom namespaces. A CI/CD pipeline helps in updating and maintains the system further and proves the concept of scalability for possible future production use.

Consequently, this thesis does not encompass full-scale deployment in live railway infrastructure or the integration of physical hardware components. Testing and validation are performed on a stand-alone machine practically on an independent computer not actual site deployment activities and hardware interfaces. However, the implementation provides a starting point for subsequent integration with actual trackside equipment, and cloud deployment, giving the stakeholders an example, where the standardization processes are moving from being subjects of discussions to actual application.

1.4. Structure of the Report

The structure of this report is organized as follows:

- **Chapter 2: Background** – Brief information about DB InfraGo AG, Digitale Schiene Deutschland, Railway Control-Command and Signaling (CCS) systems, EU-Rail, System Pillar, EULYNX, Standard Diagnostics Interfaces (SDI), OPC UA, RAMI 4.0, Unified Modeling Language (UML), CI/CD, Docker, Kubernetes, and Helm are presented.
- **Chapter 3: Project Management** – Defines specific goals and objectives, detailed specifications and work plan with a traceability matrix.
- **Chapter 4: Detailed Design** – Explains the system architecture and OPC UA Information Model, covering the design of namespaces, object structures, and the process of mapping UML diagrams to the OPC UA model. This chapter also provides integration of EU-Rail and EULYNX Interface Specifications to promote interoperability within railway diagnostics.

- **Chapter 5: Implementation** – Describes the implementation of the OPC UA server, including containerization with Docker, orchestration using Kubernetes, deployment via Helm, and the CI/CD pipeline setup. The simulation of field elements using Excel is also discussed.
- **Chapter 6: Test & Validation** – Discusses the testing methodology, test cases and validation results. It covers OPCUA server connectivity testing, information model verification, scenario simulation testing, MDM system integration, performance assessment and resource utilization. The results show compliance with functional, non functional, technical, and operational requirements.
- **Chapter 7: Conclusion & Outlook** – Conclude and outlook the outcomes and contributions of this proof-of-concept for the EU-Rail and EULYNX SDI and show the move from standardisation to practical implementation. In addition, potential future work areas are proposed including real hardware integration with various domains, cloud based deployment, enhanced security, continuous monitoring and updates to the OPC UA Information Model in accordance with future EU-Rail and EULYNX Interface Specification revisions.

2. Background

This chapter introduces the fundamental components and concepts underpinning the project, including DB InfraGO AG, Digitale Schiene Deutschland (DSD), Control Command and Signalling (CCS), Europe's Rail Joint Undertaking (EU-Rail), the System Pillar, the Transversal CCS Domain, and EULYNX. Besides these things, there are several additional sections including basic concepts like the Generic Diagnostic Concept, Interface Specification SDI, as well as the Point System in Railway Infrastructure, alongside technological foundations (UML, OPC UA, Docker, Kubernetes, RAMI 4.0). This chapter gives an overview of these components and their relation to the project by showing how they enable the implementation and development of a standardized diagnostic interface for railway control and signaling.

2.1. DB InfraGO AG

Deutsche Bahn (DB) InfraGO AG is responsible for Germany's railway infrastructure and its maintenance. The company now manages 33,400 kilometres of rail network and 5,400 stations following the 2024 merger of DB Netz AG and DB Station&Service AG. InfraGO names mean "Infrastruktur GemeinwohlOrientiert" (infrastructure oriented towards the common good) [8, 9].

Responsibilities, Services, and Strategic Focus

DB InfraGO AG specializes in the maintenance, renovation and expansion of railway tracks, stations and signaling systems, to allow for its safe and efficient operation. Key services include:

- **Track Maintenance and Construction:** Regular maintenance and timely upgrades for the network with the aim of ensuring optimal performance and safety of the network.
- **Control-Command and Signaling Systems:** Modernizing CCS systems, including digitalization and ETCS implementation.
- **Station Management:** Keeping and improving of stations to protect safety, to facilitate (i.e. accessibility) and to uphold high service levels.
- **Electric Power Supply:** Reliability of power supply for electric trains and further electrification as part of strategic initiatives.

Role of DB InfraGO AG in Europe's Rail Joint Undertaking (EU-Rail)

EU-Rail is supported by DB InfraGO AG in modernizing and digitalizing railway infrastructure and implementing advanced systems to remove interoperability barriers and building up a Single European Railway Area, known as SERA [10].

Role of DB InfraGO AG in Digitale Schiene Deutschland (DSD)

DB InfraGO AG manages DSD projects aimed at increasing the rail's capacity and reliability, incorporating digital technologies, such as DSTW and ETCS implementation, along with automation of operations and predictive maintenance tasks [11, 12].

In summary, DB InfraGO AG is part of modernizing Germany's railway infrastructure, based on digitalisation and interoperability, and contributes towards increasing capacity, reliability and efficiency following the strategic goals of DSD and EU-Rail.

2.2. Digital Schiene Deutschland (DSD)

Digitale Schiene Deutschland (DSD) is a strategic initiative to digitally transform Germany's rail network by means of digital technologies. Its main goals are doubling long distance passenger numbers in 2030, doubling the market share of freight transport, increasing rail capacity by up 35% without additional tracks. Also, this initiative help boost more efficient and cleaner rail infrastructure, critical to Germany's climate goals [11, 13].

Core Technologies and Key Features

DSD foresees a completely digital railway technique using several technologies to maximise capacity, reliability and security. The core technologies and components of the DSD are illustrated in Figure 2.1. These applications and systems each hold a key role in upgrading the railway infrastructure to be a system that is more efficient, reliable, and modern. Below are brief descriptions of each technology [14, 15]:

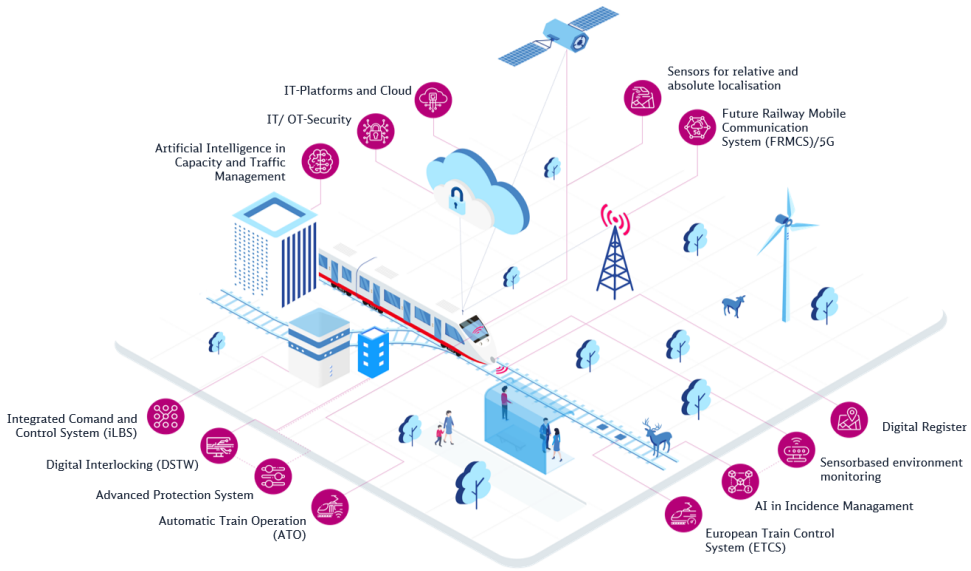


Figure 2.1.: Digital Rail Infrastructure Components, based on [15]

- **IT Platforms and Cloud:** Serves as the infrastructure for digital applications and services.
- **IT/OT Security:** Protects information and operational technologies against cyber threats.
- **AI in Capacity and Traffic Management (CTMS):** Manages capacity and traffic with AI.
- **FRMCS/5G:** Capable of high speed data transmission and highly suited for real time applications.
- **Digital Register (DR):** With digital records help improve asset management.
- **Advanced Protection System (APS):** Safety is increased by Automated protection mechanisms.
- **Automatic Train Operation (ATO):** ATO Increases efficiency and decreases human error.
- **Sensors for Localization:** Gives train positioning at any point, for safe operations.
- **Sensor-Based Environment Monitoring:** Monitors the environment for hazards.
- **AI in Incident Management:** Detects and respond to incidents quickly using AI.
- **European Train Control System (ETCS):** Ensures European networks' interoperability and safety.
- **Integrated Command and Control System (iLBS):** Centralizes functions for better coordination.
- **Digital Interlocking (DSTW):** Replaces traditional interlocking with computerized controls.

Overall, DSD is a big leap towards a green and high capacity rail system, meeting the European standards and innovation in the railway sector [16].

2.3. Control Command and Signalling (CCS)

Railway infrastructure relies on CCS systems to perform the control of trains in a safe and efficient manner. While these systems have been continued improved and a major milestones was the development and implementation of the European Rail Traffic Management System (ERTMS) in the early 1990s. ERTMS was created to prove this railway interoperability within Europe, after the first technical specifications were developed by the Union Industry of Signalling (UNISIG) consortium in 2000 and included in the CCS Technical Specification for Interoperability (TSI) by the European Commission in 2002 [12, 17, 18, 19].

Key Components of CCS

- **European Train Control System (ETCS):** Train operations are optimized and signal safety improved by controlling speed and distance through real time rail operation and tracking systems. ETCS provides interoperability of the European rail networks based on CCS TSI standards [20].
- **Future Railway Mobile Communication System (FRMCS):** Successing Global System for Mobile Communications – Railway (GSMR), high-speed data and advanced signaling called FRMCS is necessary for real-time data exchange being critical for today's railway operation [21, 22].
- **Digital Interlockings (DSTW):** DSTW replace mechanical and relay-based interlockings and helped in enhancing the train movement management, reliability and safety, trained controlled station [23].
- **Automatic Train Operation (ATO):** ATO eliminates manual operations such as starting, stopping, speed control, while increasing efficiency, using less energy and with decreased human error [24].

Implementation Strategy and Challenges

CCS technologies are applied step by step and priority is given to the main rail lines to obtain the early impact on the capacity. The high-traffic corridors are addressed first in the aim of enhancing efficiency in its overall operations. However, the integration of these technologies is a challenge particularly on those that are still in the testing phase and needs a lot of validation. Alleging failure of ERTMS projects and delays and reductions in scope, Trans-European Transport Network (TEN-T) funds have been decommitted by 50% for the initial investment in ERTMS, according to the European Parliament report [25].

2.4. Europe's Rail Joint Undertaking (EU-Rail)

Following the Council Regulation (EU) 2021/2085, the Shift2Rail Joint Undertaking is being performed through the Europe's Rail Joint Undertaking (EU-Rail). Its main goal is to establish an integrated European railway system by developing compatibility specifications and progressing with regard to rolling stock, track, operations services, and traffic management. This strategy is therefore instrumental in the realisation of the Fit for 55 package for the medium term cut in greenhouse gas emissions, the European green deal target of achieving climate neutrality by 2050; and the digital decade initiatives in the railway industry [26].

Key Components and Structure

EU-Rail Research & Innovation (R&I) program is two pillar and one deployment group program with different roles in achieving the overarching mission of EU-Rail [4]:

- **System Pillar**: Concentrates on the synthesis of a unified operational concept and system architecture. The objective of this pillar is to make sure that the whole sector has a functioning system architecture in place, in order to achieve interoperability and efficient system integration.
- **Innovation Pillar**: Research and develop new technologies as well as solution to increase railway's efficiency, capacity and sustainability. It serves as the promoter of innovative projects and technologies.
- **Deployment Group**: Provide the practical implementation of innovations across the rail sector. This Core Group concentrates in the deployment and scaling up of forward looking innovation solutions to realize practicable and timely application of research outputs and technological advancements.

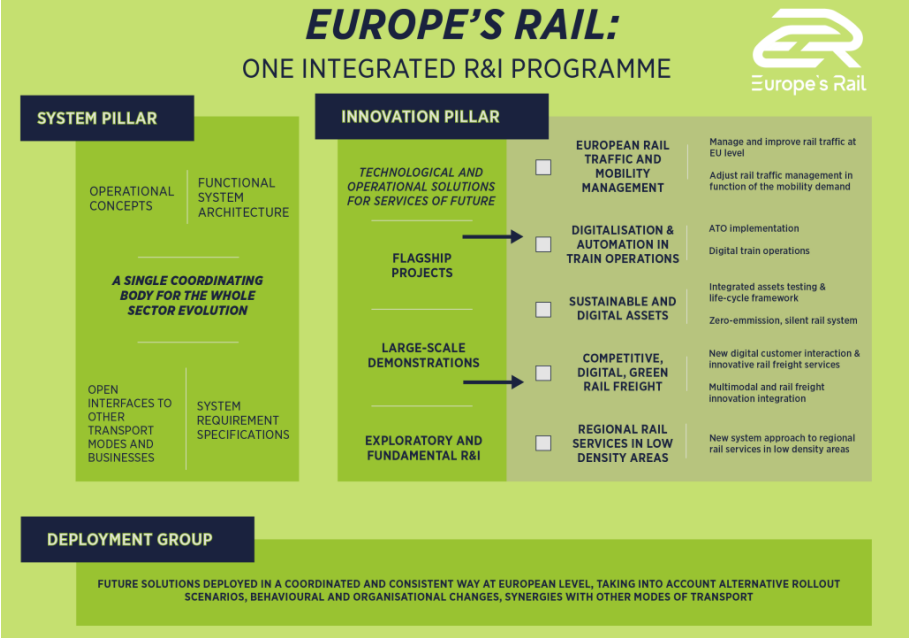


Figure 2.2.: Europe's Rail Key Components and Structure, based on [26]

Mission, Objectives, and Challenges

The figure 2.2 shows that EU-Rail is aiming to build a high capacity, complete integrated European Railway network by removing interoperability barriers and enhancing digitalization and automation. Reducing costs, increasing capacity, and raising network reliability, it was intended to enable the Single European Railway Area (SERA). However, EU Rail faces hurdles, including adaptation to political, technological and market conditions and ensuring the rail system's resilience to growing demand and climate change. To solve these challenges, EU-Rail aims to develop a uniform, efficient European rail area in order to enhance service quality throughout Europe [27].

Results and Benefits

EU-Rail's objectives align with EU policy goals, offering benefits such as [27]:

- **Meeting Customer Needs:** Improve reliability, efficiency and improve user experience in transport services.
- **Boosting Performance & Capacity:** Optimizing performance via interoperability and standardization.
- **Reducing Costs:** Innovative technologies to lower the costs.
- **Supporting Sustainability:** Promoting rail for reducing emissions over road transport.
- **Creating a Flexible System:** Architecting future technologies which are capable of adapting.
- **Strengthening Rail's Role:** Efficiency improvements to support European transport strategies.
- **Increasing Competitiveness:** Enhancing the global competitiveness of the EU rail supply industry.

Overall, EU-Rail aims to make European railways interoperable, innovative and sustainable in order to develop a more integrated, efficient and competitive network meeting customer needs and contributing to the achievement of EU policy objectives

2.5. System Pillar

The System Pillar for the EU-Rail facilitates the development of a common operational concept and overall system architecture and is the "generic system integrator" for Europe's future railway system. It is responsible for delivering governance, resources and outputs in support of a coherent and coordinated approach to the evolution of the rail system. It intends to achieve a consolidated body of different rail sector representatives to facilitate a formal functional system architecture accelerating innovation and network deployment in the European railway network [4].

Importance of the System Pillar

The System Pillar is necessary for the creation of the unified European railway system. Currently, national railways operate independently but at high costs and inefficiently. The System Pillar aims to standardize systems and processes across Europe [4], which will:

- **Reduce Costs**: Decreases the expenses on integration and deployment of new technologies.
- **Accelerate Innovation**: Using one common framework accelerates the pace of new technologies and improves performance, efficiency of railways.
- **Enhance Interoperability**: Supports operation in different countries to offer the integration and streamline of the European rail network.
- **Promote Sustainability**: Encourages the widest possible use of rail services, thus providing a form of sustainable transport solutions.

Goals of the System Pillar

The System Pillar aims at the creation of a single railway system throughout Europe [4]. key include:

- **Unified Operational Concept**: Developing a common framework that contains the essential services, functions, and interfaces, to make sure every parts of the rail system work together to provide solutions, allow smoothly work together.
- **Support for SERA**: Aiding in the building of the Single European Railway Area by setting standards that encourage digitalization and automation, as well as others local transportation.
- **Maintainability and Adaptability**: Making it easy to maintain and update, when technology changes.

For example, the System Pillar diagram in Figure 2.3 illustrates the countries' transition and integration challenges and opportunities within this System Pillar.s. Three countries, A, B and C operate independently their Traffic Management System (TMS) and Trackside Systems (TRK) on the left side. These systems comprise components of the European Rail Traffic Management System (ERTMS), i.e., trackside signaling (ERTMS–TR) and onboard units (ERTMS–OB).

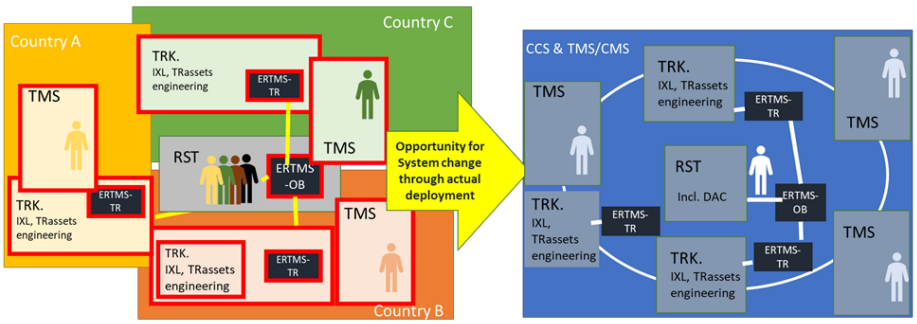


Figure 2.3.: Goals of the System Pillar, based on [4]

This issue of the interoperability arises by the fact that whenever Rolling Stock (RST) travel between countries, It must be compatible with the TMS and TRK system of the corresponding country. On the right in the diagram is control command signing system & traffic management system & Control and monitoring system (CCS & TMS/CMS). This related TMS and TRK components integrated system from different countries ensures more seamless cross border railway operating and supports Single European Railway Area.

System Pillar Organization

EU-Rail's daily operations and achievement of their goals fall under the mandate of the System Pillar Core Group run by EU-Rail Executive Director or designated Unit Heads. This group manages the coordination and advancement of various tasks, with assistance from Engineering and Administrative Services .The organizational structure is divided into core teams that manage different aspects of the System Pillar [28]:

- **Core Group**: In charge of System Pillar's objectives achievement and daily operations delivery.
- **Engineering**: Defines tools and methods for technical support, maintaining architecture catalogs.
- **Administrative**: In charge of the resources, quality control, and economic analysis.

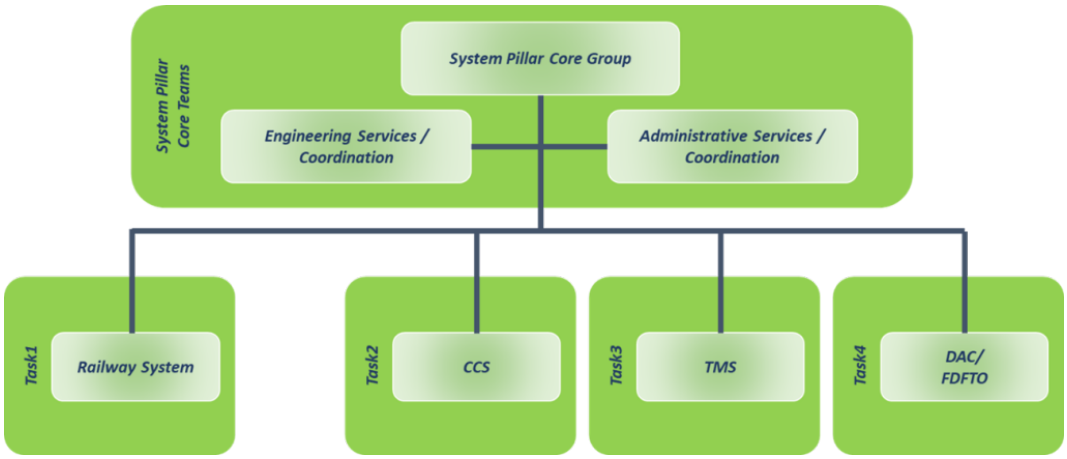


Figure 2.4.: Organizational structure of the System Pillar, based on [28]

The System Pillar is structured into several key tasks, each corresponding to a specific standardization area:

- **Task 1: Railway System**: Defines high level improvements, operational concepts and business process architecture without going into a complete system and sets top level targets for standardization.
- **Task 2: Command and Control Systems (CCS)**: Manages and controls the railway operations developing operational processes, requirements, and architecture for CCS.
- **Task 3: Traffic Management System/Capacity Management System (TMS/CMS)**: Improves efficiency and capacity through TMS/CMS and integrates operational design and architecture.
- **Task 4: Digital Automated Coupling (DAC)**: Sets the standards for DACs, and improves on operational efficiency and interoperability of rail freight, including design and migration processes.

Tasks such as CCS additionally demand significant design responsibilities and thus need to be carried out in a structured framework besides their internal functions. The overall approach divide into the more detailed work done in 'domains' and 'cross cutting roles'.

2.6. Transversal CCS Domain

Transversal Command and Control Systems (TCCS) domain is a part of Europe's Rail System Pillar Project. It is included in Task 2, which focuses on Command and Control Systems (CCS). The TCCS domain aims to standardize and support the needs of various CCS and Traffic Management Systems (TMS). This involves the coordination of engineering data, maintenance and diagnostics with the system and product configuration and assurance. Four main sub-domains consist of these tasks [29].

Scope of Each TCCS Sub-domain

Every TCCS sub-domain performs a specific function, acting on each of the complex parts railway systems, in a structured and standardised way. This ensures that the whole system builds well and the parts work well in the whole system.

- **Sub-domain 1: Engineering and Asset Data, Functional Network Topologies**: This sub-domain establishes standards for sharing engineering and asset data and functional network typologies with other systems and stakeholders. It work with other System Pillar Domain and Integration Programs (IP).

- **Sub-domain 2: Asset Condition Data and Technical Intervention Management**: Establishes standards, interfaces and protocols for obtaining asset condition data from external systems, This data is also provided for CCS systems and defines specifications for an integrated diagnostic system.

- **Sub-domain 3: CCS Configuration Management**: This area concerns methods and protocols for sharing dependable data among system elements. It also defines a standardized process for network level configuration management of CCS systems.

- **Sub-domain 4: Integrated User Interface**: This subdomain defines a system that integrates the various user interface services from different CCS systems. It also brings a framework, tools, and guidance into human and organizational factors.

2.7. Sub-domain 2 (SD2)

As discussed in Section 2.6, Sub-domain 2 (SD2) seek to develop standards and protocols for acquiring asset condition data from outside systems to strengthen CCS systems' diagnostics capabilities . This gives a detailed insight into operational requirements, activities, capabilities to monitor and diagnostics together for the common understanding within CCS system range [29]. The main areas covered in SD2 include:

- **Operational Epics**: These epics are presented as user stories, thereby clarifying roles and requirements of monitoring and diagnostics system in the CCS system. Both trackside and onboard aspects are addressed, supporting the business objectives of the System Pillar.

- **Operational Analysis**: Identifies and describes the key entities and actors participating in the monitoring and diagnostic processes. They define the roles and the functions of the necessary interactions for optimal railway operations.

- **Generic Diagnostic Concept**: Described the phases and data flows which gives a structured framework to monitor and or diagnose. It also includes bottom up initiatives such as EULYNX and the (Open CCS On-board Reference Architecture) OCORA for the purpose of being compatible with what already exist.

These components are essential for improving the process of CCS system monitoring and diagnostics by means of monitoring and diagnostic processes. However, this thesis focuses on detailing the Generic Diagnostic Concept in a later subsection. This concept is a main idea to understanding how diagnostic data can be integrated and used within the CCS framework.

2.8. EULYNX

Launched in 2014, EULYNX is a 15 Infrastructure Managers pooled European initiative to modernize railway signaling through standardized interfaces and modular architectures. It supports the Single European Railway Area (SERA) by promoting interoperability and efficiency across European Rail Networks by allowing the seamless integration of national Control-Command and Signaling systems (CCS) for the sake of safer and more efficient operations. [30].

Purpose and Mission

EULYNX's mission is to standardize the European railway signalling in order to reduce railway infrastructure management costs and complexity. The modular, interoperable interfaces allow for the transition to digital interlockings and provide open, flexible future rail infrastructure.

EULYNX Architecture and Interfaces

EULYNX Reference Architecture provides a signaling system in a modular system with standardized interfaces for efficient communication in European railway networks. Key interfaces include:

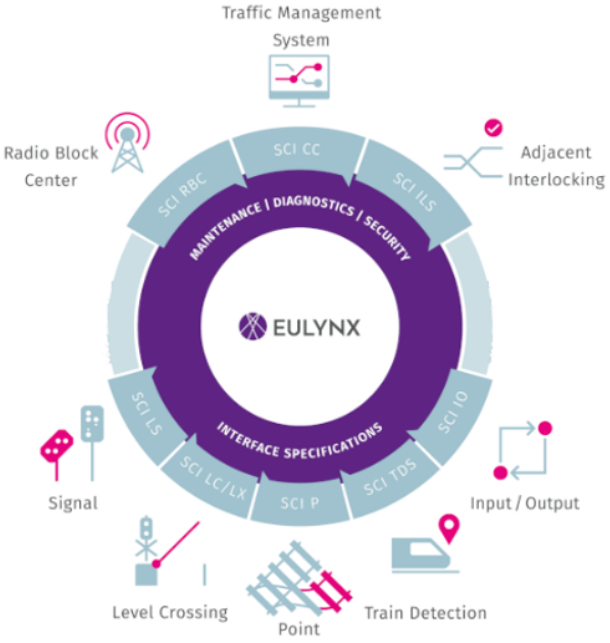


Figure 2.5.: EULYNX System Architecture Overview, based on [30]

- **Standard Communication Interface (SCI)**: Provides reliable command and control within the signalling system by linking electronic interlockings to object controllers.
- **Standard Diagnostic Interface (SDI)**: Integrates diagnostic data to drive maintenance and allows for continuous monitoring and easy identification of issues for trackside components.
- **Standard Maintenance Interface (SMI)**: Streamlines task management enterprise wide by connecting with maintenance services.
- **Standard Security Interface (SSI)**: Performs secure communication for security related functions, protecting the signaling operation from unauthorized access.

As depicted in figure 2.5, the EULYNX system architecture features key interfaces between subsystems such as signals, level crossings, points and detection systems connecting them to improve communications and enable more efficient and safer railway operations.

Collaboration with EU-Rail and System Pillar

As a result of cooperation with the EU-Rail System Pillar, EULYNX further harmonizes its standards with the European rail strategies thus contributing to the further development of the standardized interfaces. The most current Baseline 4 Release 3 (BL4R3), synchronizes the requirements of trackside assets and functions to provide for one integrated digital railway system in Europe. The following standards are already being implemented in several projects, with digital interlocking and other systems improving the interoperability, safety and performance of the European rail networks [12, 31].

2.9. Generic Diagnostic Concept

The Generic Diagnostic Concept in Sub-domain 2 (SD2) is intended to provide common framework of monitoring and diagnostic activities of the CCS system. This concept is used in an effort to achieve interoperability between the various systems to improve the reliability and effectiveness of the railway operations [29].

Concept Overview

In essence, the Generic Diagnostic Concept identifies trackside and on-board subsystems, with emphasis on conformance with existing CCS standards. However, this thesis focuses on trackside elements and, in this regard, the EU-Rail and EULYNX standards for diagnostics and monitoring data gathering through the SDI-XX interfaces are used. This information is handled by the Maintenance and Data Management (MDM) system. Subsequent work will apply these standards to other trackside CCS elements such as interlocking, moving block systems and ATO-TS with the help of other relevant sectors [29].

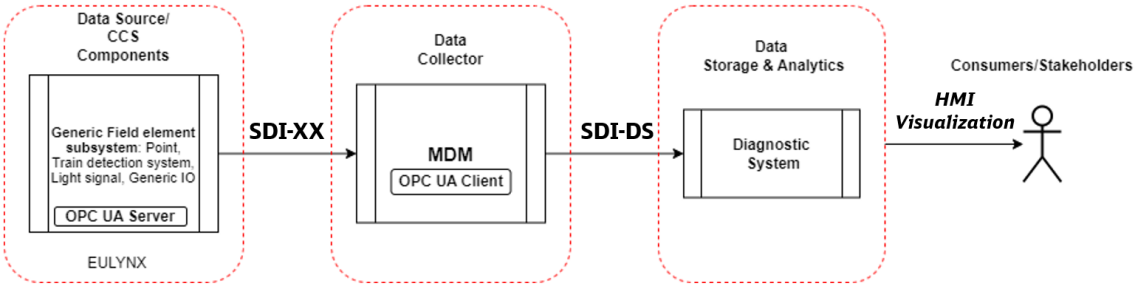


Figure 2.6.: Generic Diagnostic Concept, own illustration

As illustrated in Figure 2.6, the Generic Diagnostic Concept outlines the key components involved:

- **Data Source/CCS Components:** This comprises the Generic Field element subsystem for instance points, train detection systems, and light signals that is interfaced through an OPC UA Server.
- **Data Collector:** This is done through the MDM (OPC UA Client) which acquires information from the EULYNX Field Elements through the SDI-XX interface.
- **Data Storage & Analytics:** The data collected are transferred to the Diagnostic System and stored and analyzed through SDI-DS interface.
- **Consumers/Stakeholders:** These are the maintenance team, the infrastructure manager, the signaling team and other people, who consume the diagnostic data through the HMI Visualization.

Alignment with Standardization Activities

The Generic Diagnostic Concept corresponds to other standards and standardization practices, including EULYNX for trackside applications and OCORA for onboard applications. This alignment ensures that the diagnostic services are in line with recognized frameworks and can easily fit into other CCS sub components.

Data Categories and Standards

Generic Diagnostic Concept distinguishes two main data categories for diagnostics and monitoring purposes:

- **Standardized Data:** This includes data that will become the main focus of the TCCS domain work on detailed specifications. These data sets are important to be observed for interoperability and standardization across different systems.
- **Additional Data Sets:** These are project-specific and will not form part of the standardization efforts within the TCCS domain. Nevertheless, the transport mechanisms for standardized data will enable the transfer of these supplementary data sets.

Trackside Asset Specification

The trackside asset specifications, as detailed by the EU-Rail System Pillar and EULYNX, are encompassed within Baseline Set 4 Release 3 Publication. This release covers essential subsystems like Points, Generic IO, Train Detection Systems, Level Crossing, and Light Signals. Each subsystem includes both generic and specific requirements designed to ensure seamless integration and functionality across the railway network. SDI-P, SDI-IO, SDI-TDS, SDI-LC, and SDI-LS interfaces are used to defined transferred diagnostic data to support the overall objective of standardization and integration within the railway system [31].

Example Implementation

In this thesis, the author aims to demonstrate the Generic Diagnostic Concept through the provision of an exemplary implementation. The implementation will show how points, train detection systems, light signals, and generic I/O from the Generic Field element subsystem are correctly visualized using an OPC UA Information model based on the Trackside Asset Specification (see Subsection 2.9). Also, it will include test cases of real-world scenarios for the point subsystem in the study. This will show the advantages of standardization of interfaces and data to improve the performance of the CCS system.

2.10. Interface Specification SDI

The Interface Specification for the Standard Diagnostic Interface (SDI), as part of the Trackside Asset Specification discussed in Subsection 2.9, defines the protocols and standards that are necessary for communication between diagnostic systems and Command Control and Signaling (CCS) components. This specification thus provides for the proper capture, analysis and application of diagnostic information to improve the overall performance of railway systems.

General Requirements

The SDI is intended to be a message-oriented architecture, which has both a transport layer and an application layer. The transport layer employs the OPC UA protocol with the binary binding through the OPC UA Secure Conversation over TCP for the transfer of diagnostic data. OPC UA protocol follows a very rigid client-server architecture where server runs on the system and the client is a part of diagnostic collector.

Key requirements outlined in the SDI specification include:

- **Communication Protocol:** Data transfer is done over the OPC UA protocol, and thus uses secure and reliable communication. This protocol is structured to support a standardized exchange of diagnostic data in a standardized messaging format.
- **Data Integrity:** The OPC UA server has diagnostic messages structured as data points. Each data point has an attribute such as model type, data type (e.g., Boolean, integer, float) and trigger mechanisms (e.g., event based, condition based). Having these attributes guarantees that all the data is consistent and fairly accurate when it is spread across different systems.

- **Sampling Intervals:** A certain sampling interval for diverse diagnostic data types is specified to ensure they are monitored in a timely manner and in an accurate manner. The intervals are grouped according to the nature of the data, i.e. raw data, diagnostic information, configuration settings, and counters. Data are collected at frequency determined by operational needs.
- **Data Storage:** If communications fail, the specification requires that the diagnostic data be retained on the connected system for a specified period to avoid data loss. The attribute type determines the data storage requirements which may be raw data, diagnostic information, configuration data, counters with rules of mandatory and optional data retention.
- **Attribute Types:** Different attribute types (raw data, diagnosis, configuration and counters) are used to categorize the data points. The handling protocols for each attribute type are specific to the SDI, so that every kind of data is treated accordingly depending on its operational significance.

Telegram Definitions

Communication sessions where many OPC UA specific messages are exchanged between the client and server are called telegrams. These messages are in the OPC UA standard so they can work with different systems. A sequence of diagnostic messages registered, published and responded to by the connected system are contained in each telegram.

Specific Interface Specifications

The SDI includes several specific interface specifications tailored to different CCS components, each covering relevant classes and objects:

- **SDI-Generic:** Outlines generic requirements for SDI-XX communication and technical specifications which are the basis for the subsystems such as Field Elements, Interfaces, and Base Events. This gives a foundation for other SDI specifications, thus guaranteeing that all the systems are well coordinated.
- **SDI-P (Points):** Identifies diagnostic messages and protocols for point systems including Point class, Point Machine class, and Point Turn Event class. This guarantees that data is being collected only for the point operations and this is done efficiently.
- **SDI-IO (Generic I/O):** Specifies the diagnostic interface for generic I/O systems, detailing data points, protocols, and sampling intervals. This includes classes for instance Input, Output, and I/O Event.
- **SDI-LS (Light Signals):** Presents the diagnostic message of light signal systems, such as data synchronization and the message definitions. This includes classes which are Signal, Signal Aspect, Signal Failure Event, and etc.
- **SDI-TDS (Train Detection Systems):** Defines the diagnostic interface for train detection systems, including classes like Track Circuit, Axle Counter, and Train Detection Event.
- **SDI-LC (Level Crossing):** Details the diagnostic message for level crossing systems including the use of protocols and data. These are classes like Level Crossing, Barrier, and Level Crossing Event.

The use of these specific interfaces guarantees that all subsystems in the CCS network is compliant to defined protocols and easily integrated into the whole railway infrastructure in the best possible manner. Consequently, the Interface Specification SDI represents an essential means of standardizing diagnostic data collection and communication in the CCS system. Defining clear protocols and requirements provides that diagnostic processes are consistent, accurate and efficient and adds up to improving the reliability of railway operations as a whole.

2.11. Point System in Railway Infrastructure

Point System is a railway infrastructure product which helps to control and observe moveable track elements for points or switches. ranging from simple points through complex configurations such as slip points and moveable crossings, these elements enable trains to change tracks in safety. The Point System is an integral part of the interlocking system which guarantees that points are coupled in the proper position and locked before a movement in any train. The system relies on the Point Machine's movement of the point blades and its position detection to ensure safe operations. Figure 2.7 illustrates the component Railway point [32].

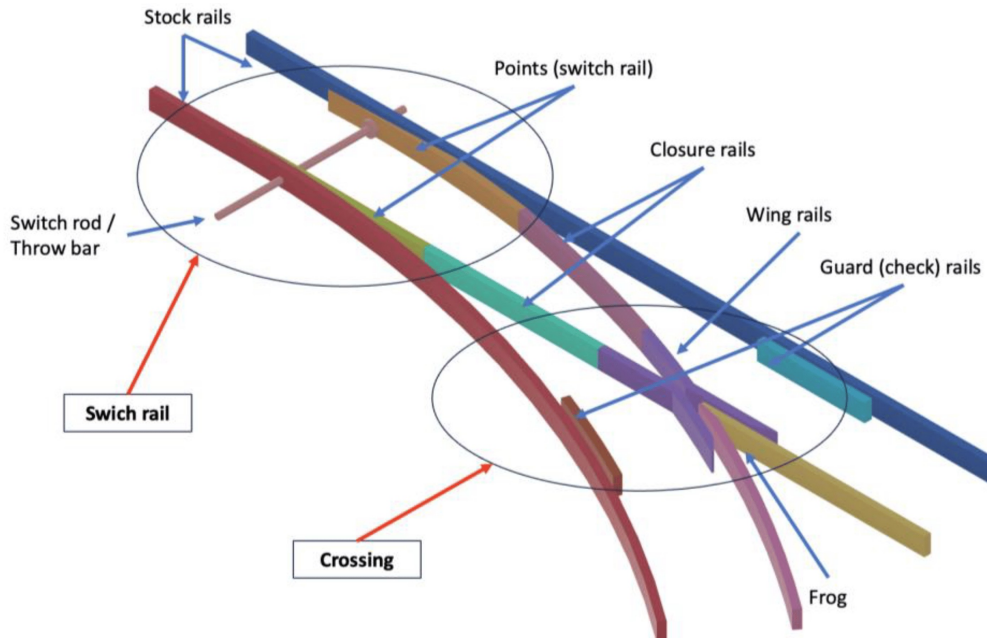


Figure 2.7.: Railway Point Components, based on [33]

Point Machine

The Point Machine is an integral part of the Point subsystem, and it physically moves the point blades and measures the position of the blades. Each Point Machine has two primary functions:

1. **Movement**: Adjusting tracks by physically repositioning the point blades.
2. **Position Detection**: Reporting the current position of the point blades to the interlocking system for monitoring and reporting of the safe operation.

Point machines can be configured in two ways:

- **Point Detector**: A configuration where the point machine measures and reports the position of the point blades only and can not move the point blades.
- **Full Functionality**: A configuration of the point machine where it can both move the point blades as well as detect their position.

A single point may be equipped with one or more point machines. In scenarios where multiple machines are used, some machines may be configured solely as point detectors, providing positional feedback without controlling movement.

A schematic of a 4 wire circuit used for point machines systems is shown in Figure 2.8. It includes four wires (X1, X2, X3, X4), without which control commands cannot be transmitted, and feedback for the position of point blades cannot be received. This setup is integrated into the point machine's internal components (Digit A, Digit B, Digit C, Digit D), as the locking and the point position detection are fundamental for safe and reliable switching operations to be performed.

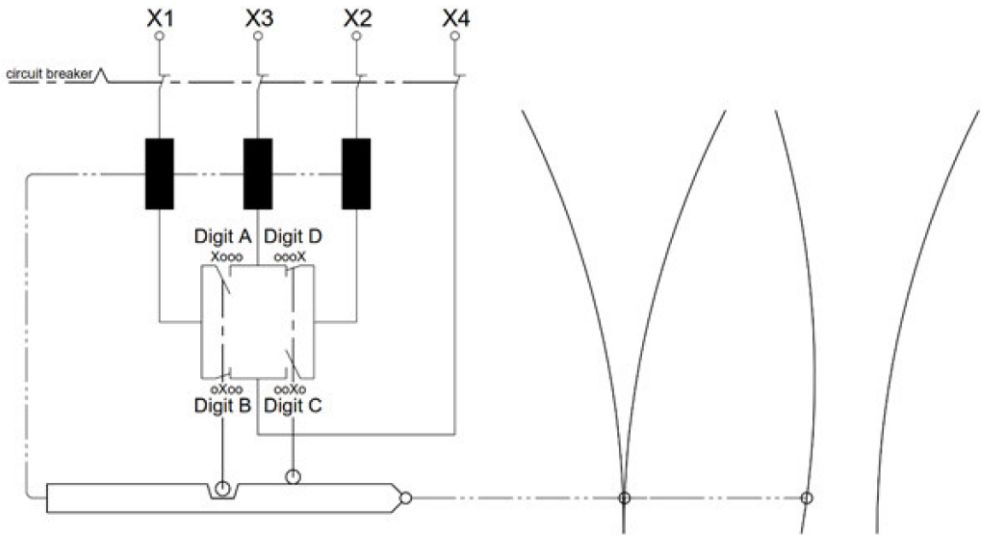


Figure 2.8.: Schematic of a 4-wire circuit for Point Machine control and detection, based on [32]

Point Machine Position

The Point Machine has to know where the point blades are located so it can align the points before any train will move. This subsystem breaks down point machings positioning into major three states:

- **End Position (Left or Right)**: Secure locking indication of point blades in left or right position.
- **No End Position**: This occurs when the point blades are insecure in either one of the end positions.
- **Unintended Position**: Caused by a trailing movement or other disruption, detected when the point blades are not in the commanded position.

Overall Point Position

When the moveable element is provided with multiple point machines, their inputs must be combined into an overall point position which is notified to the interlocking system. This overall position identifies the status of the point, either in a desired or an unintended point location. The interlocking system is updated as soon as the position changes and the subsystem does not have a memory of the reported state.

Degraded Point Position

In cases where the point machine identifies a degraded position, the system may permit use of the point (but with reduced functionality). The point position is classified into two levels of reliability.

- **End Position (Left or Right)**: The moveable element can be used for any operational need: route setting and flank protection.
- **Degraded Position (Left or Right)**: The element is usable for only certain operational requirements, including limited flank protection in degraded conditions.

Safe and efficient operations require the ability of the Point System to manage multiple point positions, including degraded states. In addition, the system is based on a well defined architecture and interfaces with the overall Command Control and Signaling (CCS) infrastructure to ensure these capabilities are successfully integrated into the railway infrastructure.

System Architecture and Interfacing

The Point System is connected to the Command Control and Signaling (CCS) through several interfaces. These are the physical point interface which connects point machines and the functional point interface which connects to the interlocking system. While the functional interface is standardized through EULYNX, the physical interface complies with the national standards.

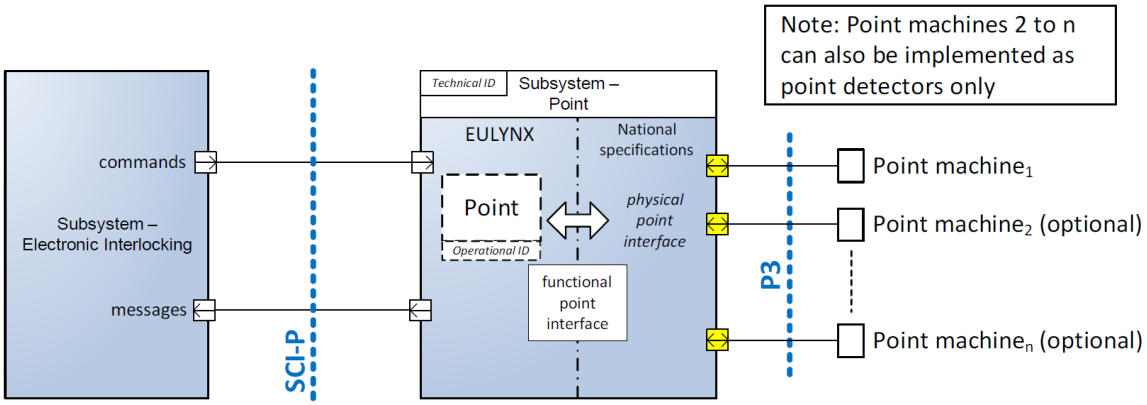


Figure 2.9.: Point subsystem architecture with Electronic Interlocking, based on [32]

Figure 2.9 depicts the architecture of the Point Subsystem and its relationship with the Electronic Interlocking system via the SCI-P protocol. It also shows primary and optional point machines and how they can function as point detectors only. These interfaces guarantee precise control of interlocking commands and proper transmission of status information to the control center.

Precise and reliable operation necessitates integration of the Point System into the the CCS infrastructure. In addition, to maintain this integration, the interface to which it is connected (i.e., the Point System) is required to exchange diagnostic data between its level of control and higher levels of control through the Standard Diagnostic Interface for Points (SDI-P).

Relevance to SDI-P (Points)

The Standard Diagnostic Interface for Points (SDI-P) defines a standardized method of communications of diagnostic data, including movement commands, position detection, and 'error' states between point machines and control systems. This interoperability aim facilitates the detection and the reaction to faults so as to guarantee a safer and more reliable operation. SDI-P is configurable for both, full function and point detect, to meet most needs.

2.12. Unified Modeling Language (UML)

In software and systems engineering, Unified Modeling Language (UML) is a standardized tool for representing system structure, behavior and interactions. It is useful across fields such as railway infrastructure, and provides a visual framework to simplify the design and communication of complex systems. UML diagrams are used in the Model-Based Systems Engineering (MBSE) framework to manage system complexity through the lifecycle mainly in safety critical environments such as railways [34].

2.12.1. UML Relationships

In UML, the system structure as well as interaction between the system are expressed through relationships of system components or classes. Four main types of relationships are as follow [34]:

- **Association:** This is a relationship between at least two classes that means an object of one class is related to an object of another class. Bidirectional or unidirectional, and the multiplicity indicates the number of instances of one class associated with instances of another. For instance, in Deutsche Bahn's Train class might be linked with Deutsche Bahn's Track class, such that a train might travel on several tracks, and each track might accommodate several trains.

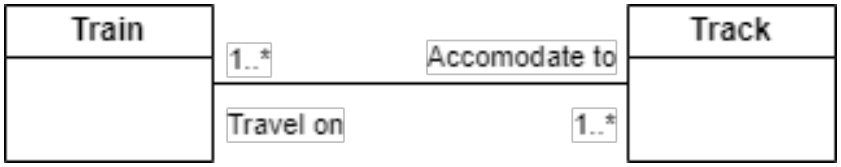


Figure 2.10.: UML Diagram of Association between Train and Track, own illustration

- **Aggregation:** A whole-part relationship is called aggregation. This means that a class (the whole) is comprised of one or more classes (the parts), but those parts do not have to belong to that whole. As such, an InterCity Express (ICE) train class could comprise a composite of Carriage classes, Dining Car classes, First-Class Carriage classes, or Second-Class Carriage classes, which can be assembled or removed based on need for the service, yet maintain independence if taken apart.

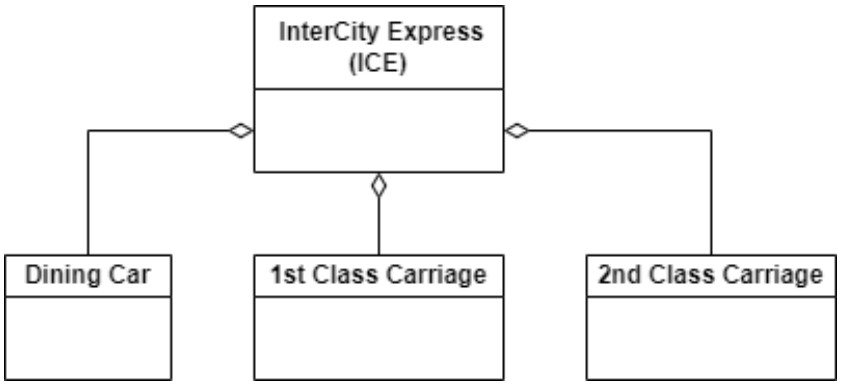


Figure 2.11.: UML Diagram of Aggregation in InterCity Express (ICE) Train, own illustration

- **Composition:** Composition is a form of association that represents a Whole Part relationship in which the part or parts cannot exist without their whole. The whole and its parts are destroyed if any of the two is destroyed. For example, in Deustche Bahn's S-Bahn network Signal Control Unit class, the Signal Lights, Signal Relays and Control Circuits are all parts of a unit and can not be apart from it. These components would also be removed if the control unit is decommissioned. These components would be removed if the control unit was decommissioned.

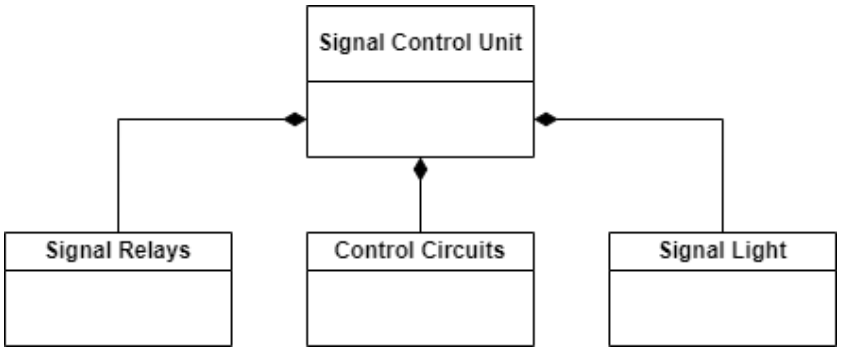


Figure 2.12.: UML Diagram of Composition in Signal Control Unit, own illustration

- **Generalization:** Generalization is an inheritance relation between a superclass, and a subclass. Namely, subclass can be instantiated, while superclass cannot be instantiated on its own. In other words, the superclass attributes can be reused in the subclass. For example, a superclass named Deutsche Bahn Train could be generalized as InterCity Express (ICE), Regional Express (RE), Stadt-Schnellbahn (S-Bahn) subclasses, all of which share common attributes (e.g. speed or route), but have specific features specific to a type of service (e.g. ICE trains are high speed and routes are long distance compared to RE and S-Bahn ones).

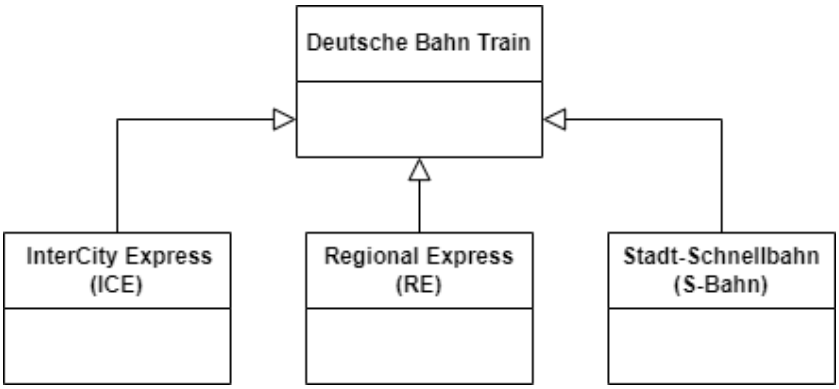


Figure 2.13.: UML Diagram of Generalization in Deutsche Bahn Train, own illustration

These relationships are essential to model how different parts of a system are related, and are thus essential to designing and understanding complicated systems, such as those used by Deutsche Bahn for operations and infrastructure.

UML in SDI Interface Specifications

The UML diagrams are used in the Interface Specification Standard Diagnostic Interface (SDI) to represent the interactions and dependencies of sub systems of railway. These diagrams which include Points, Light Signals, Points and Train Detection Systems and Level Crossings diagrams help in explaining how systems within the Command Control and Signaling (CCS) system interact with each other.

Additionally, UML is applied in the SDI document to present the aspects of architectural structure, process flows, control structures and sub-system operational states. The SDI specifications also make sure that all these aspects are modeled in UML so that all the components fit well into the system and works in the expected way.

2.13. OPC UA Overview

Open Platform Communications Unified Architecture (OPC UA) is a machine to machine communication protocol widely used in industrial automation for the secure and safe exchange of data. OPC UA has been designed as a platform independent framework that facilitates device, systems and application interoperability, allowing seamless integration in the context of a large industrial environment [35].

OPC UA System Environment

OPC UA is generic in its design and applicable across a wide range of different applications in an organization's network. A typical OPC UA system environment is shown in Figure 2.14 which exhibits deployment of the OPC UA servers at various levels of the network architecture such as Corporate Network, Operations Network and Control Network [35].

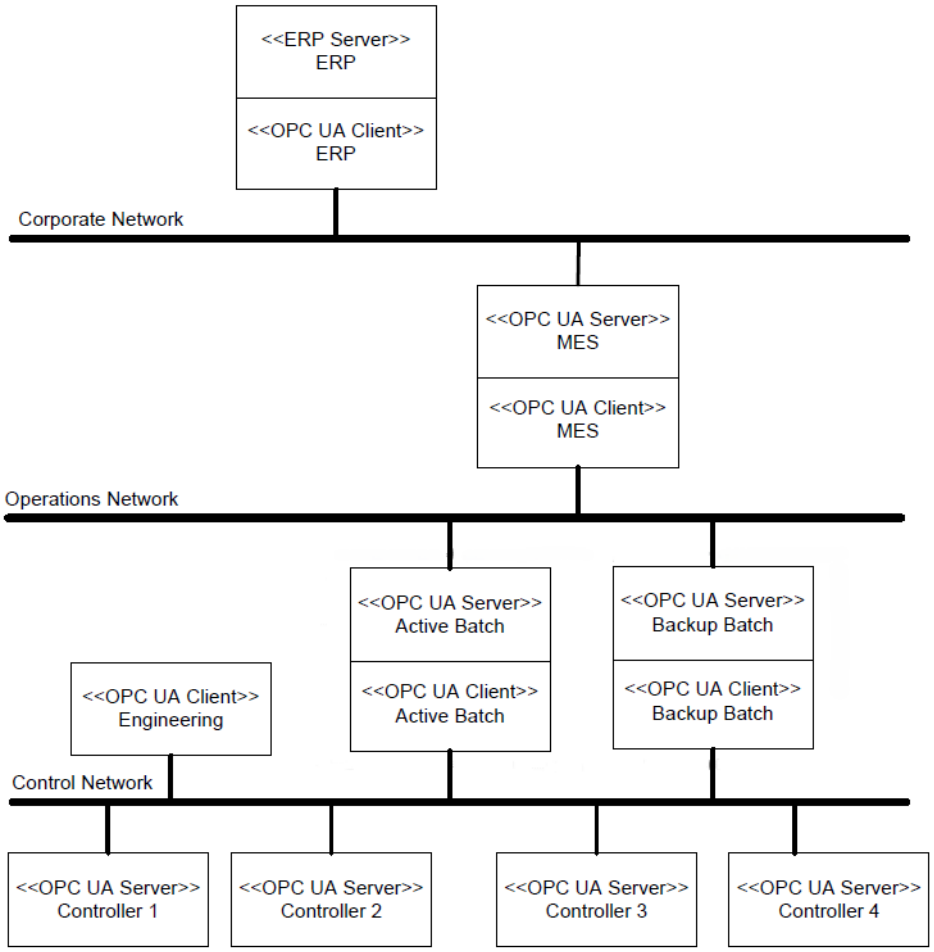


Figure 2.14.: Example of an OPC UA system environment, based on [35].

For instance, OPC UA servers run on controllers in the Control Network, as well as Batch systems in the Operations Network, and are deployed for MES applications, for example for production planning. Moreover, an ERP system employs an OPC UA client as an interface for consuming services within Corporate Network. This multi-layered deployment demonstrates the flexibility of OPC UA in the different operational contexts, using different platforms to satisfy the different needs of different layers.

Client-Server Pattern in OPC UA

In OPC UA systems, the core communication model is a Client-Server pattern, where the server provides the services and the client consumes them to do something specific. Figure 2.15 shows this interaction, which is essential to the collection, processing, and use of data in an OPC UA environment [35].

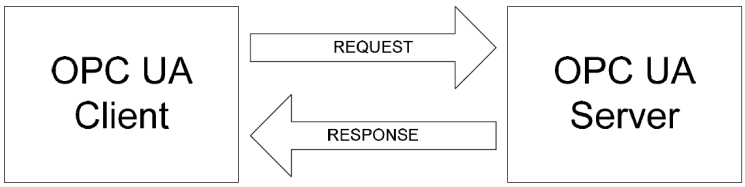


Figure 2.15.: Client-Server pattern in OPC UA, based on [35].

2.13.1. OPC UA Information Models

With its capability of information modeling, OPC UA can represent complex data structures. It represents a structured, extensible, and object oriented method of describing system components and interactions with a standardized method for exchanging meaningful data between clients and servers [35].

Nodes and References are the fundamental building block of OPC UA to represent data. A Node represents an object, variable, method or a Data Type in the system. References describe how Nodes are related to each other, thus building a network of associations that is an organization of the information model. The flexibility of accessing, processing and visualizing the data of this structure is given. The Information Model follows object-oriented principles, utilizing different types of nodes:

- **Object Nodes**: Represent physical or logical components, for example machines or sensors.
- **Variable Nodes**: Store data values, e.g. temperature readings, system status, etc.
- **Method Nodes**: Define functions which can be executed on objects, and thereby interact with them.
- **DataType Nodes**: Define what types of data are in used in the system (i.e. integers or custom structure).

Figure 2.16 shows how data coming from a temperature sensor is displayed as an OPC UA Information Model. In the case of the node "Device A", configuration and measurement data attributes including temperature and engineering unit are organized under the device's address space. This organization of relationship allows a flexible way of managing data across the different systems and devices.

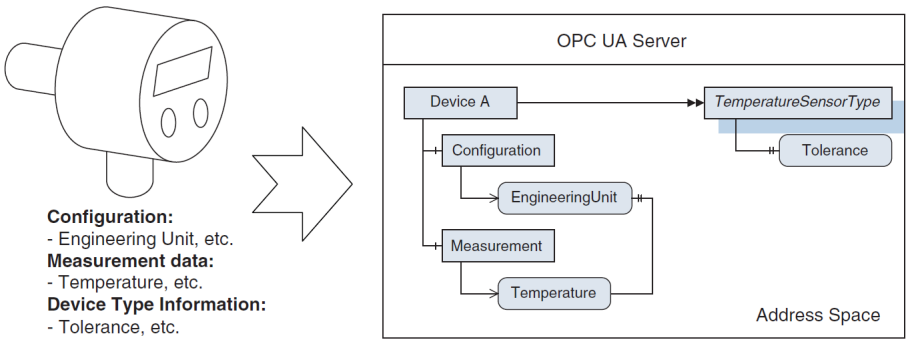


Figure 2.16.: Temperature Sensor Data Integration in OPC UA, based on [35].

Nodes and Address Space

Nodes are the key entities in an OPC UA Information Model, which encapsulates attributes that describe its properties (for instance, a sensor's current measurment). References define how Nodes connect with each other. Address Space is defined as the set of nodes by which the client can navigate the relationship of nodes to form a structured hierarchy.

Types and Extensibility

OPC UA Type Definitions for objects and variables are defined in order to support reusability and flexibility. Type Definitions enable defining custom object type and inheriting attributes and behaviors from base type. For example, Device A is the object instance, that uses TemperatureSensorType in Figure 2.16 as a base containing information specific to attributes of temperature measurement.

Modeling Rules, ValueRank, AccessLevel, and ArrayDimensions

In OPC UA, the information model is further structured with Modeling Rules, ValueRank, AccessLevel and ArrayDimensions:

- **Modeling Rules**: Choose whether nodes are mandatory or optional in the type instances, with Optional and Mandatory Placeholders that guarantee a consistent object instantiation.
- **ValueRank**: Determines if a variable is scalar or array, with how many dimensions the array has.
- **AccessLevel**: Indicates whether clients have read or write access on the node. Interactions with the data are controlled with a number of different levels of access, such as Read-Only, Read/Write, etc.
- **ArrayDimensions**: Size of each dimension is defined and it allows structured data to be represented.

The instantiation and access to nodes is guided by these attributes so that the data is represented clearly and structured in the OPC UA environment.

OPC UA in the Context of This Thesis

This thesis focuses on the use of OPC UA as a means to manage diagnostic data from railway infrastructure such as the Point Product Group Set (switches and point machines). OPC UA servers, representing trackside assets, are containerized using Docker, orchestrated using Kubernetes, and managed by Helm in order to provide high availability and scalability.

Part of the Maintenance and Data Management (MDM) system, the OPC UA client is connected to a number of OPC UA servers that manage trackside assets such as Point Machines, Object Controllers, Light Signals and I/O Controllers. The client collects maintenance and diagnostic data efficiently, from standardized interfaces (SDI-XX), in order to analyze system, for maintenance and predictive maintenance.

Collected data is then processed and stored in the Diagnostic System for view and analysis through Human Machine Interface (HMI) systems by stakeholders. The OPC UA Information Model represents the structure and behaviour of the assets and allowed for communication between OPC UA servers and MDM system.

2.14. Continuous Integration / Continuous Deployment (CI/CD)

Continuous Integration and Continuous Deployment (CI/CD) refers to a collection of automated practices of software engineering that shorten the development phase of applications followed by testing and deployment. Reducing development cycles and increasing software quality by accelerating and improving code change delivery speed and reliability is the key [36].

Concept Overview

The CI/CD methodology consists of two core practices combined together.:

1. **Continuous Integration (CI):** Developers merge their code change into a shared repository, then provokes an automated build and testing process. Key principles of CI include:
   - Automated compile and dependency resolution build pipelines.
   - Tests to verify changes against existing functionality automatically.
   - Integration with regular code so that there will be fewer conflicts.
2. **Continuous Deployment (CD):** Developers automate the step of deploying tested code to production or staging environment. This ensures that all changes are tested, and the manual intervention is minimal. Key principle of CD includes:
   - Automated deployment scripts which handle the release process.
   - Ability to track changes and do rollbacks.
   - Using tools for monitoring to assure reliable deployments.

Importance in Modern Systems

This ensures that CI/CD fixes problems that are inherent to classical development and deployment practices.

- **Faster Time-to-Market:** Less delay, faster feature and fix deployment.
- **Improved Code Quality:** Continuous testing stops issues swiftly, preventing production bugs.
- **Scalability:** Multi service systems are handled by automated pipelines via consistent process.

CI/CD in the Context of This Thesis

In this thesis, CI/CD pipelines is applied to automate the build, testing, and deployment of containerized OPC UA servers. The CI/CD processes are managed through GitHub Actions, assuring easy, reliable and efficient code integration. Key aspects of CI/CD implementation are:

- **Automated Builds:** Code commits are built to Docker images and tag the image.
- **Automated Testing:** OPC UA servers are verified to function by use of Unit tests.
- **Deployment Automation:** Servers are deployed as Helm and Kubernetes configurations.
- **Version Control and Rollback:** Error recovery is done via version control for each deployment.

Through the integration of CI/CD pipelines, the thesis shows an effective, scalable, and manageable deployment workflow for the OPC UA based diagnostics system. This corresponds to the project's operational requirements and allows for high availability, fast updates, reduced downtime.

2.15. Docker and Containerization

Docker is open source deployment platform which wraps an application and its dependencies in a lightweight container. These bring together the application, its dependencies and make it easier to run consistently on different environments. When using this approach, the system configuration discrepancies are reduced and the deployment process is simplified [37].

Docker Images and Containers

A Docker image is a read only blueprint for a container including instructions on how to create that image. It contains the application's code, libraries, and dependencies the application needs. Repositories allow us to store and share docker images which are then reusable and version controlled [37].

On the other hand, a Docker container is an executable instance of a docker image. It basically means that each container runs in its own environment, having their own file system, processes, and Network configuration, while the only thing it shares is the Kernel of the host machine. This helps to provide isolation so that containers will operate the same everywhere they run.

Containers have a transient nature, created, stopped and removed quickly, without affecting the host system. Because of this flexibility they are suited to scenarios where scalability, resource efficiency and predictable application behavior are important.

2.15.1. Docker Architecture

As shown in Figure 2.17, Docker uses a client server architecture. Docker daemon handles building, shipping and operating container, while the Docker client is used to transmit commands to the Docker daemon. The client and daemon communicate via a REST API either over UNIX sockets or network interfaces. It works whether the client and daemon are in the same machine or connected remotely and this provides flexible container management [37].

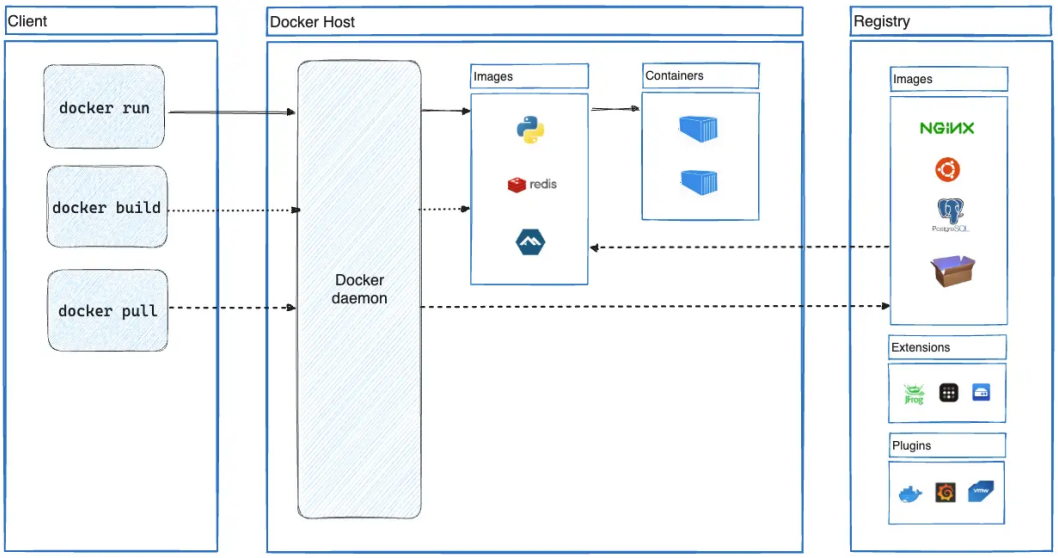


Figure 2.17.: Docker architecture showing client-server interaction, based on [37]

Docker daemon is responsible for core container management duties, image creation, container launching and resource allocation. It will listen for API requests for performing container lifecycle operations. Besides the client server model, Docker also offers a Docker Compose a tool for working with multi-container applications. Developers can define the services, networks and volumes required by an application using a configuration file, thereby letting Docker Compose take care of complex system orchestration.

Docker in the Context of This Thesis

For this thesis, Docker will be used to containerize the OPC UA servers that manage the trackside assets (Point Machines and Object Controllers). By providing containerization, Docker makes the deployment process simple by encapsulating servers along with their set of dependencies inside the container and ensures consistent and reliable deployments across many environments. The diagnostic data collected from trackside assets is managed in a uniform way across the various underlying infrastructure.

Since the system aim to scales to many OPC UA servers, managing individual containers manually becomes infeasible. As a solution to this, container orchestration tool Kubernetes will be used to automatically deploy, scale and manage several Docker containers. Kubernetes keeps it flexible, customizable and can easily handle the high availability or fault tolerance across numbers of instances. Also, Helm will be used to manage and configure deployment of these containers onto Kubernetes to ensure the swift deployment of multiple instances of the same OPC UA image.

Through adoption of Docker for containerization and Kubernetes for orchestration, the system guarantees scalability and consistent operation, especially in dealing with increasing complexities of railway infrastructure. The combination together is able to manage and monitor diagnostic data efficiently, enabling seamless deployment and maintanence of a large number of OPC UA servers.

2.16. Kubernetes and Helm

2.16.1. Kubernetes (K8s)

Kubernetes is an open source technology that automatically manages the deployment, scaling and management of containerized applications across distributed environments. It organizes applications (in terms of Pods, the smallest deployable units) which comprises one or more containers. Across a cluster of nodes (physical or virtual machines), these pods are managed [38].

Docker builds and runs containers, but Kubernetes orchestrates and scales the containers across distributed environments, making sure they're available and being used efficiently. It automatically balances workloads, manages container's lifecycle and reschedules or restarts containers if they fail. Additionally, Kubernetes provides mechanisms to scale applications per demand, and with guarantee of resilience and resource optimisation. Key components, such as Pods, Nodes, Services, and Deployments, work together to manage these containerized applications:

- **Pods:** The basic building block for deploying applications on K8s. They consist of one or more containers which share storage and networking resources, and which are configured to run based on a specification.
- **Nodes:** A set of physical or virtual machines that act as the infrastructure on which Kubernetes hosts containerized applications controlling the distribution of pods and resources across the cluster.
- **Services:** An abstraction layer exposing a group of pods that communicate as a single network service.
- **Deployments:** Kubernetes controllers take care of desired number of pod replicas, making them available and taking care of tasks like roll update or rollback during the application upgrade or failure.

In Figure 2.18, the overview of Kubernetes architecture is described at a high level showing the interactions between control plane, nodes and pods in the Kubernetes Cluster.

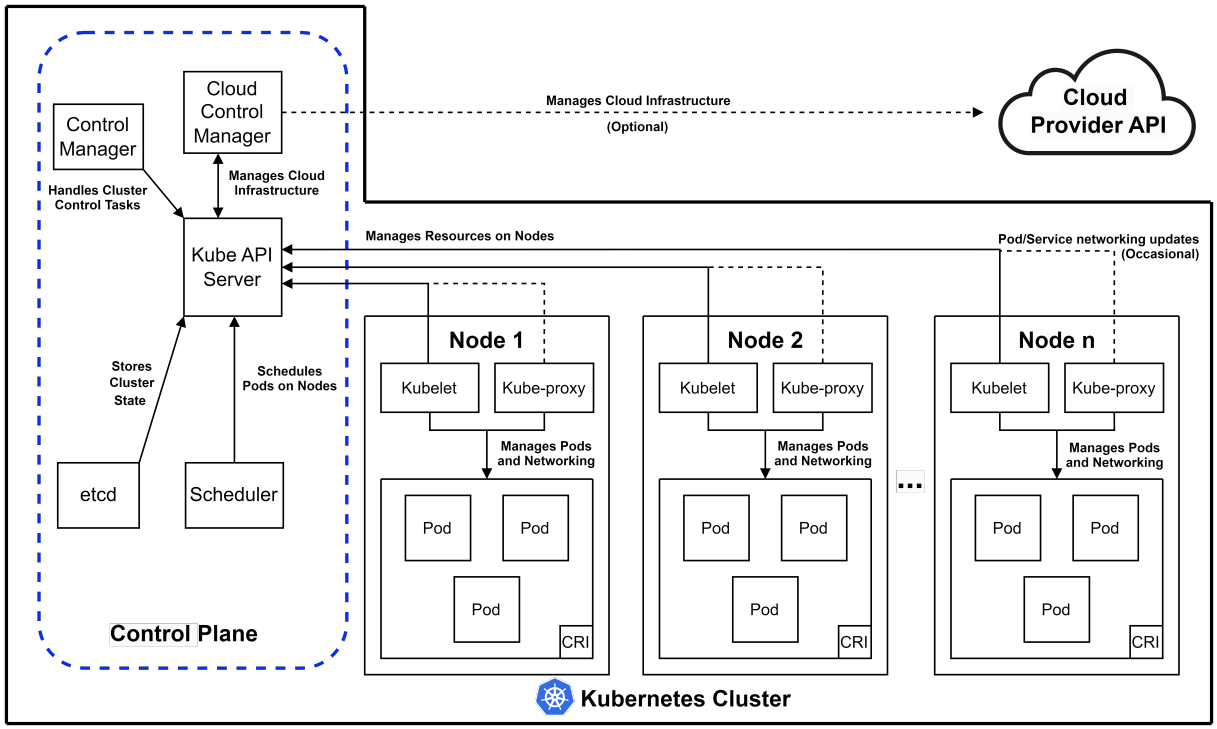


Figure 2.18.: Kubernetes Cluster Architecture, own illustration

Control Plane

The central control plane is the management layer of the cluster. Each of these are made up of several different components each responsible for maintaining the desired state of that system:

- **Kube-Api-Server:** Exposed API for Kubernetes. The control plane front end, processing both internal and external requests to mutate the cluster state.
- **Etcd:** Used to persist all cluster data (configurations, secrets, and deployed workloads state) in a distributed key value store. It makes sure that the cluster is consistent.
- **Kube-Scheduler:** Controls the Pod new assigning rule to available Nodes by resource requirements and constraints.
- **Kube-Controller-Manager:** Executes various controllers responsible to keep the cluster in the desired state, for example it makes sure how many replicas of an application should be present.
- **Cloud-Controller-Manager:** Links with cloud providers to handle resources special to be cloud environment (e.g. load balancers and storage). This component will indicate to cloud provider API the way in which cluster can interact with cloud infrastructures such as AWS, Google Cloud or Azure.

The state and performance of the cluster is bound to the communication between the control plane and the worker nodes. Kubernetes API Server is the central hub of all communication, that processes every change to the cluster like scheduling pods or scaling resources. These changes are passed through the API server to maintain cluster level consistency.

Node

The workloads require many key components to be executed, which each Node in the cluster runs a portion of. In the architecture, Nodes 1, 2 and n represent the worker nodes of the cluster, each of the worker nodes run several pods where the actual application workloads are executed. The diagram also shows dotted lines representing these communication pathways, specifically between control plane and node. The primary components present on every node are:

- **Kubelet:** An agent that's running on each node to make sure the containers in a pod are running as the control plane instructs. It communicates directly to kube-api-server to get pod scheduling instructions and report node and pod lifecycle status. Execution of the actual application workloads on each node is important.
- **Kube-Proxy:** Responsible to handle the network routing for pods of the cluster, for providing the communication of one pod to another node or pod, and also, it also manages the network policies. It makes sure network traffic gets routed and evenly divided among the pods and services inside cluster.

Pods

Kubernetes manages Pods, the smallest deployable units in Kubernetes across the nodes. Scheduling these pods is the responsibility of a control plane and the kubelet on the node ensures these pods get to run as specified. Nodes and the control plane communicate over the Kubernetes API and the kube-proxy to continue to provide network policies. Container in each pod typically runs one or more and are managed via the:

- **Container Runtime Interface (CRI):** Makes Kubernetes flexible in working with different container runtimes, thereby how the containers will operate on nodes.

Autoscaling

The Horizontal Pod Autoscaler (HPA) scales pod replicas based on CPU or memory usages. It enables the system to adapt quickly to changes in workload automatically. Autoscale ensures that resources are utilized optimally, mostly in an environment that has fluctuating demand.

Kubernetes has the ability to automate container orchestration processes such as autoscaling doing which makes it a feasible solution to manage complex, large scale deployment. For the purpose of this thesis, Kubernetes is used to manage deployment of many OPC UA servers to scale, have high availability, and optimize resource utilization within the infrastructure. In addition to this, Helm makes it easier to deploy and manage Kubernetes resources with pre configured charts that make it easy to automate many of the complexities that may arise in multi server environments.

2.16.2. Helm

Helm is a package manager for Kubernetes; it configures and runs applications (called Helm Charts) by grouping together Kubernetes objects in one template. Helm is an intermediary between users and Kubernetes that helps users to configure and deploy complex systems with a few simple commands [39].

Key Components of Helm

Figure 2.19 describes Helm architecture showing how the resources like Helm Chart, Helm CLI and the Kubernetes control plane communicate to facilitate application deployment in a convenient and flexible manner. The key components of Helm architecture comprises of [39]:

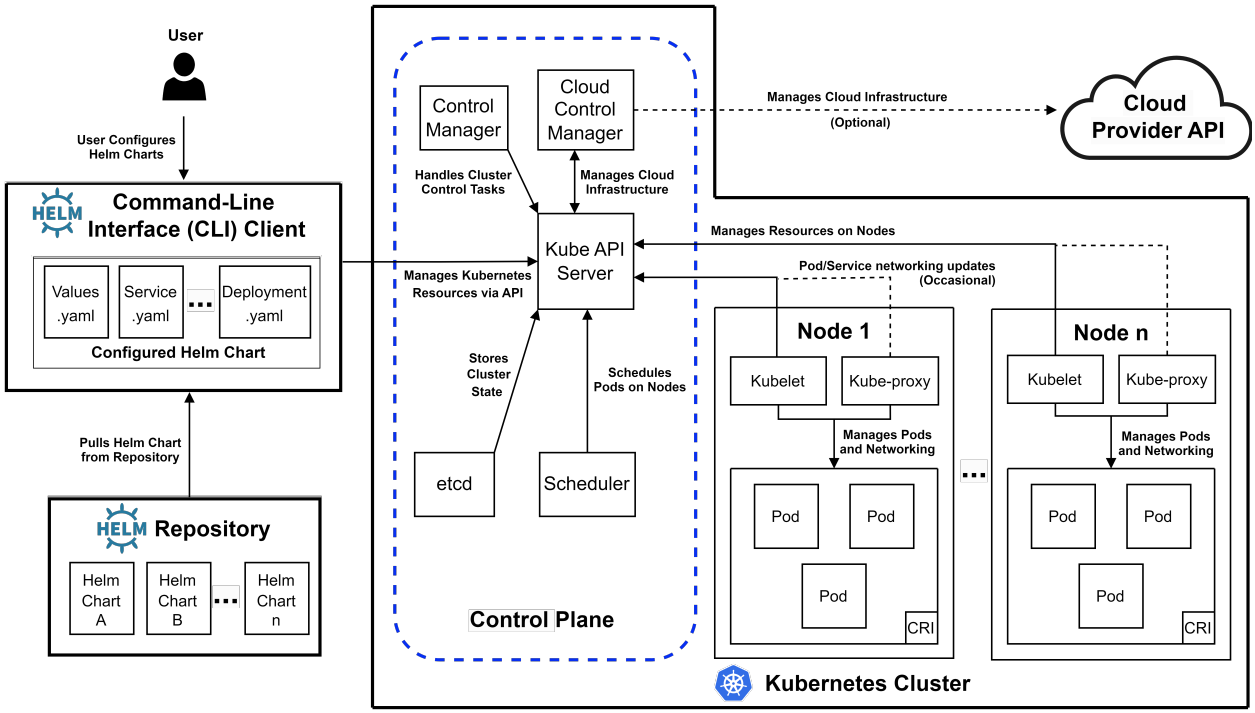


Figure 2.19.: Helm Architecture, own illustration

- **Helm Repository:** Helm Repository is where Helm Charts are stored. The repository can store multiple charts which represents different applications or system components. Helm Charts can be pushed by developers to the repository and being pulled down by users for deploying. Versioning enables to manage easily upgrades and rollbacks of applications on Helm repositories. Figure 2.19 shows some example of charts A, B, and n being stored in a repository as different charts for different Kubernetes configuration or applications.
- **Helm Charts:** Helm Chart is a set of templates and configuration (YAML) for a Kubernetes application. There are multiple files in each chart such as:
  - **Values.yaml:** Default values that can be overridden at deployment time.
  - **Service.yaml:** Define how the application services and interacts on Kubernetes network.
  - **Deployment.yaml:** Contains the definition for application deployment, management, replicas, and update strategies.

  These are modular, so all the components (databases, microservices, monitoring tools) can be managed independently but work together in one Kubernetes cluster.
- **Helm CLI (Command-Line Interface):** Helm CLI is the main tool to interact with helm. From the Helm Repository, users can issue multiple commands to run applications. Some key commands include:
  - **helm install:** Helm Chart installs a Helm Chart onto a Kubernetes cluster.
  - **helm upgrade:** Helm Chart is upgraded with existing changes or configuration changes applied.
  - **helm rollback:** It reverses a deployment to a previous version, in case of a failure or an issue.

  Helm Repository is accessed via CLI, where Helm Chart is being downloaded, and users can make changes to files e.g Values.yaml, Service.yaml, and Deployment.yaml before deploying the chart into Kubernetes cluster. Once configured, users operate the application deployment through the Helm CLI, by interacting with the Kubernetes API.

Helm Workflow and Data Flow

Figure 2.19 shows how Helm and Kubernetes interact, and the following steps outline Helm's operation.

1. **Helm Chart Storage in Repository**: Helm Charts are created by developers, it contains Kubernetes resource definitions (pods, services, deployments, etc.). These charts are stored in the Helm Repository and versioned and shareable. Each Helm Chart independently manages specific parts of the system.

   To extend the above example, Chart A could represent database services, Chart B could have microservices, Chart n – monitoring tools and so on. Each chart can be upgraded or rolled back at a different rate to provides flexibility over with different parts of the system.

2. **Helm CLI Pulling Charts**: The Helm CLI is used by users to pull the chart that's desired from the repository. Following pulling, the parameters of deployment can be customised using different configuration .yaml files.

3. **Deployment via Kubernetes API**: The modified Helm Chart is then sent by Helm CLI to the Kubernetes API Server, which in turn processes this chart and takes care of deploying the defined resources across the cluster. Kubernetes components (etcd, scheduler, controller manager) allocate the required resources.

Helm helps simplify the deployment process, and organize Kubernetes resources into reusable templates, enabling users to manage even large systems with moderate ease. This thesis uses Helm to manage deployment and scaling of multiple OPC UA servers, such that the servers are configured consistently and efficiently allocated to network infrastructure resources.

Kubernetes and Helm in the Context of This Thesis

Whereas Docker has the capability to containerize an application but, for large scale deployments, advanced orchestration is needed. Kubernetes offers three key benefits to achieve auto scaling, load balancing and resource management, which makes it a great choice to deploy multiple OPC UA Servers across distributed environments for this thesis. Helm adds to this with simple ways to configure servers via reusable templates and changing the replicas, resources, and network policies. Kubernetes and Helm together enable consistent, reliable, and scalable control of jumble OPC UA server deployments across the infrastructure.

2.17. Reference Architecture Model Industrie 4.0 (RAMI 4.0)

As illustrated in Figure 2.20, The Reference Architecture Model Industrie 4.0 (RAMI 4.0) is a three axis level model of the structured representation of fundamental elements of an asset. This model allows to decompose the complex interdependencies of an asset, through combining all three dimensions at each stage in an asset's lifecycle in an attempt to incorporate everything relevant [40].

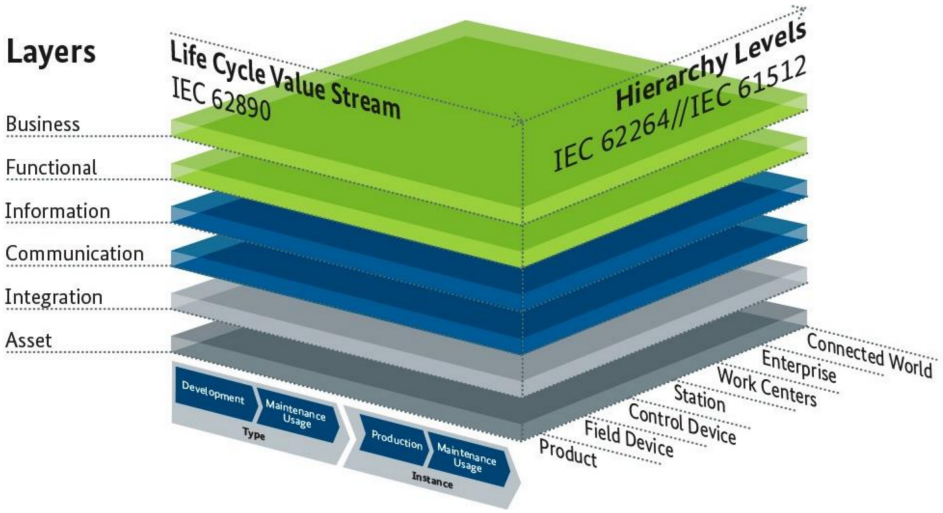


Figure 2.20.: RAMI 4.0 , based on [41]

The three axes comprising RAMI 4.0 are:

- **Architecture axis ("Layers"):** Made up of 6 layers, each displays information necessary for asset role.
- **"Life Cycle & Value Stream" axis:** Describes an asset life cycle and value added process from production, use and disposal.
- **"Hierarchy Levels" axis:** Assigns functional models to certain levels in order to provide a unified representation across industries.

The primary objective of RAMI 4.0 is to describe assets and asset combinations precisely. It is important to note that RAMI 4.0's description is logical, and actual implementations may vary. For instance, it is possible to logically describe MES functionality for a machine by using one level and implementing that functionality using a different level, reflecting the flexibility of the model.

2.17.1. Architecture Axis ("Layers")

Vertical architecture axis consists of six layers, characterizing an asset's characteristics, system structures, functions and function specific data. The structural characteristics of an asset or asset combination are defined in these layers that together contribute to the logical representation of assets and their relationships which are Business, Functional, Information, Communication, Integration, and Asset.

Business Layer

This layer captures the commercial layer of an asset, in terms of conditions of an organization, financial matters, function integrity, business models, legal conditions, etc. It connects different business processes and orchestrates services on the "Functional" layer.

Functional Layer

This layer defines which functions of an asset have to be executed from a logical point of view, how they are executed within the Industrie 4.0 system. This layer deals with digital function descriptions, horizontal integration, runtime environments for services and business processes, and for applications and technical functionality.

Information Layer

This layer is based on the data used, produced or manipulated by an asset's technical functionality. Further, it includes runtime environments for event processing, rule execution, formal model and rule descriptions, data persistence, data integration, acquiring new data, and so on.

Communication Layer

This layer outlines Industrie 4.0-compliant access to asset information and functions by other assets. It defines how data is used, distributed, and accessed within the system.

Integration Layer

This layer bridges the gap between the physical and information worlds. It encompasses the infrastructure for function implementation, technical process control, and event generation.

Asset Layer

This layer represents the physical existence of assets. It is the interface between humans and the information world and the connection of the assets to the "Integration" layer.

2.17.2. Life Cycle & Value Stream Axis

The "Life Cycle & Value Stream" axis describes an asset's representation at different times of its life cycle from production to disposal.

2.17.3. Hierarchy Axis

The "Hierarchy" axis reflects factory reference architecture models (e.g., levels are enterprise, work centers, station) . Various industries' requirements were addressed by integrating Industrie 4.0 specific terms like 'connected world', 'field device', 'product', etc.

RAMI 4.0 in the Context of This Thesis

This thesis uses RAMI 4.0 as a structured way to locate and organize the SDI OPC UA model in the railway system. RAMI 4.0 defines what functionality is connected to which component in a system and maps these to specific layers (e.g., Business, Functional, Communication) and hierarchy levels. This approach also makes complex tasks simpler, guarantees interoperability with EU-Rail and EULYNX standards, and helps explaining it out how each part is associated with the overall functionality for unified and scalable railway network.

3. Project Management

3.1. Requirements

This project's requirements specify the features and functions and the objectives required for an OPC UA based Standard Diagnostics Interface (SDI) to be successfully implemented for the Command and Control Signaling (CCS) system, in the context of the System Pillar Project – Transversal CCS system – Subdomain 2, which defines the diagnostics requirements for railway infrastructure. The main objectives of this project are to achieve the following:

- Support real time diagnostics data collection from field elements of the railway infrastructure.
- Provide the ability to ensure system interoperability with EULYZN compliant infrastructure.
- Keep scalability, flexibility and high availability for future expansion.
- Conform to the OPC UA Information Model for diagnostics.

Functional, non functional, technical and operational requirements are classified. These categories make sure that the project objectives at high level are traceable to specific implementation tasks.

3.1.1. Requirements Structure

This work includes the classification structure for the requirements for this project, in a hierarchical format, namely the general requirements, detailed requirements (functional, non-functional, technical, and operational). These serve as the means of traceability from high level project objectives to specific implementation tasks.

- **General Requirements (GRQ)**: Derived from requirements of the overall project objectives, which in turn reflect user's needs and connects to specific project tasks.
- **Detailed Requirements**: Define in detail how the system will behave and perform.
  - **Functional Requirements (FR)**: Discuss the operational behaviour of the system.
  - **Non-Functional Requirements (NFR)**: Focuses on the system quality attributes and constraints.
  - **Technical Requirements (TR)**: Describe the technical aspects necessary for system implementing.
  - **Operational Requirements (OR)**: Concern on system deployment, monitoring, and maintenance.

3.1.2. General Requirements (GRQ)

The main project objectives induce general requirements which satisfy high level system's goal and these requirements are large in scope and provide guidelines for the detailed specifications.

1. **GRQ-001: Cross-layer collaboration**
   - **Description:** Must allow for cross layer service oriented collaboration at both horizontal (system level) and vertical level.
   - **Priority:** High
2. **GRQ-002: Real-time Performance**
   - **Description:** The system must be able to perform real time performance for critical processes.
   - **Priority:** High

3. **GRQ-003: Compliance with Industry Standards**
    - **Description:** The system must comply to EU-Rail and EULYNX Interface Specification Standard Diagnostic Interface (SDI) to specify that it is standardized and interoperable.
    - **Priority:** High
4. **GRQ-004: Flexibility and Scalability**
    - **Description:** The system must be flexible and scalable such that the system can be expanded by adding more field elements and increased data collection rate.
    - **Priority:** High
5. **GRQ-005: System Monitoring and Maintenance**
    - **Description:** The system must support continuous monitoring, maintenance support and access to data for maintenance, disaster recovery and regular system updates to be operationally stable.
    - **Priority:** Medium

3.1.3. Detailed Requirements (DR)

Functional Requirements (FR)

The functional requirements detail the system's operational behaviours defining what the system is required to do to address user needs and meet general requirements.

1. **FR-001-01: Real-time Data Collection from Field Elements**
    - **Description:** Real time diagnostic data from field elements like Points, Train Detection Systems and Light Signals have to be collected by the system through OPC UA servers.
    - **Priority:** High
2. **FR-001-02: Data Access for Maintenance**
    - **Description:** The data collected has to be available to the Maintenance and Data Management (MDM) systems for further usage (e.g. analysis and predictive maintenance).
    - **Priority:** High
3. **FR-002-01: System Interoperability**
    - **Description:** The system must comply with EULYNX for interoperability between CCS components, for legacy and new systems.
    - **Priority:** High
4. **FR-003-01: Flexible and Scalable System**
    - **Description:** The system must be scalable, and it needs to be able to be expanded to include new field elements and to increase the data collection frequency without degrading system performance.
    - **Priority:** Medium
5. **FR-004-01: User Interface for Data Visualization**
    - **Description:** Relevant stakeholder must be able to view and monitor real time diagnostics data from the collected diagnostics data through a Human Machine Interface (HMI).
    - **Priority:** Medium
6. **FR-005-01: OPC UA Information Model Compliance**
    - **Description:** The implementation of railway field elements must use the OPC UA Information of Model for nodes, attributes and data types and accurately represent railway field elements.
    - **Priority:** High

Non-Functional Requirements (NFR)

Nonfunctional requirements state the quality and performance characteristics of the system by which it must achieve reach the performance, security, and reliability objectives.

1. **NFR-001-01: System Performance**
   - **Description:** The system must process and transmit high volume critical data, in near real time, under conditions where latency is minimized.
   - **Priority:** High
2. **NFR-002-01: Data Security**
   - **Description:** OPC UA secure protocol must be used with the system to protect data with encryption and authentication to protect diagnostics data against unauthorized access.
   - **Priority:** High
3. **NFR-003-01: High Availability and Fault Tolerance**
   - **Description:** The system has to create high availability platforms with redundancy and fail over mechanisms to keep in service even in the event of component failure.
   - **Priority:** High
4. **NFR-004-01: Maintainability**
   - **Description:** The system must be easy to maintain providing updates patches and enable scaling with little downtime.
   - **Priority:** Medium
5. **NFR-005-01: Compliance with Railway Industry Standards**
   - **Description:** The solution must comply with EU-Rail and EULYNX standards of the railway industry because of regulatory issues.
   - **Priority:** High

Technical Requirements (TR)

Technical requirements define what is required for implementation of a project in the form of technical restrictions and technical functionalities.

1. **TR-001-01: Implementation of OPC UA Server for at least One Subsystem**
   - **Description:** The thesis must implement at least one OPC UA server for a subsystem within the EU-Rail and EULYNX interface specification, such as Points, Light Signals, or Train Detection Systems. For testing purpose, that subsystem must support simulation of value changes.
   - **Priority:** High
2. **TR-002-01: Containerized Deployment of OPC UA Servers**
   - **Description:** All OPC UA servers must be containerized to support scalability, efficient resource management, and fault tolerance. The containerized solution should allow for automated orchestration and deployment across the infrastructure.
   - **Priority:** High
3. **TR-003-01: Dynamic Simulation of Field Element Changes**
   - **Description:** The implemetation must be able to dynamically simulate changes to field elements (such as the point machine position) for testing.
   - **Priority:** Medium

4. **TR-004-01: Real-Time Data Synchronization**
    - **Description:** Real time data synchronization of the OPC UA servers to the MDM systems must be ensured.
    - **Priority:** High
5. **TR-005-01: UML Mapping to OPC UA Information Model**
    - **Description:** The UML models of all subsystems must be mapped to the corresponding OPC UA Information Model, and all subsystems must be model base on UML.
    - **Priority:** High
6. **TR-006-01: CI/CD Pipeline Integration**
    - **Description:** Implementing a Continuous Integration Continuous Deployment (CI/CD) pipeline is mandatory to automate building, testing and deployment process for OPC UA servers.
    - **Priority:** Medium

Operational Requirements (OR)

This involve system deployment for monitoring and maintenance to assure efficient working in practice.

1. **OR-001-01: Simulated Deployment Environment**
    - **Description:** Testing of the system requires it to be deployed in a simulated environment that replicates real-world railway conditions.
    - **Priority:** Medium
2. **OR-002-01: Monitoring and Logging System**
    - **Description:** Servers performance and security must be continuously monitored and logged.
    - **Priority:** High
3. **OR-003-01: Backup and Disaster Recovery Plan**
    - **Description:** The system must have a disaster recovery plan having regular backups and must be able to recover in case either fail.
    - **Priority:** Medium
4. **OR-004-01: Regular System Maintenance**
    - **Description:** Schedules for maintenance of software and checks at hardware components must be defined to avoid software and hardware component failure.
    - **Priority:** Medium

3.1.4. Traceability Mapping

Mapping Between Detailed and General Requirements

This table ensures that each general requirement is addressed by specific detailed requirements.

General Requirement	Detailed Requirements
GRQ-001: Cross-layer collaboration	FR-001-01, FR-001-02, FR-002-01, FR-005-01, TR-001-01, TR-005-01, NFR-002-01, NFR-004-01
GRQ-002: Real-time Performance	FR-001-01, FR-004-01, TR-004-01, NFR-001-01, NFR-003-01
GRQ-003: Compliance with Industry Standards	FR-002-01, FR-005-01, TR-002-01, TR-005-01, NFR-005-01
GRQ-004: Flexibility and Scalability	FR-003-01, TR-002-01, TR-003-01, NFR-004-01
GRQ-005: System Monitoring and Maintenance	FR-001-02, OR-001-01, OR-002-01, OR-003-01, OR-004-01, NFR-003-01, TR-006-01

Table 3.1.: Mapping Between Detailed and General Requirements

Mapping Between Detailed Requirements and Use Cases

This table links each detailed requirement to the relevant use cases.

Detailed Requirement	**Use Case**
FR-001-01: Real-time Data Collection from Field Elements	UC-01: Field Element Monitoring
FR-001-02: Data Access for Maintenance	UC-02: Centralized Maintenance Data Access
FR-002-01: System Interoperability	UC-01: Field Element Monitoring
FR-003-01: Flexible and Scalable System	UC-05: System Updates and Scaling
FR-004-01: User Interface for Data Visualization	UC-01: Field Element Monitoring
FR-005-01: OPC UA Information Model Compliance	UC-01: Field Element Monitoring
NFR-001-01: System Performance	UC-01: Field Element Monitoring
NFR-002-01: Data Security	UC-01: Field Element Monitoring
NFR-003-01: High Availability and Fault Tolerance	UC-05: System Updates and Scaling
NFR-004-01: Maintainability	UC-05: System Updates and Scaling
NFR-005-01: Compliance with Industry Standards	UC-01: Field Element Monitoring
TR-001-01: Implementation of OPC UA Server for at least One Subsystem	UC-01: Field Element Monitoring
TR-002-01: Containerized Deployment of OPC UA Servers	UC-03: Deployment Automation
TR-003-01: Dynamic Simulation of Field Element Changes	UC-04: Simulated Testing Environment
TR-004-01: Real-Time Data Synchronization	UC-01: Field Element Monitoring
TR-005-01: UML Mapping to OPC UA Information Model	UC-01: Field Element Monitoring
TR-006-01: CI/CD Pipeline Integration	UC-03: Deployment Automation
OR-001-01: Simulated Deployment Environment	UC-04: Simulated Testing Environment
OR-002-01: Monitoring and Logging System	UC-01: Field Element Monitoring
OR-003-01: Backup and Disaster Recovery Plan	UC-06: Disaster Recovery Planning
OR-004-01: Regular System Maintenance	UC-05: System Updates and Scaling

Table 3.2.: Mapping Between Detailed Requirements and Use Cases

3.2. Specification

This section specifies how the system will satisfy functional, non functional, technical and operational requirements as a bridge between project objectives and technical implementation. It serves as a guide to the detailed design and implementation of the SDI for the CCS system, specifying what has to be conducted.

3.2.1. Functional Specifications (FS)

These functional specifications describe how the system will operate to meet the functional requirements.

1. **FS-01: Real-time Data Collection**
   - **Description:** OPC UA servers will enable the system to continously collect real time diagnostics from railway field elements such as Points, Level Crossing, and Light Signals.
   - **How to Achieve:** To make diagnostic data such as a point positions immediately available to the MDM systems, the system will use the OPC UA data access service to read the diagnostic values and then publish them via the Publish Subscribe mechanism. The architecture reduces latency by optimized network transfer and data throughput.
   - **Related Requirement(s):** GRQ-001, GRQ-002, FR-001-01, FR-001-02
2. **FS-02: Data Accessibility for Maintenance**
   - **Description:** The collected data will be made available to Maintenance and Data Management (MDM) systems for further usage (e.g. analysis and predictive maintenance).
   - **How to Achieve:** MDM systems will subscribe to the server's data points and receive data from OPC UA servers via standard client-server communication. The server will support maintenance teams to access historical data to be able to analyze trends over time.
   - **Related Requirement(s):** GRQ-005, FR-001-02
3. **FS-03: System Interoperability**
   - **Description:** The system will fully comply with EU-Rail and EULYNX Interface Specification SDI to ensure interoperability between new and legacy CCS components.
   - **How to Achieve:** Based on EU-Rail and EULYNX Interface Specification SDI, the OPC UA servers will be developed per each field element and thus it will be possible to provide a unified interface across all field elements.
   - **Related Requirement(s):** GRQ-001, GRQ-003, FR-002-01
4. **FS-04: Scalable Architecture**
   - **Description:** The system will support scalability so new field elements can be added as well as a higher rate of collected data without a loss in performance.
   - **How to Achieve:**
     a) The OPC UA Information Model must cover all field elements from the Standard Diagnostics Interface (SDI) according to EU-Rail and EULYNX Interface Specification as a template.
        – This template will have DataTypes, ReferenceTypes and ObjectTypes related to each trackside asset component (e.g. SDI-Generic, SDI-Point, SDI-IO, and SDI-LC).
        – The model will be able to instantiated by railway manufacturers. This provides adaptability and scalability at the level of OPC UA Information Model to create instances of their specific needs.
     b) Containerize the OPC UA servers using Docker images and use following DevOp tools for scalable deployment:
        – **Kubernetes:** Manage orchestration and scale horizontally as new field elements are added while doing load balancing across multiples OPC UA server instances

  - **Helm:** Configure and deploy Kubernetes resources which makes replication and scaling simple, according to demand, and also simpler management of containerized instances.
- **Related Requirement(s):** GRQ-004, FR-003-01

5. **FS-05: Human-Machine Interface (HMI) for Visualization**
   - **Description:** Through a Human Machine Interface (HMI), the system will enable relevant stakeholder to view and monitor real time diagnostic data, providing them the information to make informed decisions.
   - **How to Achieve:** The real time data collected by OPC UA servers will be visualized with a commercial-off-the-shelf (COTS) HMI for instance UaExpert. The use of this tool will enable relevant stakeholder (e.g. maintenance team) to see what the current status of their field elements, such as points or signals, are and look at historical trends in order to try to find potential problems before they actually impact railway operations.
   - **Related Requirement(s):** GRQ-002, FR-004-01

3.2.2. Non-Functional Specifications (NFS)

These non-functional specifications address the quality attributes the system must achieve.

1. **NFS-01: High Performance and Low Latency**
   - **Description:** The system will transmit critical diagnostics data in near-real-time with low latency.
   - **How to Achieve:** OPC UA optimized binary encoding over TCP/IP for low latency will be used by the system. Message priority handling will ensure that diagnostics data will be transmitted with minimal delay, and network performance will be monitored for bottlenecks.
   - **Related Requirement(s):** GRQ-002, NFR-001-01, NFR-004-01
2. **NFS-02: Secure Data Transmission**
   - **Description:** Data will be protected from the unauthorized access by using secure protocols.
   - **How to Achieve:** OPC UA message encryption and secure channels will be used. Data will be AES-256 encrypted during transmission and authenticated by username and password to ascertain only authorized users can access diagnostic data.
   - **Related Requirement(s):** GRQ-001, NFR-002-01
3. **NFS-03: High Availability**
   - **Description:** The system will be able to work continuously with automatic failover mechanisms.
   - **How to Achieve:** Kubernetes will provide high availability by dealing with pod failures and make sure there is a redundant instance of OPC UA servers. If a server fails, Kubernetes will automatically replace the failed container with a new one and minimal downtime will occur.

     Helm will be also used to configure all those K8s resources related to replica counts, services and health checks in a modular fashion allowing for simpler scaling as well as maintaining availability.
   - **Related Requirement(s):** NFR-003-01, TR-002-01
4. **NFS-04: Maintainability**
   - **Description:** The system will be easily maintainable and updatable with minimal downtime.
   - **How to Achieve:** Using containerized deployment, updates and patches can be applied with minimal downtime by employing Kubernetes' rolling updates feature. This ensures that new versions of the OPC UA servers can be deployed incrementally while keeping the system operational. Helm will be used to manage and deploy these updates seamlessly, allowing for easy configuration and version control during system maintenance.
   - **Related Requirement(s):** NFR-004-01

5. **NFS-05: Standards Compliance**
    - **Description:** The system will comply with railway specification EU-Rail and EULYNX.
    - **How to Achieve:** EU-Rail and EULYNX Interface Specification SDI will apply to all development processes. Manual verification of the system against the standards compliance will be performed to assure the system complies with the essential specifications.
    - **Related Requirement(s):** GRQ-003, FR-005-01, NFR-005-01

3.2.3. Technical Specifications (TS)

These technical specifications outline the specific technologies and methods for implementing the system.

1. **TS-01: UML Mapping to OPC UA Model**
    - **Description:** UML models will be mapped to the OPC UA Information Model for consistency between design and implementation.
    - **How to Achieve:** UML Field elements such as Points, Light Signals and Train Detection Systems will be mapped directly from UML model to the nodes, attributes and references of the OPC UA Information Model. UaModeler will be used to develop the OPC UA Information Model that matches these UML models, and the resulting design and implementation will be fully aligned with the SDI specification.
    - **Related Requirement(s):** GRQ-001, FR-002-01, TR-005-01
2. **TS-02: OPC UA Server Implementation**
    - **Description:** Implement the OPC UA server for at least one trackside subsystem, using the Points system as an example, and using EU-Rail and EULYNX SDI OPC UA Information Model instantiations.
    - **How to Achieve:**
        a) The implementation of EU-Rail and EULYNX SDI OPC UA Information Model will include ObjectTypes, DataTypes, and ReferenceTypes to reflect the structures for all trackside asset, e.g. SDI-Generic, SDI-P, SDI-IO, SDI-LC, and etc. according to their interface specification
        b) The instantiation of the OPC UA Information Model will at least focus on one trackside subsystem, hence:
            – The instance object for the Points system will be created using UaModeler and exported as an XML file, which serves as the primary structure of the OPC UA server.
            – Dynamic values for the Points system, defined in an external Excel file, will be integrated with this model to simulate real-time updates.
        c) The OPC UA servers will be developed using Python with at least following necessary library:
            – **asyncua:** Asynchronous OPC UA server library, supporting nodes, event triggers, and real-time diagnostic event simulation.
            – **pandas:** Simulates real time data updates by loading and processing diagnostic data from Excel files that trigger diagnostic events.
            – **openpyxl:** Reads Excel files for importing configurations so that simulation of different scenarios is possible but is structured.
            – **os and asyncio:** Manages environment variables and asynchronus inputs so that one may set up and generate server and event with ease.

d) These servers will be able to:

- Communicate with Maintenance and Data Management (MDM) systems regarding diagnostics information.
- Help enable real time diagnostics and event triggered maintenance updates for relevant stakeholder (e.g. maintenance team).

- **Related Requirement(s):** GRQ-003, FR-003-01, TR-001-01, NFR-003-01, NFR-004-01, TR-001-01

3. **TS-03: Containerized Deployment with Kubernetes and Helm**
   - **Description:** To achieve scalability, efficient resource management, and fault tolerance all OPC UA servers will be containerized.
   - **How to Achieve:** Docker will be used for containerizing OPC UA servers. Container orchestration will be carried out by Kubernetes, while Helm will be used to configure, manage, and deploy Kubernetes resources, such that deployment is simplifying scaling. Helm will handle updates, scaling and fault tolerance, deploying each server as a Kubernetes pod which will automatically recover from failure.
   - **Related Requirement(s):** GRQ-004, TR-002-01
4. **TS-04: Dynamic Simulation for Testing**
   - **Description:** Simulating dynamic information, reflecting field elements' operation behavior, will be provide in OPC UA Server for testing purposes.
   - **How to Achieve:** Pre configured values in Microsoft Excel will be used to achieve dynamic simulations of field elements. A loop will run continuously updating the values in real time, and these dynamic changes will be reflected in the OPC UA server, so that the diagnostics interface can be tested accurately.
   - **Related Requirement(s):** GRQ-005, FR-001-01, TR-001-01, TR-003-01
5. **TS-05: Real-Time Data Synchronization**
   - **Description:** Real-time synchronization between OPC UA servers and the MDM will be provide.
   - **How to Achieve:** OPC UA's data subscription services will be used to implement a subscription based mechanism. MDM system will subscribe to specific data points (NodeIDs) of OPC UA servers and in real-time, any change in field elements will push to MDM system using OPC UA Publish Subscribe model.
   - **Related Requirement(s):** GRQ-002, FR-001-01, NFR-001-01, TR-004-01
6. **TS-06: CI/CD Pipeline Integration**
   - **Description:** The continuous integration continuous deployment (CI/CD) pipeline will be implemented for the automatic build, test and deploy the processes for OPC UA servers.
   - **How to Achieve:** GitHub Actions will be used to automate such taks as code validation, container build, testing, and deployment into Kubernetes on the CI/CD pipeline. What this automation does is minimize manual intervention and keeps the deployment process consistent.
   - **Related Requirement(s):** GRQ-005, NFR-004-01, TR-006-01

3.2.4. Operational Specifications (OS)

The system's deployment, monitoring, and maintenance is described by these operational specifications.

1. **OS-01: Simulated Deployment Environment**
   - **Description:** Test will be performed in a simulated environment of railway system in order to mimic real world railway conditions.
   - **How to Achieve:**
     a) Dynamic scenarios defined in Microsoft Excel are used to create a simulation environment.
     b) The diagnostic data for the field elements Points subsystem are updated from an Excel file with real time data to simulate real world events.
     c) Diagnostic events will be triggered, system behavior can be tested and data values from the Excel file will be continuously looped and updated in OPC UA server.
     d) The deployment of the OPC UA server will be managed by following DevOps tools:
        – **Docker**: Containerize OPC UA server and its dependency.
        – **K8s**: Orchestrates the deployment, making resource management efficient, and scalable.
        – **Helm**: Set up and deploy the simulation environment under conditions that allow scale and scope of the simulation to be altered.
   - **Related Requirement(s):** OR-001-01
2. **OS-02: Continuous Monitoring and Logging**
   - **Description:** The OPC UA servers will be continuously monitored and logged in order that the performance, security and the maintenance of the servers can be tracked for stable operation.
   - **How to Achieve:**
     a) **UaExpert**: A client tool to monitor OPC UA Server attribute values, for example, diagnostic data, real-time updates and system health metrics. It will be used to support, maintenance logging of maintenance actions and help to identify potential maintenance needs.
     b) **Docker Desktop's built-in Kubernetes management tools**: Used to monitor server performance, system resource usage, downtime, performance bottlenecks. These tools will help maintaine schedules and diagnostics for system.
   - **Related Requirement(s):** GRQ-005, OR-002-01,
3. **OS-03: Backup and Disaster Recovery**
   - **Description:** There will be a disaster recovery plan which takes regular backups to maintain integrity of data and availability of data.
   - **How to Achieve:** Backups and Disaster Recovery will be manual. More robust mechanisms for disaster recovery will be integrated either for future automation or from external solutions.
   - **Related Requirement(s):** OR-003-01
4. **OS-04: Regular System Maintenance**
   - **Description:** Fault tolerance and long term system stability will be maintained via a defined maintenance schedule including routine system checks, updates, and repairs.
   - **How to Achieve:** Rolling updates will be applied in Kubernetes through Helm to avoid downtime and will be monitored periodically for maintenance. Advanced automated tools may be added in future improvements.
   - **Related Requirement(s):** OR-004-01, GRQ-005

3.2.5. Traceability Mapping

Mapping use cases to relevant specifications and specifying a mapping from each requirement to a corresponding specification will be described in this subsection. This approach guarantees that all functional, non-functional, or technical and operational aspects are fully discussed, designed and implemented, and all requirements identified are met.

Mapping Between Specifications and Use Cases

Specification	**Use Case**
FS-01: Real-time Data Collection	UC-01: Field Element Monitoring
FS-02: Data Access for Maintenance	UC-02: Centralized Maintenance Data Access
FS-03: System Interoperability	UC-01: Field Element Monitoring
FS-04: Scalable Architecture	UC-05: System Updates and Scaling
FS-05: HMI for Data Visualization	UC-01: Field Element Monitoring
NFS-01: High Performance	UC-01: Field Element Monitoring
NFS-02: Secure Data Transmission	UC-01: Field Element Monitoring
NFS-03: High Availability	UC-05: System Updates and Scaling
NFS-04: Maintainability	UC-05: System Updates and Scaling
NFS-05: Standards Compliance	UC-01: Field Element Monitoring
TS-01: OPC UA Server Implementation	UC-01: Field Element Monitoring
TS-02: Containerized Deployment	UC-03: Deployment Automation
TS-03: Dynamic Simulation for Testing	UC-04: Simulated Testing Environment
TS-04: UML Mapping to OPC UA Model	UC-01: Field Element Monitoring
TS-05: Real-Time Data Synchronization	UC-01: Field Element Monitoring
TS-06: CI/CD Pipeline Integration	UC-03: Deployment Automation
OS-01: Simulated Deployment	UC-04: Simulated Testing Environment
OS-02: Continuous Monitoring	UC-01: Field Element Monitoring
OS-03: Backup and Disaster Recovery	UC-06: Disaster Recovery Planning
OS-04: Regular System Maintenance	UC-05: System Updates and Scaling

Table 3.3.: Mapping Between Specifications and Use Cases

Mapping Between Requirements and Specifications

Requirement	Specification
GRQ-001: Cross-layer collaboration	FS-01, FS-03, NFS-02, TS-01
GRQ-002: Real-time Performance	FS-01, FS-05, NFS-01, TS-05
GRQ-003: Compliance with Industry Standards	FS-03, NFS-05, TS-02
GRQ-004: Flexibility and Scalability	FS-04, NFR-004-01, TS-03
GRQ-005: System Monitoring and Maintenance	FS-02, TS-04, TS-06, OS-02, OS-04
FR-001-01: Real-time Data Collection	FS-01, NFS-01, TS-04, TS-05
FR-001-02: Data Access for Maintenance	FS-01, FS-02
FR-002-01: System Interoperability	FS-03, TS-01
FR-003-01: Flexible and Scalable System	FS-04, TS-02
FR-004-01: User Interface for Data Visualization	FS-05, NFS-01
FR-005-01: OPC UA Information Model Compliance	NFS-05
NFR-001-01: System Performance	NFS-01, TS-05
NFR-002-01: Data Security	NFS-02
NFR-003-01: High Availability and Fault Tolerance	NFS-03, TS-02
NFR-004-01: Maintainability	NFS-04, TS-02, TS-06
NFR-005-01: Compliance with Industry Standards	NFS-05
TR-001-01: OPC UA Server Implementation	TS-02, TS-04
TR-002-01: Containerized Deployment	TS-03, NFS-03
TR-003-01: Dynamic Simulation of Field Element Changes	TS-04
TR-004-01: Real-Time Data Synchronization	TS-05
TR-005-01: UML Mapping to OPC UA Information Model	TS-01
TR-006-01: CI/CD Pipeline Integration	TS-06
OR-001-01: Simulated Deployment Environment	OS-01
OR-002-01: Monitoring and Logging System	OS-02
OR-003-01: Backup and Disaster Recovery Plan	OS-03
OR-004-01: Regular System Maintenance	OS-04

Table 3.4.: Mapping Between Requirements and Specifications

3.3. Work Plan

The V model is a common systems development life cycle model that is driven by prioritizing testing and verification to assure system quality. This model focuses on performing concurrent testing activities with the development of system in parallel and with testing activities accelerating as the system nears completion. The project's work plan phases are shown in Figure 3.1 showing the schedule, by phase, for the completion of the project. The V model methodology allows the project to adopt a structured development process that enables to meet explicitly requirements and thoroughly evaluate the system. It is carried out within the framework of the user requirements and expectations [42].

The below proposed work plan sequence consists of the following phases, concerning the design and experimental implementation of a Standard Diagnostics Interface for Control-Command and Signalling using an OPC UA server approach based on EULYNX Field Elements.

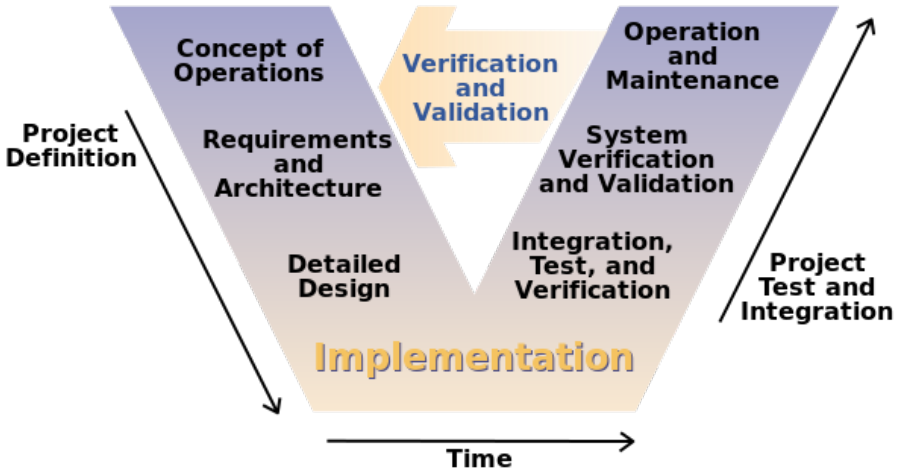


Figure 3.1.: V-model for project management, based on [42]

Phase 1: Requirements Analysis

Activities:

- Evaluate user requirements and project objectives and convert them into functional, non functional, technical and functional requirements.
- Review and perform a thorough study on documents from EU-Rail and EULYNX, in particular on Trackside Asset Specification, and the standard diagnostics interface (SDI) for field elements. They describe how diagnostic interfaces, generic requirements of system, and relevant user information. These components are designed to work together with the system. the detailed diagnostic messages (data point IDs and values) for the field elements (Generic, Points, Light Signals, I/O, Level Crossing and Train Detection Systems) as described in the EU-Rail and EULYNX documentation and shown in UML diagrams.

Expected Outcomes:

- Detailed project documentation that includes functional, non functional, technical and operational requirements and well defined project scope and project objectives.
- Understand diagnostic interface and data point definition details for Points, Light Signals, and Train Detection Systems, with UML diagrams.
- A traceability matrix connecting requirements to future design and development actions.

Phase 2: Detailed Design

Activities:

- Based on the requirements defined in Phase 1 refine the system architecture so that all of the elements are completely specified for design and implementation.
- Analyze the UML provided by EU-Rail and EULYNX, plan and map the OPC UA Information Model for the translation of all trackside field elements (Points, Light Signals, Train Detection Systems) into OPC UA nodes, attributes and references.
- Containerize the deployment with Docker, orchestrate with Kubernetes and helm for config deployment.
- Defining out CI/CD pipeline with GitHub Actions for the processes of building, testing and deploying the Opc Ua servers in a dockerized environment.
- Design critical and non critical failure scenarios of Points Subsystem. Real time diagnostics and event driven data handling will be shown to expose critical failures and non critical failures. For real time simulation, these scenarios will be executed in Excel and will be integrated into the Python code.

Expected Outcomes:

- A detailed system design document for the OPC UA Information Model and mapping thereof from UML diagrams for Points, Light Signals and Train Detection Systems.
- Architecture based on detailed of the containerized architecture of the OPC UA servers on the basis of Docker, Kubernetes, and Helm to achieve the scalability, fault tolerance and easy management.
- This GitHub Actions CI/CD pipeline plan automates the development, tests, and deployment processes to integrate new feature at run at continuous integration.
- Real time failure simulations are defined with critical and non critical scenarios.

Phase 3: Implementation

Activities:

- Use UaModeler to develop the OPC UA Information Model based on UML diagrams provided by EU-Rail and EULYNX for trackside (field) element, such as Points, Light Signals, Train Detection Systems, and etc.
- focusing on creating instances for our Points system and exporting the model as an XML file to use in the OPC UA server implementation.
- Implement the OPC UA server using Python with asyncua, pandas, openpyxl, os and asynncio libraries. Simulate real time diagnostic data over the instance XML model for the Points system.
- Failure scenarios will be developed and integrated and then dynamically simulated using Excel as input, and integrated into the Python server code to originate real time updates and events.
- Containerize the OPC UA server with Docker to make it ready for scalable deployment.
- Using Kubernetes and Helm to deploy and orchestrate the containerised OPC UA server in a configured and controlled way.
- Develop and test OPC UA server locally with Kubernetes using Docker Desktop.

Expected Outcomes:

- A fully implemented OPC UA server for Points with real time data handling and diagnostics.
- Pre-defined of both critical and non critical failure scenarios with correponding diagnostic events.
- An OPC UA server in a Docker container, which ready to be deployed.
- Files configuring the Kubernetes and Helm for orchestrating and managing deployment.
- Validated and tested deployment environment using Docker Desktop for Kubernetes.

Phase 4: Integration, Test, and Verification

Activities:

- Real time diagnostics and monitoring is integrated into the OPC UA server with the Maintenance and Data Management (MDM) systems.
- Test the dynamic data updates and trigger diagnostic events to the OPC UA server as well as validate the instance. This includes the critical and non critical failure scenarios from Phase 3.
- Execute integration tests to make sure everything works all together well inside a containerized computer, verifying the total system performance.
- When deployed in Kubernetes verify the OPC UA server operates with required level of scalability and tolerance to faults
- Verify automatic OPC UA server deployment in the Kubernetes out of the CI/CD pipeline.

Expected Outcomes:

- OPC UA server successfully integrated with MDM systems and supporting real time diagnostics.
- The Points system OPC UA server functioning with dynamic data updates and the event triggering based on simulated scenarios were verified.
- Results from load tests proving system performance and scalability requirements.
- The CI/CD pipeline is verified and automated the deployment process in kubernetes environment.

Phase 5: System Verification and Validation

Activities:

- Compliance with EU-Rail and EULYNX Interface Specification in regards to Standard Diagnostics Interface (SDI) and OPC UA Information Model.
- Ensure that the implementation of the overall system architecture meets requirements with respect to the project's functional, nonfunctional, technical, and operational requirements.
- Execute end to end testing of the OPC UA server system integrated to the Maintenance and Data Management (MDM) systems to test flow of data correctly from field elements (Points) to MDM for real time monitoring.
- Review the all failure scenarios (critical and non) and make sure the system behaves as expected and write the log into the MDM systems.
- Formal system verification documentation detailing the test results, compliance checks and validation outcome shall be prepared.

Expected Outcomes:

- Compliance with EU-Rail and EULYNX interface specifications SDI verified.
- Complete verification that the stated system architecture and implementation meet all of the listed requirements (functional, non functionality, technical, operational).
- End to end integration testing between OPC UA servers and the MDM systems were successful.
- All failure scenarios have documented test results to show that the system behaves as expected.

Phase 6: Deployment and Maintenance

Activities:

- Fully implemented OPC UA server system shall deployed into the simulated production environment where all components is configured correctly and operational.
- Use UaExpert as a monitoring tool to see or check system behavior while being deployed and operated. This will include checking real time data updates, system diagnostics and event triggering.
- Make the system ready for scalability by the potential of expanding into more field elements such as Light Signals and Train Detection System in the future.
- Set up basic manual monitoring and logging process so that critical system logs are logged and can be referenced for debugging purposes or were being have logged for the purposes of audit.
- Automate future updates and deployment with the CI/CD pipeline as deploying, as well as testing of all new features or fixes should be handled using Github Actions.
- Make a disaster recovery plan count on manual backup and recovery procedure to revenge to the state that can revivify in contingency of breakdown.
- Make rolling updates in Kubernetes with Helm to apply system update without downtime, so that the system availability and stability still exist in the time of implementation.

Expected Outcomes:

- Deployment of OPC UA server with real-time diagnostics successfully done in a simulated environment.
- UaExpert provides real time monitoring of system behavior with logs for debugging or auditing.
- Scalable system architecture which capable of being extended for further field elements as needed.
- Basic monitoring and logging were established and the system itself is ready to be used.
- Automated functional CI/CD pipeline to feature updates and deployments.
- Manual backup and recovery strategies in the context of disaster recovery plan.
- Successfully applied rolling updates with continuous availability of the system.

4. Detailed Design

The detailed design of the OPC UA based diagnostics system for railway control-command and signaling (CCS) systems is presented in this chapter. The system architecture is refined, the OPC UA Information Model, Scenario Simulation and Failure Handling, OPC UA server, deployment strategies using Docker, Kubernetes, Helm, and the CI/CD pipeline are also realized to be designed. The emphasis is on developing approaches to scalability, interoperability, and real time diagnostics of the trackside field elements.

4.1. System Architecture

For monitoring and diagnostic in a railway CCS system, the system architecture is containerized using Docker, orchestrated by Kubernetes, with Helm managing configuration. The architecture takes a scalable, resilient approach that allow simple to manage and supports continuous collection, processing and data analysis.

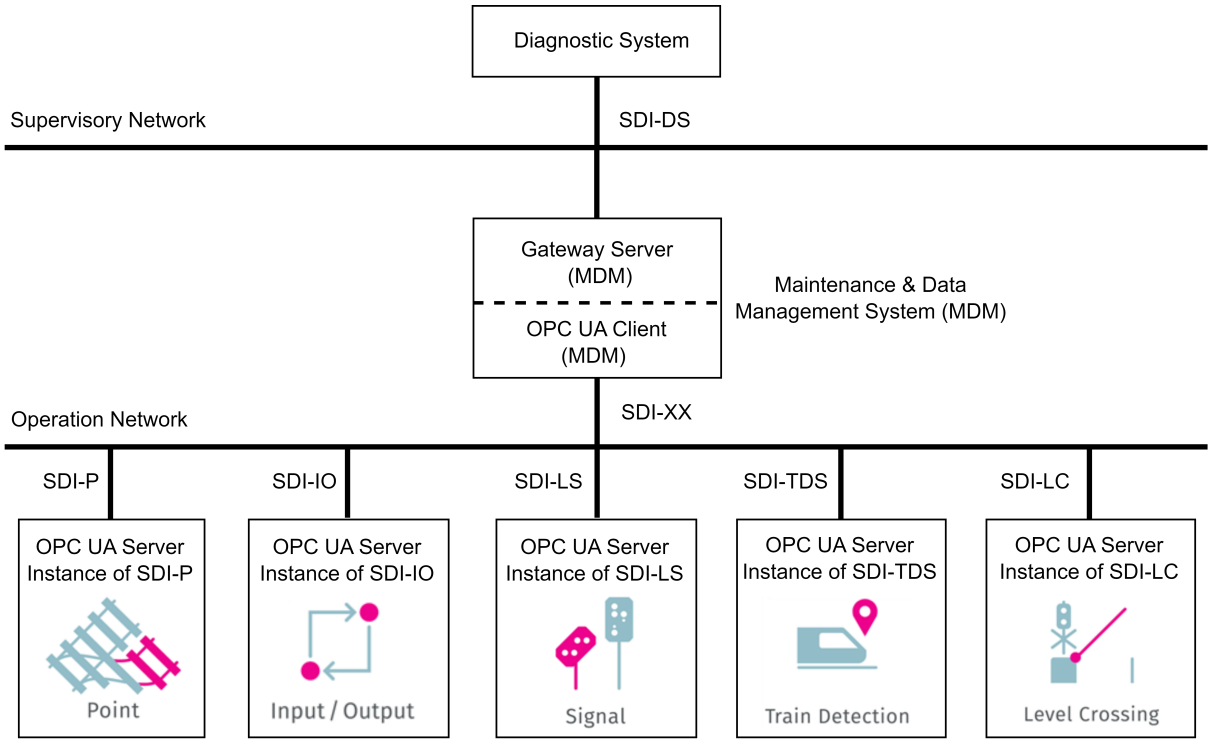


Figure 4.1.: System Architecture Overview, own illustration

Figure 4.1 shows two main networks; the Operational and Supervisory Networks. In the Operational Network, OPC UA server instances collect real time data from CCS subsystems and communicate with the Maintenance and Data Management (MDM) system that processes and sends necessary data to the Diagnostic System in the Supervisory Network.

This structure allows for scalability, fault tolerance, and it keeps monitoring continuous. The following will discuss the system's main functionalities and main components.

4.1.1. Concept Overview

The architecture supports the following key functionalities:

- **Monitoring and Diagnostics:** CCS operations, performance and health are continuously monitored by the system that detects and reports CCS failures to support maintenance and fault detection.
- **Data Collection:** Dianogsitc Data is collected from CCS track side field elements (e.g., points, signals, train detection) such as counter, raw data, etc. for use in analysing further.
- **Data Provisioning:** Algorithms are used for detection of anomalies, diagnostics of the faults and prediction of possible upcoming issues, based on collected data, that will be stored and analyzed.
- **Predictive Maintenance:** Faults are identified and predictive maintenance recommendations are made using analysis tools so that repairs and replacements may be carried out before performance is affected.

4.1.2. Major Components

There are three main components of the architecture: OPC UA Server Instances, the Maintenance and Data Management System (MDM), and the Diagnostic System. They function on two networks: Operational Network and Supervisory Network.

- **OPC UA Server Instances:** For the railway network, each OPC UA server instance is responsible for managing diagnostic and operational data for associated signaling subsystems, e.g. Points, Light Signals, and Train Detection Systems. These servers:
  - Collect continuously real time field data (status, performance, health metrics).
  - Deliver data to the MDM system through standard diagnostic interfaces (SDI-XX).
  - Containerized by Docker, use K8s for orchestration, and Helm for deployment consistency.
- **Maintenance and Data Management System (MDM):** The MDM system is used to collect data of OPC UA servers in the Operational Network. Key components are:
  - **OPC UA Client:** Collect diagnostic and operational data from OPC UA servers.
  - **Gateway Server:** Shares processed data across the Supervisory Network by linking the MDM system to the Diagnostic System via SDI-DS.
- **Diagnostic System (SDI-DS):** The Diagnostic System works at the Supervisory Network level thus the centralized monitoring and diagnostic services. It gathers high level data from MDM system via Gateway server, and provides forecast for Predictive Maintenance, fault detection and other system health analysis on the entire CCS (Control Command Signaling) system.

4.1.3. Network Overview

The system is organized in two separate networks: the Operational Network and the Supervisory Network, having each its own purpose in the system's functioning.

- **Operational Network:** The OPC UA servers which manage the different signaling subsystems locate in the Operational Network. Real time data from trackside field elements (Points, Signals, TDS) is collected by each server and transmitted to the MDM system via SDI-XX using OPC UA for the communication protocol.
  - For scalability, high availability and fault tolerance the OPC UA servers are deployed as Docker containers and managed by Kubernetes and Helm.
  - It is designed for future expansion, e.g. interlocking systems, moving block, ATO-TS (Autofunction Train Operation – Trackside).

- **Supervisory Network:** Offers a centralized facility through the Diagnostic System via Gateway Server, which receives processed data from the MDM.
  - The communication protocol between Gateway Server and Diagnostic System remains undetermined, for its future requirement and integration consideration.
  - The Diagnostic System provides real-time diagnostic insights on specific trackside asset groups or networks, as well as performance data, supporting maintenance and system upgrade decisions.

4.1.4. Data Flow and Processing

The OPC UA servers collect data from elements at the trackside and transmit that data to the MDM system via standardized SDI interfaces (SDI-XX). This data is organized in a structured manner by the MDM system and transmit to the Diagnostic System via SDI-DS for maintenance and operational insight analysis.ts.

- **Data Collection:** Real time data from a Points, Signal, or Train Detection system is gathered by OPC UA servers and sent to the MDM system.
- **Data Packaging:** The data collected is arranged in a standard format with the MDM system, ready for further processing.
- **Data Transmission:** Diagnostic System receives the packaged data from the MDM system so that this data can be centrally analyzed to support maintenance and decision making.

4.2. OPC UA Information Model Design

The representation of diagnostic data across field elements within the Control Command Signaling (CCS) system is standardized using the OPC UA Information Model design. System components and their relationships are encapsulated by ObjectTypes, VariableTypes, DataTypes and EventTypes, respectively. This model brings consistency and interoperability between subsystems (such as Points, Light Signals, Train Detection,I/O Controllers, Level Crossings) according to EU-Rail and EULYNX Interface Specification SDI.

In this design, ObjectType is used to define a trackside field element, which has Variables, DataTypes and References for capturing diagnostic information. This structure allows for predictive maintenance and fault detection with a standardized, interoperable diagnostic data for suppliers and manufacturers.

4.2.1. Namespace Structure and Object Instantiation

First, an OPC UA Information Model is created in the generic namespace with templates for subsystems, i.e. Points, Train Detection and Light Signals, based on EU-Rail and EULYNX Interface Specification SDI. The diagnostic data and the relationships between them for each subsystem are presented using this generic model.

Each subsystem class can be instantiated in the manufacturer namespace as a specific real world equipment once the generic templates have been defined. For instance, PointType template in the generic namespace is customized for manufacturer before instantiation in the manufacturer namespace. This separation between the generic and manufacturer namespaces provides modularity, consistency and scalability because manufacturers can define their own product's diagnostic information model.

4.2.2. Key Elements of the OPC UA Information Model

The primary components of the OPC UA Information Model include:

- **ObjectTypes:** Represent different subsystems (e.g., SDI-P, SDI-TDS) and their diagnostic properties.
- **VariableTypes:** Store the state of subsystems (e.g., Position, Aspect, Fault Status).
- **DataTypes:** Specify the data format for variables (e.g., Boolean, Enumeration, DateTime).
- **EventTypes:** Enable the tracking of specific events (e.g., diagnostic alarms or system health monitoring).

4.2.3. Subsystem Representation in OPC UA

UML class diagrams of each subsystem in the CCS are mapped to the corresponding OPC UA ObjectTypes in the sense that the diagnostic data complies with the SDI interface specifications (EU-Rail, EULYNX Interface Specification SDI). Guaranteed interoperability across systems and suppliers is provided by this mapping.

- **Point System (SDI-P):** Mapped to PointType, this includes variables such as:
  - **Position:** Provides the current position of the point machine.
  - **End Position Reached:** Confirms whether the point has reached its final position.
- **Light Signals (SDI-LS):** Mapped to LightSignalType, include variables such as:
  - **Luminosity Current:** Reports the current luminosity level of the light signal.
  - **Logical Light Point Status:** Provides the operational status of individual light points (e.g., LEDs).
- **Train Detection System (SDI-TDS):** Mapped to TrainDetectionSystemType, with variables such as:
  - **Occupation Status:** Indicates if the track section is occupied by a train.
  - **Passing Status:** Tracks whether a train has passed a certain point on the track.
- **I/O Controllers (SDI-IO):** Mapped to IOControllerType, handles input/output field elements, such as:
  - **Logical Input Value:** Represents the current value of logical input channels.
  - **Flashing Period Configuration:** Defines the flashing configuration for specific outputs.
- **Level Crossing (SDI-LC):** Mapped to LevelCrossingType, includes variables such as:
  - **Barrier Status:** Provides the status of the level crossing barrier.
  - **Warning Lamp Status:** Reflects the operational state of the warning lamps at the crossing.

Each subsystem is characterized by ObjectType, which defines a common diagnostics interface for implementations. The instantiation of these subsystems in the manufacturer namespace uses templates from the generic namespace in order to achieve consistency and modularity of the system.

4.2.4. Mapping UML Relationships to OPC UA Information Models

Based on the EU-Rail and EULYNX Interface Specification SDI, the UML class diagrams for the CCS subsystems are translated into OPC UA ObjectTypes including their relationships, ensuring consistency of standardized diagnostic data. Various types of UML association symbols are employed to define relationships between class objects, which are mapped to OPC UA with equivalent constructs. This mapping include:

- **Generalization in UML:** Mapped to the HasSubtype reference in OPC UA. Inheritance here means that subtype node inherits attributes and references from its parent type.
- **Associations in UML:** Shows relationships between two classes, and those relationships can be mapped to one of the many OPC UA reference types, such as HasComponent or GeneratesEvent. It can also be represented by ObjectPlaceholder or Object depending on the nature of the association.
- **Composition in UML:** A strong ownership relationship whereby one class owns another. This is modeled in OPC UA using the HasComponent reference. It can also be presented as ObjectPlaceholder or Object in accordance with the nature of the composition.
- **Aggregation in UML:** A whole part relationship in which one class consists of one or more parts without strictly owning them. This is modeled in OPC UA using the HasComponent reference, Both ObjectPlaceholder or Object can be used used if appropriate for the nature of the aggregation.

For detailed examples of these mappings, refer to the diagrams in the Appendix A.1.1 - UML to OPC UA Mapping Diagrams. These diagrams illustrate how the UML relationships (e.g., Generalization, Association) for subsystems like Points, Light Signals, Train Detection Systems, I/O Controllers, and Level Crossings are translated into OPC UA ObjectTypes, Variables, and References.

4.2.5. Mapping SDI Attribute Types to OPC UA Node Types

Fundamental nature of the data and its purpose as part of the system determines the correlation between attribute types outlined in the EU-Rail and EULYNX Interface Specification SDI and OPC UA node type. Below is the breakdown of why each attribute type is associated with a specific OPC UA node type:

1. **Raw Data:** This is uninterpreted data measured. Continuous values, or discrete states, can be included. Typically these types are represented as BaseDataVariableType or BaseAnalogType nodes because these types give the ability to represent scalar values and continuous analog signals.
   - **BaseDataVariableType:** Continuous raw data which can be represented by numeric values.
   - **BaseAnalogType:** Designed specifically for analog data that has other properties such as engineering units and range.
2. **Diagnosis:** Provide indications on the status or condition of a system and most often are expressed using discrete values, such as enumeration or boolean. The following TypeDefinitions are used to represent these attributes in OPC UA:
   - **BaseDataVariableType:** Represent discrete values as it is suitable for diagnosis attributes.
   - **MultiStateDiscreteType:** A specialized subtype of BaseDataVariableType for attributes with a finite set of discrete states, and thus well suited for representing diagnostic statuses.
3. **Counter:** Contains diagnostics or occurrences of specific data measurements or events. It is used to track operations or cycles events. In OPC UA, a counter applies the BaseDataVariableType and is good for the display of diagnostic data increments over time.
4. **Configuration:** Defined specifically by the manufacturer or operator, not from measured data. In OPC UA, the OPC UA attributes are mapped to PropertyType nodes. PropertyType nodes were designed to describe properties or features of objects in the OPC UA address space and certainly serve well to represent configuration data.

A comparison table mapping attribute types from Interface Specification SDI to OPC UA node types is given below:

SDI Attribute Types	OPC UA Type Definition	Description
Raw Data	BaseDataVariableType/ BaseAnalogType	Represents direct measurements from field elements, such as voltage or temperature, that can be continuous or discrete. (e.g., Voltage or Humidity)
Diagnosis	BaseDataVariableType/ MultiStateDiscreteType	Indicates system status through discrete values like enumeration or boolean states, suitable for fault detection and monitoring. (e.g., Enumeration or Boolean).
Counter	BaseDataVariableType	Tracks count of specific events or operations, such as the number of cycles or activations. (e.g., Turn Counter)
Configuration	PropertyType	Defines system settings or characteristics, usually set by operators or manufacturers. (e.g., Manufacturer or Operator settings)

Table 4.1.: Mapping of SDI Attribute Types to OPC UA Node Types

4.2.6. Mapping SDI Data Types to OPC UA Data Types

The Interface Specification SDI outlines generic data types in UML diagrams to model diagnostic data flow. These types are mapped to OPC UA built-in data types for interoperability:

SDI Data Types	OPC UA built-in data types	Description
Long	UInt64 (UA:9)	Mapped to UInt64 (UA:9), accommodating a vast integer range from 0 to 18,446,744,073,709,551,615, sufficient for large-scale numerical requirements.
Real	Float (UA:10)	Mapped to Float (UA:10), representing a single-precision (32-bit) floating-point number, utilized for precision-dependent measurements.
Integer	Int16 (UA:4) / UInt16 (UA:5) / Int32 (UA:6) / UInt64 (UA:9)	Integer is mapped by four types: Int32 for medium ranges, UInt64 for large ranges, UInt16 for smaller ranges (e.g. voltage and power), and Int16 for small values (e.g. index numbers).

Table 4.2.: Mapping of SDI Generic Data Types to OPC UA Types

4.2.7. DataTypes and VariableTypes

To ensure consistency in the format of data exchanged, the OPC UA Information Model utilizes predefined DataTypes and VariableTypes.

- **DataTypes:** Standard data formats such as Boolean, String, DateTime, NodeId, and enumerations for specific subsystem states are included. For instance, Table 4.3 lists the values of the possible states of point movement failure defined by the PointTurnFailureStatus enumeration.
- **VariableTypes:**Used to capture variables related to motor operations (e.g. current speed and torque) for a specific use of diagnostics, which is MotorTurnDataType.

EnumString & SymbolicName	Value	Description
None	0	No point movement failure
Timeout	1	Point movement failed due to timeout
UnsuccessfulStartOfMovement	2	The start of point movement was unsuccessful
AllPmStoppedButCommandedPositionNotReached	3	All point machines stopped, but commanded position not reached
NoDrivePower	4	No power available to drive the point machines
Other	5	Other miscellaneous reasons for point movement failure

Table 4.3.: Enumeration: PointTurnFailureStatus

Table 4.3 provides an example of custom Enumeration DataType of PointTurnFailureStatus that gives standardized feedback on point machine failure statuses in the OPC UA Information Model. In this example, the model achieves interoperability across systems with its conformance to the UML Class Diagram presented in the EU-Rail and EULYNX Interface Specification SDI.

4.2.8. AccessLevel and Historizing Settings

To ensure consistent data access as well as utilization of data, each and every variable of the OPC UA server must be set to its type regarding Access Level and Historizing properties. The following settings are applied:

- **Raw Data and Counter Variables:**
  - **Historizing:** False
  - **AccessLevel:** 1 (CurrentRead)

  Only current value read operations are available for these variables, to guarantee that OPC UA clients can only obtain the latest static data and do not get access to historical values.
- **Configuration and Diagnosis Variables:**
  - **Historizing:** True
  - **AccessLevel:** 5 (CurrentRead/HistoryRead)

  These variables allow both read operations of current value and historical data which allows for complete data access for monitoring and diagnostics.

These AccessLevel and Historizing settings are implemented uniformly to provide consistency for the data access across the entire OPC UA Information Model.

4.2.9. Event Types

Logging and operational purposes are represented via hierarchy of event types within the OPC UA Information Model. The OPC Foundation provides the default event types at the foundation, EU-Rail and EULYNX extend the default event types specific for their diagnostic interfaces. Figure 4.2 shows the overall event hierarchy.

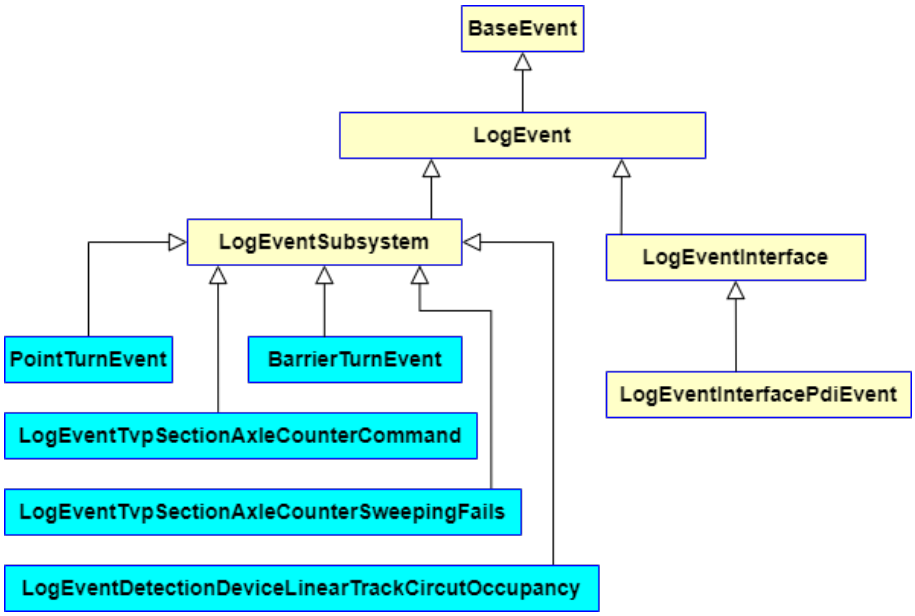


Figure 4.2.: Event Hierarchy in EU-Rail and EULYNX Interface specification SDI Generic, own illustration

- **BaseEventType:** Default event type from OPC UA.
- **LogEventType:** Introduced by EU-Rail and EULYNX, with two key subcategories:
  - **LogEventInterfaceType:** For events related to system interfaces.
  - **LogEventSubsystemType:** For events specific to subsystem activities.

Detailed descriptions of these event types and their specific use cases are provided in the sections below.

Base Event Types

BaseEventType: OPC Foundation provides the foundational event type. This includes the basic structure needed for event handling in the OPC UA model, for example attributes of type EventId, EventType, LocalTime etc. All other event types inherit from the BaseEventType, which is also abstract.

Log Event Types

LogEventType: EU-Rail and EULYNX introduce LogEventType in order to extend the BaseEventType with the specific EventType, consisting of two main categories with events related to interfaces and subsystems. It inherits BaseEventType, and adds MessageId, which is the unique logged event identifier.

The LogEventType is divided into two key subcategories:

- **LogEventInterfaceType:** This type is related to system interface. Following Sub-event type included:
  - **LogEventInterfacePdiEventType:** This event type is created to log activities of the application level in the Safe Communication Protocol (SCP) and it contains specific errors and event notifications that relate to the PdiError and PdiEventNotification.
- **LogEventSubsystemType:** Events of this type are specific to subsystems of the diagnostic model. It is a base for various event types specific to subsystems. Some examples of these are:
  - **PointTurnEventType:** Events related to movement of points within the system are captured by PointTurnEventType, and attributes such as CommandedPosition, FailureReason, IsEndPositionReached are logged.
  - **BarrierTurnEventType:** Events related to barrier movements, including failure reasons and movement statuses are logged.
  - **LogEventTvpSectionAxleCounterCommandType:** Logs commands received for axle counters in a specific train section.
  - **LogEventTvpSectionAxleCounterSweepingFailsType:** Events related to failures when sweeping axle counters are captured
  - **LogEventDetectionDeviceLinearTrackCircuitOccupancyType:** Logs occupancy detection events for linear track circuits.

These custom event types, defined by EU-Rail and EULYNX, ensure that critical system events are logged consistently, supporting both interface-related and subsystem-related diagnostics.

4.2.10. DefaultInstanceBrowseName

The OPC UA server must declare all terminal object types derived from UML Generalization relationships with a DefaultInstanceBrowseName variable. It is used to provide a standard naming convention for each instance of the address space in the OPC UA server, and the variable DefaultInstanceBrowseName is configured with the following characteristics:

- **Node Class:** Variable of PropertyType.
- **Data Type:** QualifiedName, holding a name within a namespace.
- **Access Level:** 5 (CurrentRead | HistoryRead), supporting real-time and historical data reading.
- **Historizing:** True, maintaining a record of value changes over time.

This ensures consistent identifiability of every instance across the OPC UA server providing reliable data exchange and management.

Hence, diagnostic data for the field elements are standardized and structured by mapping the UML class diagrams to OPC UA Information Models in order to enable a seamless interoperability between different manufacturers and systems. Based on this model, the diagnostic data is uniformly accessible and usable for maintenance, fault detection, and system optimization.

4.2.11. Exporting the OPC UA Model as XML

Two XML files are exported: One is from the generic namespace, containing examples of subsystem templates (e.g., Points, Train Detection, Light Signals), and the other from the manufacturer namespace with the Point subsystem specific instantiated. The manufacturer namespace depends on the generic templates so both of the files are required. The Python OPC UA server imports these XML files to load full model for monitoring, diagnostics, and scenario simulation.

4.3. Scenario Simulation and Failure Handling Design

In this section, the design of failure and operation scenarios that the OPC UA server will simulate to show its real time diagnostics and failure detection ability for real time CCS systems is presented. These scenarios simulate the normal and failure conditions, and give structured framework to test the server's diagnostic response as per these conditions.

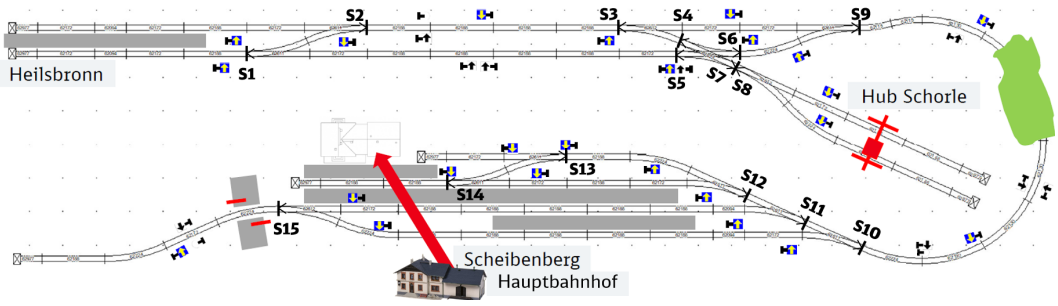


Figure 4.3.: Test Track Layout: Scheibenberg 2.0, own illustration

The Scheibenberg 2.0 test track layout, a model used at DB InfraGo for testing purposes is shown in Figure 4.3. It is a scaled down version of a European Train Control System (ETCS) Level 2 installation. The track has 15 point switches controlled by object controllers that manage their movement and position and handle groups of five point switches.

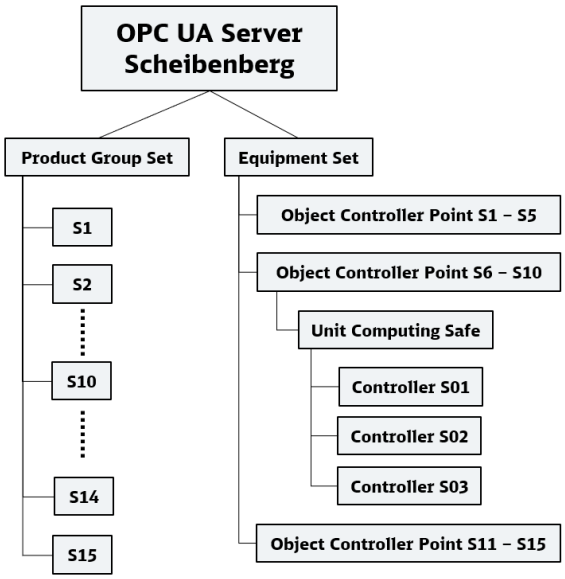


Figure 4.4.: Conceptual representation of OPC UA Server Scheibenberg, own illustration

Figure 4.4 conceptually illustrates the OPC UA server's organization and the equipment it manages, serving as the basis for the failure scenarios. The focus is on object controllers and their assigned point switches.

As a result, the scenarios described in this section relate to failures of Point Switch S10 and the object controller responsible for Point Switches S6 to S10. This layout enables the simulation of realistic railway conditions and the validation of the failure detection and diagnostic capabilities of the system.

4.3.1. Motivation for Scenario Selection

Inspired by a real world railway journey concept, the selected failure scenarios are based. For Scenario 1, the hypothetical train leaves the station of Heilsbronn, crosses switches S2, S3 and S9 and fails at switch S10. Once this critical failure is resloved the train goes on moving through Scheibenberg Hauptbahnhof, past S15 and to the end of the line.

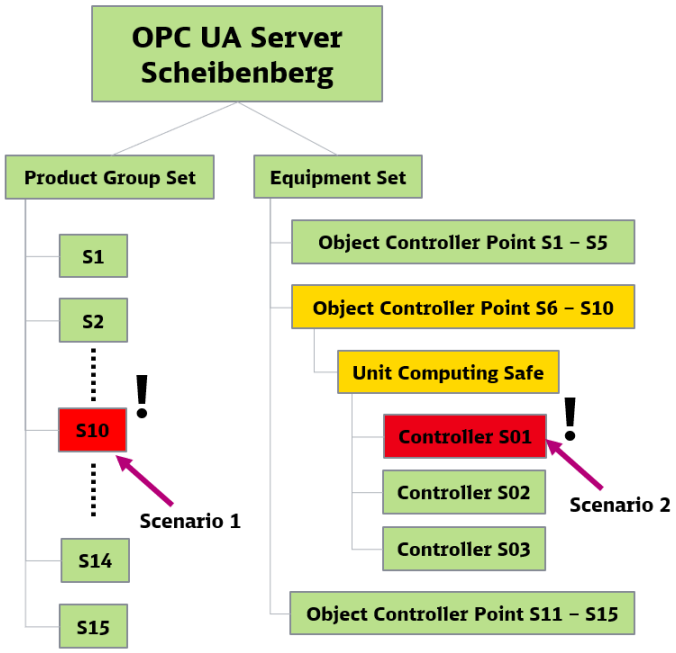


Figure 4.5.: Failure scenarios in OPC UA Server Scheibenberg, own illustration, own illustration

As shown in Figure 4.5, the two failure scenarios are described as follows:

- **Scenario 1:** A critical failure at Point S10 needs immediate attention; Operation will be disrupted until it can be resolved.
- **Scenario 2:** A non critical failure in Controller S01 of the Unit Computing Safe of Opject Contoller Point S6 - S10. This failure does not stop the operation, but it is logged, and tracked for later resolution.

This thesis does not simulate moving train but the journey is used as a framework to show the importance of the failures. The goal is to show how these events are captured, logged, and delivered to the OPC UA server Standard Diagnostic Interface (SDI) for consumption by clients such as UaExpert. These simulations verify the server's capability to handle critical and non critical failures in real time and confirm that the SDI can be used to support operational reliability and safety in railway systems.

4.3.2. Diagnostic Approach for Simulating Failures

For point switches (S1–S15), the OPC UA server simulates both real world failures and normal operations using predefined diagnostic data created in Excel. An advantage of this approach is that it enables the real time testing of the server's ability to detect, log and respond to both critical and non critical problems. The server updates node values and triggers diagnostic events as each scenario progresses, and clients such as UaExpert can monitor these. If no scenario is active the server returns to a 'default' state where any continuous performance monitoring can continue.

4.3.3. Scenario 1: Critical Failure – Point Switch Failure

This is a critical failure when Point Switch S10 does not reach its end position. Failures of point switches are essential for directing trains and can cause serious disruption to operations. The point switch failure is detected by the system, repair is initiated, and the status of the point switch is monitored during the repair.

Scenario Breakdown

- **Initial State**: Point S10 starts in the "Left" position, fully operational and able to move. The system is ready to execute commands.
- **Failure Detected**: The movement of Point S10 is commanded to move, but the end position of movement fails to reach there. If the point remains in an unintended position the system detects this and logs the failure. When this happens the alert is critical and immediate maintenance is needed.
- **Repair Initiated**: Once the system detects the issue, it alerts the maintenance team and the repair process starts. At this time, the point switch remains stuck and the system logs its status. Based on this, the repair is in progress and consequently the diagnostic variables are updated to reflect that.
- **Repair Completed**: When the repair is done the system checks that Point S10 is operational at full functionality. The system is able to confirm that it is able to move again.
- **Movement Verification**: The point is repeatedly commanded to move left and right, while monitoring movement to confirm the problem has been resolved.

The following Table 4.4 summarizes the major variables which update at different step in the scenario.

Variable	Initial	Failure	Repair Initiated	Repair Completed	Final Status
AggregateAbleToMoveStatus	Able	Not Able	Not Able	Able	Able
LastCommandedPosition	Left	Left	Left	Left	Left
PointAbleToMoveStatus	Able	Not Able	Not Able	Able	Able
Position	Left	Unintended Position	Unintended Position	Left	Left
MovementStatus	Not Moving	Not Moving	Not Moving	Moving To Left	Moving To Left
StatusTechnical	OK	Failure Critical	Failure Critical	OK	OK
IsEndpositionReached	True	False	False	True	True

Table 4.4.: Scenario 1: Critical Failure – Point S10 Fails to Reach End Position

The changing state of Point S10 during failure, repair, and recovery is captured in the key variables, as seen by Table 4.4. These variables are monitored by the system continually, in real time, to detect problems, log failures, and to support maintenance efforts.

Conclusion of Scenario 1

This scenario the OPC UA server is able to manage critical failures by detect, repair, and verify restored successfully. The system tracks progress during the repair process and monitors on a continuous basis to identify problems. The point switch is also verified to work correctly after repairs to prevent future disruption.

4.3.4. Scenario 2: Non-Critical Failure – Object Controller Failure

This is a non critical failure on the Unit Computing Safe, which controls Point Switch S6 to S10. While this failure does not shut operations down immediately, it must be addressed, or it will continue to degrade. Diagnostic variables update throughout the progression of failure through sub-equipment issues.

Scenario Breakdown

- **Cooling Fan Failure**: First, the system detects that a cooling fan has failed. CoolingFanStatus variables are updated with the issue.
- **Temperature Increase**: The temperature rises due to the relative cooling fan failure. This is logged by the system using temperatureStatus, which now indicates an abnormally high temperature.
- **RAM Degradation**: When overheating strikes, it begins to degrade the RAM. Warnings are raised and the variable ramHealthStatus is updated.
- **CPU Degradation**: As it heats up though, the system also begins to degrade CPU performance. This is reflected in the variable CpuHealthStatus and maintenance warnings are raised.
- **RAM Failure and Controller Fallback**: The RAM eventually fails and the controller enters fallback mode. The non critical failure RamHealthStatus and StatusTechnical are updated.
- **Controller Failure (Critical Failure)**: The failure is transitioning from non-critical to critical, as the controller is failed. It needs to be replaced immediately.
- **Unit Computing Safe (Non Critical Failure)**: In StatusTechnical, the Unit Computing Safe itself first is warned and then turns to non critical failure and, finally, degraded state of the whole system. Full functionality must be restored through replacement.
- **Component Replacement and Restart**: The new SerialNumber and SoftwareRevision are logged as the faulty components are replaced, and the system is restarted to insure the problems have been fixed.

The following summarizes the key variables in this scenario, shown in the following Table 4.5:

Variable	Cooling Fan Fail	Temp Too High	RAM Degraded	CPU Health Degraded	RAM Failure	Controller Failure
CoolingFanStatus	Failure	Failure	Failure	Failure	Failure	Failure
TemperatureStatus	Normal	Too High	Too High	Too High	Too High	Too High
RamHealthStatus	Normal	Normal	Degraded	Degraded	Degraded	Failure
CpuHealthStatus	Normal	Normal	Normal	Degraded	Degraded	Failure
StatusTechnical	OK	OK	Warning	Warning	FailureNonCritical	FailureCritical
OperationStatus	In Operation	In Operation	In Operation	In Fallback	In Fallback	In Fallback

Table 4.5.: Scenario 2: Non-Critical Failure within Unit Computing Safe

Conclusion of Scenario 2

This scenario shows how the system can deal with non–critical failures by logging problems, sending off warnings, and logging component changes. In fallback mode the system remains operational and will continue to function while the failures are resolved.

4.3.5. Integration of Scenario Data into OPC UA Server

For both Scenario 1 and Scenario 2, the diagnostic data will be organized in an Excel file listing all variable states. The OPC UA Server uses this file to launch and operate each scenario, simulating failures, logging events of interest, and showing recovery actions.

Server's performance will be validated using UaExpert, a Commercial Off-The-Shelf (COTS) client, to ensure that the expected diagnostic information is displayed and the server can handles each failure scenario.

4.4. OPC UA Server Implementation Design

Real time monitoring and diagnostics of railway field elements can be handled by the OPC UA server. The CCS systems and supervisory networks use it to monitor continuously all trackside subsystems: Points, Light Signals, and Train Detection Systems using it as an interface. The design of the OPC UA Server and its main functions, such as real time data access, event generation and scenario simulation, are described in this section.

4.4.1. Design Principles

To operate efficiently, be scalable, and be compatible with the wider CCS system, the design of the OPC UA server follows below design concept:

- **Modularity:** Subsystems and field elements are represented as distinct nodes on the server with data such as position and fault status.
- **Instance Representation:** Actual equipment such as Points and Train Detection Systems are represented as templates and manufacturers create real world OPC UA Server instances from these templates.
- **Event-Driven Architecture:** Events are triggered by operational changes, like reaching an end position or subsystem failures, aiding real-time diagnostics.
- **Scenario Simulation:** Operational changes (eg. reaching an end position or subsystems failures) trigger events that support real time diagnostics.
- **Asynchronous Operation:** The server manages multiple tasks and connections with the help of asynchronous programming, to support real time communication.

4.4.2. Core Libraries and Tools

Key Python libraries have been included in the design of the OPC UA server for efficient, reliable and structured integration for real time diagnostics and event handling:

- **`asyncua`:** Based on its asynchronous capabilities, asyncua is selected as the OPC UA server foundation, supplying node management, event trigger, and real–time diagnostic event simulation. This library is needed to ensure interactive server-client interactions in the server design.
- **`pandas`:** Real time data updates are simulated directed from Excel files and are stored and processed in the form of diagnostic data using pandas. In the design, this approach triggers diagnostic events in order to perform dynamic scenario testing.
- **`openpyxl`:** Reading Excel files that contain configuration data information for supporting structured scenario simulations is enabled by openpyxl. With this design, the server can address many different scenarios using predefined settings which are imported.
- **`os` and `asyncio`:** The server design is designed to use the os and asyncio libraries to handle environment variables and asynchronous I/O in order to setup flexible servers and generate events.

4.4.3. Server Initialization, Instance Creation, and Real-Time Operation

The OPC UA server design is targeted to instance creation from OPC UA Information Model and real-time data updates and event handling for simulation and monitoring of railway field elements. Two XML files are required to set up the server: one defines the manufacturer namespace with Point subsystem instantiated objects and the other the generic namespace with the subsystem templates. These two XML files are essential, because the namespace of the manufacturer XML depends on templates in the generic namespace.

Server Initialization

The initialization of the server for the asynchronous, real time operations is done with asyncua library. The server's name, port and endpoint are all configured by dynamic environment variables for flexibility of deployment. The server follows the structure defined in Section 4.2 - OPC UA Information Model to be consistent with all of its subsystems.

- **XML Import:** Initialization consist in importing two XML files. The first one, with the generic namespace, defines templates for the subsystems: Points, Train Detection Systems and the Light Signals. The first comes from the user (or client) namespace, and offers instantiated objects exclusively for the Point subsystem. The structural basis for each subsystem is established by this dual XML import.

Node Data Loading

To load the node details (e.g. its properties and data types) from external Excel files, pandas has been taken into account for the design. This allows each field element, including Points and Train Detection systems to be accurately represented in the server's address space. The XML based model has its complement the node data that helps to further customize as well as to perform the scenario simulations.

Subsystem Instantiation

And after XML is imported, the server instantiates real world objects based on an OPC UA Information Model. For instance, a specific point machine is instantiated as objects (S1 - S15), enabling real time monitoring and diagnostics. This instantiation design allows clients to have concrete, application specific data to operate on.

Scenario Simulation

The server reads operational condition values from predefined Excel files to enable scenario based testing. The scenarios are real world events (eg, equipment failure) so the system design supports extensive testing and validating of diagnostic capability.

- **Predefined Scenarios**: Excel files containing the scenarios are loaded and executed in sequence, updating the environment by updating the nodes and triggering the events. This approach allows observation of system responses to a variety of conditions (e.g. subsystem failures).

Real-Time Data Updates

This server was designed to offer real-time, continuous updates of the data for the Point instances. This setup reads the values from predefined data in an Excel file and the values then appear immediately in the OPC UA server, so clients such as UaExpert will see the updated values at the same time. This feedback mechanism operates in realtime, enabling clients to observe the state of each object instance as the state changes in a synchronized way.

Event Handling

The design of the server is event driven where it can trigger an event and broadcast to any number of clients when there is an operational change or failure, i.e. when a point machine does not reach a commanded position. This event handling capability allows clients to obtain immediate feedback and supports real time diagnostics.

4.4.4. Security and Access Control

Some restrictions of client access to information are implemented on the server design in order to constrain how information can be viewed or modified.

- **Node Access Levels:** For each node, specific access levels are defined, whether or not a node can be read or modified. For example, there are certain read only nodes and some are read/write nodes.
- **Historizing:** Certain nodes allow historical data tracking to be enabled so that clients can retrieve past values for analysis. This enables long term trend recognition and repeated fault diagnosis.

In summary, the OPC UA server is designed as a real-time interface for the monitoring and diagnosis of railway field elements. It supports testing, validation and monitoring of the server by instantiating subsystem templates from the OPC UA Information Model and scenario based simulations. This is built to be modular, event driven and asynchronous to handle complex real time operations and secure client communication.

4.5. Deployment Architecture

A deployment architecture is designed to allow OPC UA servers managing signaling field elements to be highly available, scale, and automate. Docker is used to containerize and Kubernetes orchestrates it, Helm manages the configuration and deployment. The building, testing and deployment of OPC UA server instances is automated using a CI/CD pipeline.

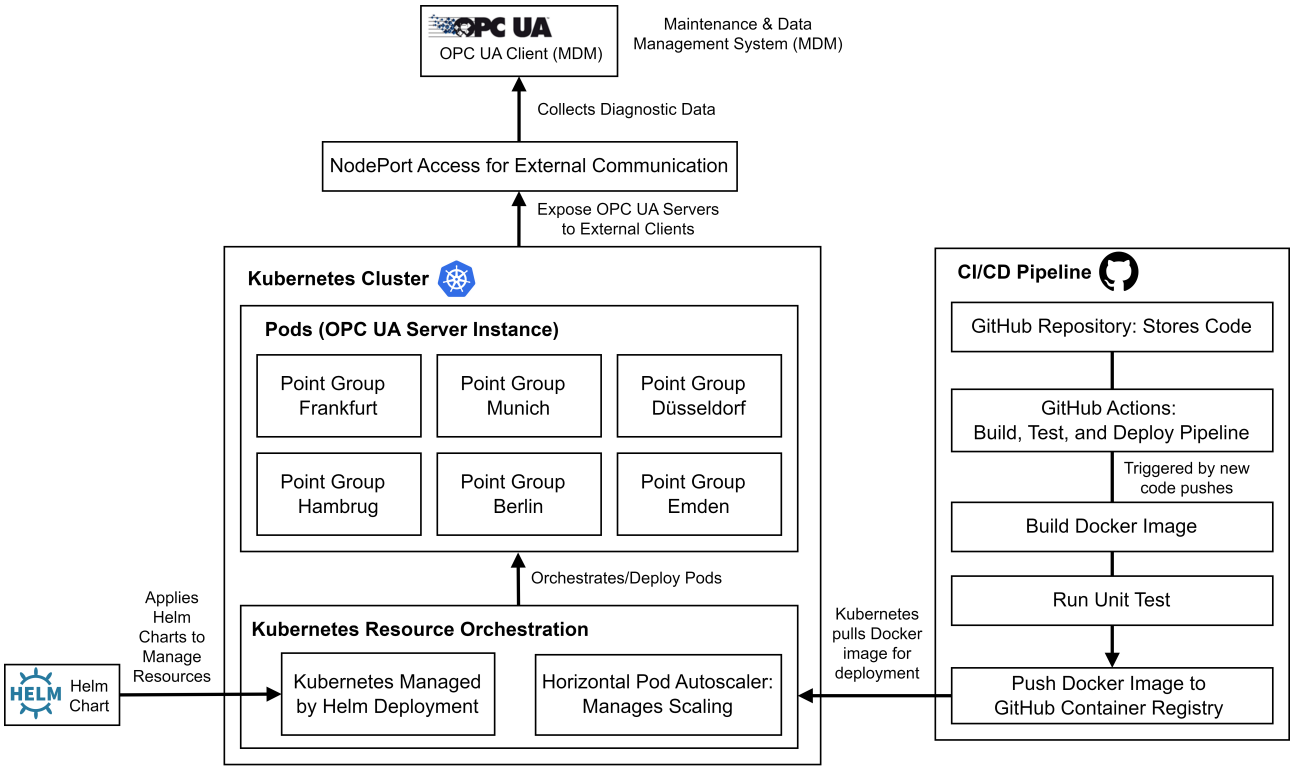


Figure 4.6.: Deployment Architecture with Kubernetes, Helm, and CI/CD Pipeline, own illustration

The deployment architecture, as shown in Figure Figure 4.6 comprises of multiple key components, as Docker containers, Kubernetes clusters, Helm for configuration management and the CI/CD pipeline running on top of GitHub Actions.

4.5.1. Containerization with Dockerfile

For this design, Docker is used to containerize the OPC UA server instances. Encapsulating the server and its dependencies into a lightweight, portable image guarantees that the deployment of the server to any environment will be consistent. In order to make sure the deployments are consistent in different environments, the Dockerfile has to include the following configurations:

Base Image and Setup

The OPC UA server Docker image is designed to based on the official python:3.12-slim image. This is chosen as a base image due to its very minimal footprint, with full Python runtime, with no unnecessary components. This makes this image lightweight and the deployment to multiple environments is more efficient with small containers. The "WORKDIR /app" command is used to change the working directory inside the container. It means that application files and dependencies will be in a structurely position.

System Dependencies Installation

It is a design with system-level dependencies that support server functionality and library compatibility. Key dependencies include:

- **`gcc`**: Compiles Python packages with C extensions.
- **`libffi-dev`**: Supports system level interactions as well as cryptographic operations.
- **`libssl-dev`**: Provides a secure communication over SSL/TLS transport protocol.
- **`make`**: To build and compile Python libraries.

There is a retry mechanism in a Dockerfile for the "apt-get" command from network interruptions while installing the OS. The build process also removes unnecessary cache files to further reduce the image size and further improve deployment efficiency.

Application and Python Dependencies

With "COPY" command, application files are copied into the container. This guarantees that all of the needed files are on hand for the server to function. By using "pip install –no-cache-dir -r requirements.txt", it ensures that all the libraires specified in requirements.txt are installed with no saving unnecessary package caches.

Exposing the OPC UA Server Port

The "EXPOSE 4840" opens the necessary port (4840) to be used by external communication. This is the standard port to OP UA server-client communication and external system connect this port to server.

Launching the Server

On container start up, the command "CMD ["python", "server.py"]" runs the OPC UA server. With this, the server is up and running as soon as the container is deployed, streaming real time diagnostic data from field elements (e.g. Point Subsystem).

4.5.2. Version Control and Dependency Management

An additional file named ".gitignore" is designed to be included in the system which efficiently manages version control. This file prevents certain files and directories to be version tracked and will thus not be stored in repository unless they are required. The key configurations in the ".gitignore" to excludes several categories of files are as followed:

- **IDE Config Files**: Workspace settings such as ".vscode/", .idea/, etc are ignored.
- **System Files**: Excluded are ".DS_Store" (macOS) and "Thumbs.db" (Windows) OS files.
- **Python Bytecode**: Files created by compiling Python, such as "pycache/" and ".pyc" are not tracked.
- **Build Artifacts**: To prevent clutter, all directories starting with "build/", "dist/", or ".egg" are excluded.
- **Virtual Env.**: Any project specific environment directories e.g. '.env', '.venv/' are exempted.

This design organizes the repository so that it contains only necessary source code and configurations.

4.5.3. Dependency Management with requirements file

A "requirements.txt" file has been designed to manage dependencies by listing required Python libraries used by the OPC UA server in order to work properly. This configuration enables the consistent installation of all packages needed through core server functionality as well as interfacing with outside services. Specific design requirements are deployed with the key dependencies in the "requirements.txt" file:

- **`asyncua 1.1.0`**: Supports the OPC UA protocol.
- **`pandas 2.2.2`**: Used for handling and manipulation of data.
- **`openpyxl 3.1.5`**: Excel based simulations are supported.

For specific design objectives, these dependencies have been chosen for real time data management, to secure communications, to ensure that the server functioned properly and securely in a dynamic environment.

4.5.4. Docker Registry Access

In this design, to add security to the Docker registry, there is a "access_registry_docker.txt" file which can help for managing the access. It contains the minimum amount of information needed for authenticated interactions with the GitHub Container Registry and facilitates consistent and secure deployment of the OPC UA server across environment. Key design elements include:

- **Authentication Credentials**: The Docker registry username and Personal Access Token (PAT) are stored securely and accessed from GitHub Secrets for usage in CI/CD workflows.
- **Automated Registry Access**: Automatic, secure login is achieved on the Docker registry in the CI/CD pipeline and controlled interactions with the container registry are enabled.
- **Standardized Container Deployment**: Deployment parameters are defined in the design to guarantee the consistent execution of the OPC UA server's container on different environments.

With these setup, there is guaranteed secure access and secure deployment of the OPC UA server into different environments.

4.5.5. Kubernetes Orchestration

K8s is used to orchestrate the deployment of OPC UA server containers in the design, in order to provide scaling, high availability, and automatic recovery from failures. An OPC UA server instance is encapsulated in a pod and kubernetes manages the pods dynamically according to the system requirements configured.

Pod Management and Replica Control

Core configurations for managing OPC UA server pods are defined inside a "deployment.yaml" file and leveraged for the design as follows:

- **Replica Count**: "values.yaml" defines the number of pod instances through the "replicaCount" parameter to let Kubernetes scale the deployment on the basis of load.
- **Rolling Update Strategy**: "updateStrategy" supports rolling updates by incrementally updating pods and thereby ensuring minimal downtime. "maxUnavailable" and "maxSurge" control how many pods are affected by updates at any one time.
- **Container Settings**: These are defined for each OPC UA server instance with a name, port, and endpoint that are set via environment variables ("OPCUA_SERVER_NAME" and OPCUA_SERVER_PORT) to allow easy customization.
- **Labeling and Selection**: Labels, e.g. "app: opcua-servers", are meant to gather and distinguish the associated set of pods so that K8s stays in control of the right group of pods under the deployment.

These setup's approach of deployment is scalable, and allows for flexibility to define customized server configurations, and facilitates reliable updates.sure service continuity.

Horizontal Pod Autoscaling (HPA)

Horizontal Pod Autoscaling (HPA) is designed to automatically adjust the number of running pods, according to a resource utilization metric such as CPU and memory usage. In the "hpa.yaml" file the HPA settings are configured include:

- **Target Utilization**: The usage is limited by "targetCPUUtilizationPercentage" and "targetMemoryUtilizationPercentage" to lay threshold for services CPU and memory usage.
- **Minimum and Maximum Replicas**: To limit the number of replicas this design uses the "minReplicas" and "maxReplicas" parameters. These parameters determine a range along which fluctuating loads can be handled affordably with high availability and resource efficiency.

Kubernetes is therefore scaled to manage system loads, where the application pods are scaled in relation to traffic or resource demands using this HPA configuration.

Networking and Service Exposure

For the purpose of exposing the OPC UA servers to external systems, service resources defined in "service.yaml" are used. With the "NodePort" access type, Kubernetes routes internal ports to ports outside of the Kubernetes cluster, so OPC UA clients can connect from outside the Kubernetes cluster. The unique port assigned to each server instance, specified in "values.yaml" makes sure to route to correct pod.

Overall, the Kubernetes configuration is targeted to handle the OPC UA server instance in a more efficient way including automatic scaling, resilient networking and high availability for efficient real-time diagnostics within a distributed system environment.

4.5.6. Helm for Configuration Management

Helm was used in the design to allow for simplified and templated deployment and management of multiple OPC UA server instances. In Helm, a centralized "values.yaml" file controls dynamic configuration change such as replica counts, ports or resource allocations on all server instance. This design helps in consistent configuration management and allows for easy update via rolling update strategy of Kubernetes.

Templated Deployment and Services

Helm templates are used to configure deployment and services of OPC UA servers, in "deployment.yaml" and "service.yaml". These templates call parameters from "values.yaml" to make deployments flexible and scalable. Key design parameters include:

- **Replica Count**: This setting sets the number of pod instances created for each OPC UA server controlled by "replicaCount" parameter in "values.yaml".
- **Rolling Update Strategy**: This rolling updates with "maxUnavailable" & "maxSurge" to maintain at least one pod during an updates to keep the downtime to a minimum.
- **Image Settings**: The "image" section contains repository, pull policy and tag information to specify which image version will be deployed.
- **Service Ports**: Each server instance has it's own unique port for outside communication, defined in "service.yaml" and configurable per server in "values.yaml".
- **Environment Variables**: "deployment.yaml" has custom environment variables templated for server name, port and endpoint to allow for per server instance tailor made configurations.

Centralized Configuration with values.yaml

The centralized configuration provided by the 'values.yaml' file can be configured without touching individual templates and can be deployed to the entire deployment. Key configurable parameters include:

- **Replica Count**: Adaptable to adjust resource allocation amount of replica pods with the "replicaCount".
- **Rolling Update Parameters**: Update strategy configurations under "updateStrategy" specify "maxUnavailable" and "maxSurge" for high availability during updates.
- **Service Ports and Endpoints**: Configures service ports and OPC UA server endpoints per instance allowing effective routing and communication with external clients.
- **Resource Limits**: Limits CPU and memory resources for pod and requests resources to optimize pod performance and to make sure that the pod has sufficient resource to work.

The design is set up with Helm so that it is easy to update and rollback with "values.yaml". K8s use rolling update strategy to apply those changes in real time with the minimum service disruption. This templated, centralized approach, supports efficient deployment and consistent configurations for all server instances.

4.5.7. CI/CD Pipeline Integration

A CI/CD pipeline is also included in the design, on the right side of Figure 4.6, to automate integration, testing, and deployment of the OPC UA server images. This pipeline is powered by GitHub Actions and triggers with any changes pushed to the GitHub repository to guarantee straight and reliable deployments. The CI/CD pipeline is structured with the following stages:

- **Build**: Each time the pipeline runs, it checks out the latest code, builds the Docker image for the OPC UA server and provides a consistent image.
- **Push to Registry**: The container is tagged and pushed securely to the GitHub Container Registry (GHCR) which makes it available for Kubernetes to pull during deployment.
- **Deploy and Test**: At this stage, a K8s cluster is created in GitHub Actions using "kind" (Kubernetes in Docker). Then the OPC UA server is deployed through Helm so that the pipeline can test that the Helm chart, Kubernetes configurations, and Docker image all function properly in an isolated environment

This CI/CD pipeline design creates a secure sandbox environment to test the end to end deployment workflow safely reducing the risk of getting into trouble with a live environment. The pipeline provides automated stages of building, pushing and testing thereby guaranteeing a reliable deployment process with as little manual intervention as possible.

4.5.8. Scaling and High Availability

Kubernetes is used to achieve scalability and high availability of OPC UA server instances (as shown in Figure 4.6). Horizontal Pod Autoscaler (HPA) measures resource metrics, cpu and memory usage, and adjusts active pod count on demand in the system. This automated scaling mechanism supports the system to adapt to increased workloads during peak load periods while conserving resources during low periods thereby optimizing both performance and resource efficiency.

4.6. Detailed Design Summary

In this chapter, the detailed design of the OPC UA based diagnostics system for railway control–command and signaling (CCS) systems has been presented. The OPC UA Information Model, Scenario Simulation and Failure Handling, deployment strategy was designed including the use of Docker for containerization, Kubernetes for orchestration, Helm for configuration management and a CI/CD pipeline for automated deployment. As depicted in Figure 4.6, this architecture results in high availability, scalability and minimal downtime with a modular and fault resilient solution to real time diagnostics and fault handling on the track side field elements. The components are designed to increase system resilience, interoperability and continuous monitoring capabilities to provide a solid basis for railway diagnostics and predictive maintenance.

5. Implementation

This chapter explains the implementation of the OPC UA-based diagnostics system for railway control-command and signaling (CCS) systems. The implementation closely follows the design outlined in Chapter 4 - Detailed Design, covering the modeling of the OPC UA Information Model using UaModeler, OPC UA server implementation, the project's repository structure and setup configuration, containerization, deployment using Kubernetes and Helm, continuous integration and deployment (CI/CD), Each section describes how the design decisions were translated into actual implementation steps.

5.1. System Prerequisites

To make sure for a consistent and functioning deployment of the OPC UA based diagnostics system there are many tools and components that need to be preinstalled and configured correctly. An overview of the software, versions and the purpose of each component used with this Implementation is provided in Table 5.1.

Component	**Version**	**Purpose and Notes**
Helm	3.15.4	Manages Kubernetes configuration and deployments, and allows templated and version controlled setup for safe updates.
Kubernetes (kubectl)	1.30.2	Kubernetes cluster node management, service updates and monitoring based command line client.
Kustomize	5.0.4	Overlays configuration without touching the base files and customizes Kubernetes resources. For resource configuration.
Visual Studio Code	1.94.2	Code and configuration files recommended IDE. Integrated extensions make it easier to read and edit.
Docker	27.1.1	OPC UA server containerized. Enables wsl 2 integration and Resource Saver to work efficiently.
Ubuntu (WSL 2)	22.04	A Docker distribution for WSL 2 on Windows. Works on local systems, ensure compatibility with containerized environments.
Python	3.12	Server script for OPC UA execution compatible with async libraries and data handling packages.
UaModeler	1.6.4 459	An Information Model design tool for CCS system elements that structures diagnostic data.
GitHub Container Registry	Required	Place to store Docker images that allow version control and ease access to Kubernetes deployments.

Table 5.1.: System Prerequisites for OPC UA-based Diagnostics Deployment

Table 5.1 lists the tools and configurations providing the basis setup necessary to deploy the OPC UA based diagnostics system, guaranteeing compatibility, performance and smooth integration of the components.

5.2. Modeling the OPC UA Information Model using UaModeler

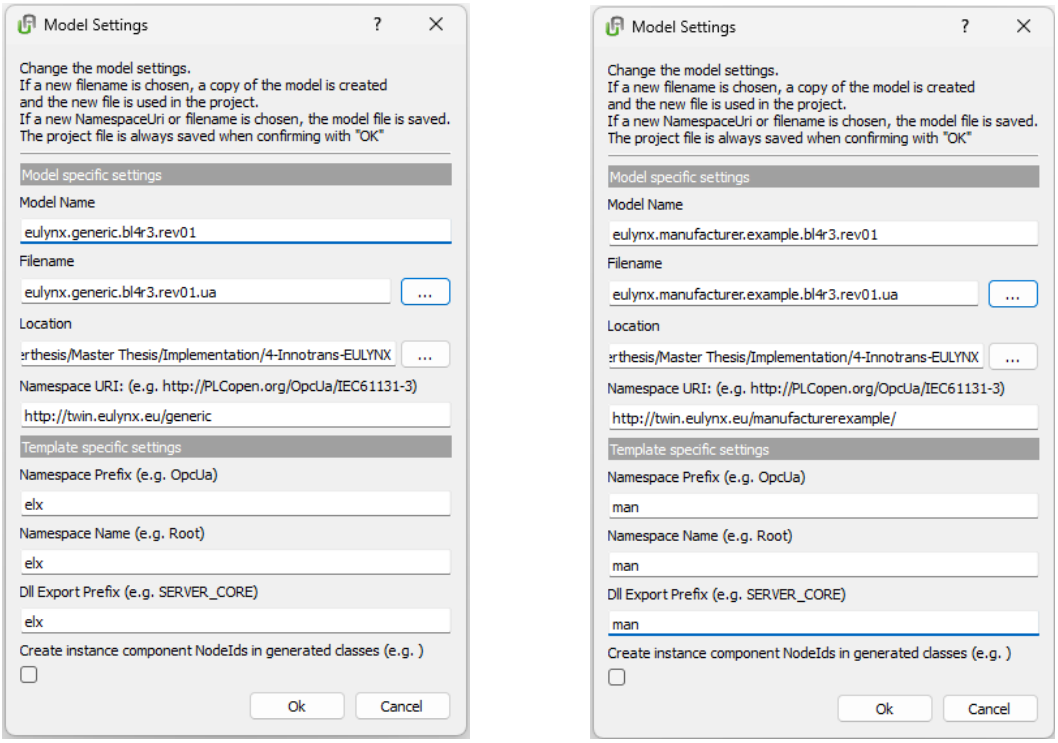
The OPC UA Information Model, based on EU-Rail and EULYNX Interface Specification SDI, provides a standardized structure for railway diagnostics. Built on OPC UA core spec (Version 1.04.6), the model includes a custom SDI interface template and a manufacturer-specific model to simulate real-world diagnostics, optimizing structure and reducing redundancy, as discussed in Section 4.2 - OPC UA Information Model Design in Chapter 4 - Detailed Design.

5.2.1. UaModeler Version and Namespace Configuration

The modelling was done by using UaModeler Version 1.6.4 459 version adhering to OPC UA standards for namespace 0 and configurations for both generic and manufacturer namespaces for flexible modelling. OPC Foundation creates namespace 0, while two additional namespaces were created for this project which are explained in Subsection 4.2.1 - Namespace Structure and Object Instantiation in Chapter 4 - Detailed Design.

- **Namespace 1:** Based on the EU-Rail and EULYNX Interface Specification SDI, this namespace defines standard templates for trackside subsystems e.g. Points, Train Detection, and Light Signals.
- **Namespace 2:** A manufacturer-specific instance built on top of Namespace 1, showing how suppliers implement diagnostic information using these templates.

Figures 5.1a and 5.1b below represent practical implementations of these namespaces with more detail on the Manufacturer and Generic models within UaModeler. These settings clearly separate the generic model based on the template and the manufacturer specific instance.



(a) Generic Namespace Configuration

(b) Manufacturer-Specific Namespace Configuration

Figure 5.1.: UaModeler: Overview of Namespace Configuration, own illustration

For this project, three main namespaces were used with the following URIs and versions:

```
NS=0, http://opcfoundation.org/UA/, Version 1.04.6 (OPC UA Core Types)
NS=1, http://twin.eulynx.eu/generic/, Version 4.3.1 (Generic Model)
NS=2, http://twin.eulynx.eu/manufacturerexample/, Version 4.3.1 (Manufacturer Example)
```

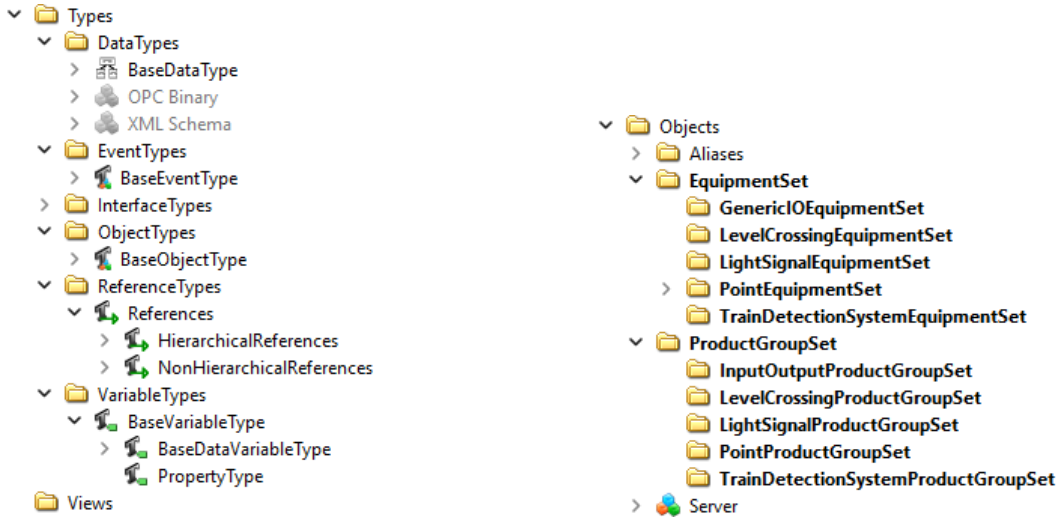
Figures 5.1a and 5.1b also shown each namespace has its specific declared parameters in model setting:

- **Namespace Prefix:** The namespace nodes are distinguished by the prefix: "elx" is for the Generic model and "man" is for the Manufacturer-specific model.
- **Namespace Name:** The Generic model has root element "elx", whereas the Manufacturer-specific model uses the root element "man" for supplier's implementations.
- **DLL Export Prefix:** When exporting code both namespaces used their prefixes "elx" for Generic and "man" for Manufacturer for class names.

By specifying these parameters in UaModeler, the two namespaces are built in a way which complies to both the OPC UA standard and EU-Rail and EULYNX Interface Specifications SDI. This design is modular, which allows for interoperability between different manufacturers' implementations and allows for simplified model management on both client and server systems.

5.2.2. Structure of the OPC UA Information Model

This implementation use Namespace 0 as the foundation for the overall structure of the OPC UA Information Model, while at the same time defining core types and standardized definitions from the OPC Foundation. As demonstrated in Subsection 4.2.2 - Key Elements of the OPC UA Information Model in Chapter 4 - Detailed Design, the key components of OPC UA forced modules, i.e., BaseDataType, BaseEventType, BaseObjectType, and Reference Type are necessary to build the custom namespaces in this project.



(a) Type Folder (b) Object Folder

Figure 5.2.: UaModeler: Overview of Information Model Structure, own illustration

Figure 5.2 shown the OPC UA Information Model is organized into two main folders: Types and Objects. Each important in defining and instantiating the diagnostic model:

- **Types Folder**: This has all the types needed to structure the diagnostic model.
  - **DataTypes**: Defines the types of data in the model for variables.
  - **EventTypes**: Types of events that can be triggered on the system are defined.
  - **ObjectTypes**: Different types of objects used in the model are defined.
  - **ReferenceTypes**: Described relationship between different nodes in the model.
  - **VariableTypes**: Defines the types of variables and their value structure.
- **Objects Folder**: Contains instance objects based upon the ObjectTypes defined in the Types folder.
  - **EquipmentSet**: Instances Object based on EquipmentType, such as PointEquipmentSet.
  - **ProductGroupSet**: Instance Object based on SubsystemType, e.g. PointProductGroupSet.

The Types and Objects folders are hierarchically organized while the custom namespaces (Namespace 1 and 2) provide flexible and standardized framework to design diagnostic systems. The resulting structure is able to facilitate instantiation of theinstance models by the suppliers in a way that maintains interoperability with different implementations of railway diagnostics.

5.2.3. Generic Model - Namespace 1

In UaModeler, Namespace 1 was created to define the core templates for each subsystem using the EU-Rail and EULYNX Interface Specification SDI. ObjectTypes and Variables are the main parts of this namespace and represent templates to create real world objects in Namespace 2. Almost all elements in this namespace closely follow the specifications in the detailed design phase, relative to the whole system architecture.

Key ObjectTypes and Additional Attributes

The basic unit of the Information Model is the ObjectType. There is an ObjectType for each subsystem or component, and Variables hold the data points that belong to these subsystems. During the modeling process, it is ensured that each component meets EU-Rail and EULYNX Interface Specifications SDI, which is described in Subsection 4.2.3 - Subsystem Representation in OPC UA in Chapter 4 - Detailed Design.

The following structure defines the essential attributes every ObjectType must contain to keep consistent operation and interoperability in the model.

Type

- **NodeClass:** Describes whether the node is Object or Variable.
- **Namespace:** The namespace it belongs to are specified. (e.g., http://twin.eulynx.eu/generic).
- **Name:** Represents a name of the node defining its role in the subsystem (e.g. BarrierType).
- **IsAbstract:** Specifies whether the ObjectType is abstract, and hence cannot be instantiated.

Additional Attributes

- **NodeId:** An unique identifier for each node within the namespace.
- **DisplayName:** Node names that are human readable (e.g., BarrierType).
- **BrowseName:** The name of the OPC UA Information Model that will be standardized.
- **Description:** The description of the ObjectType inclusing what it is, what it does (e.g., ‘A movable obstacle used to deter road traffic and pedestrians at level crossings’).

Children

- **NodeClass:** Determines if each child node is a Variable, or Object.
- **Name:** The name of the children nodes (e.g., BarrierStatus, ExpectedPosition).
- **TypeDefinition:** Describes the type of child node (e.g., BaseDataVariableType, PropertyType).
- **Modelling Rule:** Indicates whether the child node is Mandatory, Optional, Placeholder, or None.
- **DataType:** Specifies which type of data the Variable holds (e.g., Boolean, Integer).

References

- **ReferenceType:** Describes the type of relationship (e.g., GeneratesEvent).
- **Target:** A target node the reference points to (e.g., BarrierTurnEventType).

These ObjectTypes provide a high-level structure for subsystems in the railway environment. The examples above illustrate the attribute requirements for each ObjectType, while a comprehensive database of all ObjectTypes and their configurations is available in the supplementary Excel file provided with this project.

Variable

Each ObjectType is linked to Variables that capture the subsystem's diagnostic and operational status. These variables, defined by specific DataTypes (e.g., String, Enumeration, MultiStateDiscreteType) and governed by Modelling Rules, follow the attribute type mapping in Subsection 4.2.5 - Mapping SDI Attribute Types to OPC UA Node Types, and the DataTypes & VariableTypes describe in Subsection 4.2.6 - Mapping SDI Data Types to OPC UA Data Types in Chapter 4 - Detailed Design. The following Table 5.2. lists sample variables and their associated attributes:

Variable Name	Parent	DataType	TypeDefinition	Modelling Rule	NodeId
SubsystemIdentification	SubsystemType	String	PropertyType	Mandatory	ns=1;i=6343
IsTimeSynchronised	SubsystemType	Boolean	BaseDataVariableType	Optional	ns=1;i=6556
StatusTechnical	SubsystemType	StatusTechnical (Enumeration)	BaseDataVariableType	Mandatory	ns=1;i=6038

Table 5.2.: Sample Variables in Namespace 1

- **SubsystemIdentification:** A mandatory string variable that uniquely identifies the subsystem.
- **StatusTechnical:** An enumeration variable that reports the technical status of the subsystem.
- **IsTimeSynchronised:** An optional boolean variable indicate time synchronisation status

The variables listed above are only representative examples out of the larger set of variables implemented in Namespace 1. A detailed list of all implemented variables with their specifications is available in the supplementary Excel file accompanying this project.

Reference

In the implementation, ObjectTypes also include specific reference types (e.g. IsImplementedBy and GeneratesEvent), in addition to hierarchical references. These references define key relations between model components, as described in Subsection 4.2.4 - Mapping UML Relationships to OPC UA Information Models in Chapter 4 - Detailed Design. The Table 5.3 shows examples of ObjectTypes, ReferenceTypes, and target associations in the model.

ObjectType Name	ReferenceType	Target
SubsystemType	IsImplementedBy	EquipmentType
PointType	GeneratesEvent	PointTurnEventType
EquipmentType	IsPartOfRedundancyGroup	RedundancyStatus
OpcUaServerType	ServesInterface	SDI_Type and SMI_Type

Table 5.3.: Examples of ObjectTypes and Their Reference Types

- **IsImplementedBy:** Associates an ObjectType to its implementation, e.g. EquipmentType.
- **GeneratesEvent:** Show that the ObjectType can send event data to an EventType for logging.
- **IsPartOfRedundancyGroup:** Indicates that the ObjectType is inside a redundancy group.
- **ServesInterface:** Used to indicate that the ObjectType is serving as interface to some other ObjectType.

For instance, IsImplementedBy reference links the SubsystemType to its implementation (EquipmentType). This presents the same relationship whereby each EquipmentType in the field elements is linked to the SubsystemType, which represents the structured association of equipment with their respective subsystems in the system architecture. The complete implementation of all ObjectTypes and their references is documented in the supplementary Excel file, which serves as a comprehensive database for this model's reference structure.

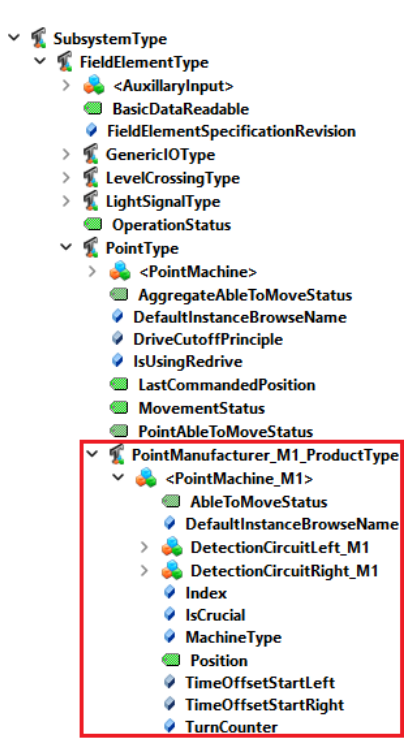
After having defined the templates in Namespace 1, Namespace 2 is implemented to instantiate all of the templates and use them to simulate actual railway components. These instances are necessary for validating system performance and to align with the design diagnostic and operational requirements as defined in Section 4.4.3 - Scenario Simulation and Failure Handling in Chapter 4 - Detailed Design.

5.2.4. Manufacturer Example Model - Namespace 2

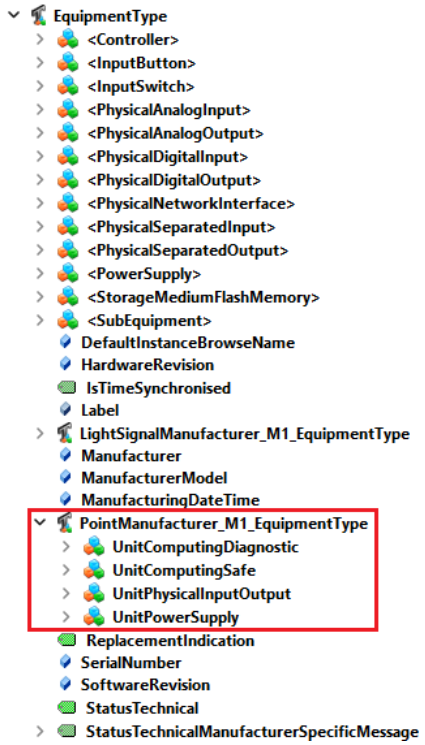
The manufacturer specific implementations are created by implementing Namespace 2 on top of the generic templates from Namespace 1. In the Chapter 4 - Detailed Design, Subsection 4.2.1 - Namespace Structure and Object Instantiation explained how namespace structure is reflected when instantiating objects, which aligns with the Points (S1–S15) and the respective Object Controllers as explained in Section 4.4.3 - Scenario Simulation and Failure Handling. To support instantiation of these objects, two custom ObjectTypes were implemented in Namespace 2, to accommodate attributes and behaviors specific to a manufacturer.

- **PointManufacturer_M1_ProductType:** Inherits the Generic PointType and contains attributes particular to the manufacturer's product, for example, detection circuits, and custom positions.
- **PointManufacturer_M1_EquipmentType:** Inherited from EquipmentType, it represents manufacturer specific object controllers with diagnostics and object and attributes related to hardware.

Extending generic templates in Namespace 1 with manufacturer specific attributes provides both standardization as well as flexibility. Two ObjectTypes developed in Namespace 2 exemplify how manufacturers can customize the given standardized templates to reflect their specific product requirements and thus align with the EU-Rail and EULYNX Interface Specification SDI.



(a) PointManufacturer_M1_ProductType



(b) PointManufacturer_M1_EquipmentType

Figure 5.3.: Custom ObjectType in Manufacturer Example Namespace, own illustration

Figures 5.3 show the structure of custom ObjectTypes in OPC UA Information Model which inherit key attributes from generic templates (PointType and EquipmentType), to support manufacturer specific needs with new variables and objects, while still maintaining interoperability. These adhere to the system architecture specified in Subsection 4.2.3 - Subsystem Representation in OPC UA in Chapter 4 - Detailed Design.

Instantiation of Point (S1–S15)

In Namespace 2, individual point instances (S1 through S15) are instantiated from the custom PointManufacturer_M1_ProductType ObjectType, which extends the generic PointType from Namespace 1. They include manufacturer specific attributes like custom diagnostics and detection circuits, which are manufacturer product line specific. This extension provides capability for enhanced monitoring and fault detection.

The Table 5.4 below represents a summary of representative point instances, their relationships, and references. The rest of the list follows the same pattern, changing only NodeId and instance name.

ObjectType Name	Parent	Reference Type	Target	NodeId
S1	PointProductGroupSet	IsImplementedBy GeneratsEvent	ObjectControllerPoint_S1_S5 PointTurnEventType	ns=2;i=5012
S2	PointProductGroupSet	IsImplementedBy GeneratsEvent	ObjectControllerPoint_S1_S5 PointTurnEventType	ns=2;i=5017
…	…	…	…	…
S6	PointProductGroupSet	IsImplementedBy GeneratsEvent	ObjectControllerPoint_S6_S10 PointTurnEventType	ns=2;i=5037
…	…	…	…	…
S14	PointProductGroupSet	IsImplementedBy GeneratsEvent	ObjectControllerPoint_S11_S15 PointTurnEventType	ns=2;i=5077
S15	PointProductGroupSet	IsImplementedBy GeneratsEvent	ObjectControllerPoint_S11_S15 PointTurnEventType	ns=2;i=5082

Table 5.4.: Point Instances in the Manufacturer Example Namespace

As Table 5.4 shows, using the "Reference Type" and "Target" columns, each point instance is connected to its corresponding object controller via IsImplementedBy. GeneratesEvent further enables each point to trigger PointTurnEventType event, supporting real-time. It enables real time transmission of diagnostic data and efficient coordination of the railway control system.

Instantiation of Point Machine (Point Machine S1 – Point Machine S15)

Similarly, each point instance also contains an associated PointMachine instance using the Modelling Rule "MandatoryPlaceholder," as it is necessary for executing the point's physical operations. The following table summarizes representative point machine instances and their relationships. Like the point instances, the full list of point machines follows the same pattern.

ObjectType Name	Parent	NodeId
PointMachine_S1	S1	ns=2;i=5015
PointMachine_S2	S2	ns=2;i=5020
…	…	…
PointMachine_S6	S6	ns=2;i=5040
…	…	…
PointMachine_S14	S14	ns=2;i=5080
PointMachine_S15	S15	ns=2;i=5085

Table 5.5.: Point Machine Instances in the Manufacturer Example Namespace

Since all the PointMachine instances are directly linked to their respective point instances (S1–S15) without additional specific references, the columns for ReferenceType and Target are omitted in this table.

Instantiation of Object Controllers Points (OCP S1_S5 - OCP S11_S15)

Grouping of object controllers that control multiple points is in namespace 2. All of these object controllers are created by instantiating custom PointManufacturer_M1_EquipmentType ObjectType from Namespace 1 that extends generic EquipmentType from Namespace 1

Each object controller covers a range of points (e.g. S1–S5) and manages their operations and diagnostics on the basis of the particular on site operating area. This is an industry relevant, practical approach to communication between subsystems, transmission of diagnostic data and monitoring of the railway system.

ObjectType Name	Parent	Managed Points	NodeId
ObjectControllerPoint_S1_S5	PointGroupSet	S1, S2, S3, S4, S5	ns=2;i=6001
ObjectControllerPoint_S6_S10	PointGroupSet	S6, S7, S8, S9, S10	ns=2;i=6002
ObjectControllerPoint_S11_S15	PointGroupSet	S11, S12, S13, S14, S15	ns=2;i=6003

Table 5.6.: Object Controller Point Instances and Managed Points

As summarized in the Table 5.6, each object controller manages points which reflect an on site infrastructure management. To give further illustration, four key units of each ObjectControllerPoint is described. Collectively, these units handle the operational status and fail safe features of the points. Each unit and it's corresponding role is outlined in Table 5.7.

Component	Description
UnitComputing Diagnostic	Manages diagnostic data collection and processing, containing: • **Controller D01**: Diagnostic controller for data processing. • **PhysicalNetworkInterface D01**: Network interface for communication. • **UCD Storage**: Storage unit for diagnostic data.
UnitComputingSafe	Manages safe operations and redundancy, containing: • **Controllers (S01–S03)**: Safe controllers with their own redundancy objects. • **RedundancyStatus Controller S01–S03**: Links to Controller Redundancy Group. • **PhysicalNetworkInterfaces (S01–S02)**: Network interfaces for safe communication. • **RedundancyStatus PhysicalNetworkInterface S01–S02**: Links to Physical Network Interface Redundancy Group. • **UCS Storage**: Storage unit for safe operation data.
UnitPhysical Input/Output	Manages physical inputs and outputs for Points, containing: • **PointOut (1-5)**: Drives specific Points (SXX) in the system.
UnitPowerSupply	Manages power supply and redundancy, containing: • **PointPowerSupply (01-05)**: Supplies power to Points. • **RedundancyStatus P01 -P05**: Tracks redundancy status and links to PointPowerSupplyRedundancyGroup.

Table 5.7.: Structure of ObjectControllerPoint

Redundancy Groups: To ensure system reliability, redundancy mechanisms are implemented in the key units. The secondary purpose is to serve as failover mechanisms so that if a component is down, the other one takes over and operations continue. For maintaining uninterrupted system performance, the redundancy groups are organized by below function:

- **Controller Redundancy Group**: Provides redundancy of controllers to enable backups to take over controllers during failure.
- **Physical Network Interface Redundancy Group**: Provides redundant for network interfaces, guarantees continuous communication.
- **Point Power Supply Redundancy Group**: Provides redundancy for power supplies in case of unit failures to keeps power at an acceptable level.

To reduce downtime in case of component failure, necessary for continuous railway operation, we employ these redundancy groups. While these are not included within this project's scenarios, they are consistent with the SDI Task Force's plan to show standard on-site practices to manufacturers for typical system reliability.

Variables for Point and Object Controller Instances

The implementation of variables for the S1-S15 Points and Object Controller Points (OCP) instances allow to monitor and diagnose system operation and performance. Based on both generic and manufacturer specific namespaces these variables provide real time information used to track the system health and detect problems.

Due to the high number of implemented variables within the system, only the selected examples are provided in Appendix A.2.2 - Variables and Enumeration for OPC UA Instances. Such examples are concentrated on the most relevant attributes and relations excluding the whole dataset, which can overwhelm the recipient.

- **Point Instance Variables**: Each point instance includes key variables, (e.g., LastCommandedPosition, MovementStatus), which keep track of the point's operated state. This section also includes variables for PointMachine instances associated (Physical operations, e.g. switching and holding positions).
- **Object Controller Point Variables**: These variables are in terms of Object Controller Points that control multiple points. The appendix highlights:
  - **Redundancy Status and Operation**: Provides Tracks redundancy status and operation for Controller and its group to be fault tolerant.
  - **Network Communication Diagnostics**: Monitors network diagnostics namely PhysicalNetworkInterface state for stable data transmission.
  - **Storage Capacity and Health**: Monitors the storage capacity and health for UnitComputingSafe's storage, critical to integrity of the stored data.

  These variables are particularly important in order to monitor system health and performance under both normal and failure conditions.
- **Enumeration Data Types**: Enumeration data types for Point and Object Controller instances defining operational states (for example PointPosition, StatusTechnical) required for interpreting system diagnostics is also included in the appendix.

Hence, The Appendix A.2.2 - Variables and Enumeration for OPC UA Instances is a detailed reference with examples of already implemented variables, their data types, node identifiers (NodeIds), and connections to other nodes in the OPC UA Information Model. Despite the full set, used in the system, being far more comprehensive, these examples give insight into the structure and purpose of the main variables included.

5.2.5. Examples of Creating Types and References in UaModeler

The ObjectTypes, Variables, DataTypes and ReferenceTypes of UaModeler are implemented in a structured manner to comply to the EU-Rail and EULYNX Interface Specification SDI. These elements provide the OPC UA Information Model template, as a basis for defining railway subsystems and diagnostic data.

Detailed implementation examples of creation process of ObjectType, Variable, DataType and ReferenceType are provided in Appendix A.2.1 - UaModeler Implementation Examples, to keep the main document in focused. The appendix offers full examples of all the core implementation principles described below.

ObjectTypes

ObjectTypes are used to define the structure and behavior of a subsystem/component, and are key attributes, such as NodeClass, Namespace, Name and IsAbstract. Relationships between system components are also established using ObjectTypes. A detailed example of implementing SubsystemType as an ObjectType is provided in the Appendix A.2.1 - Example of ObjectType Creation in UaModeler.

Variables

Operational and diagnostic data that must be monitored is captured by Variables linked to ObjectTypes. These rules are followed for their inclusion into the subsystem, either mandatory or optional. Detailed creation steps of each class and key variables such as LastCommandedPosition or SubsystemIdentification are created to track the system status, as described in Appendix A.2.1 - Example of Variable Creation in UaModeler.

Enumeration DataTypes

Standardized value sets for variables are defined with Enumeration DataTypes, which confirm consistent data representation. For instance, the positions available in the PointPosition enumeration are Left, Right, and UnintendedPosition. A example of implementing Enumeration DataTypes is provided in Appendix A.2.1 - Example of Enumeration DataType Creation in UaModeler.

ReferenceTypes

ReferenceTypes define activity relationships between nodes, e.g. connecting a subsystem to its equipment or an event trigger. The IsImplementedBy ReferenceType is used to link subsystems to EquipmentType, is explained in Appendix A.2.1 - Example of ReferenceType Creation in UaModeler.

Thus, these structured implementations facilitated development of a robust, standards compliant OPC UA Information Model that characterized the detailed representation of railway subsystems and diagnostic data.

5.2.6. Exporting the Generic and Manufacturer Example Namespace as XML

Once the OPC UA Information Model has been defined in UaModeler the next step is to export Generic (Namespace 1) and Manufacturer Example (Namespace 2) namespaces in XML format. The models are exported so that the model can be easily integrated in to OPC UA server code allowing real world application and alignment to EU-Rail and EULYNX Interface Specification SDI, as detailed in Subsection 4.2.11 - Exporting the OPC UA Model as XML in Chapter 4 - Detailed Design.

Export Procedure in UaModeler

The following is the steps of exporting namespaces to XML in UaModeler.

1. **Navigate to the Project Panel:** Open the UaModeler project and choose which Namespace to export (the Generic or Manufacturer Example namespace).
2. **Select Export XML Option:** In Figure 5.4, right click on the namespace and choose the option Export XML from the context menu. This produces an XML representation of the Information Model, i.e. the ObjectTypes, Variables and References.
3. **Configure Export Settings:** Configure the following in the export dialog:
   - **Namespace URI:** Enter the Namespace URI as described in Subsection 5.2.1 - UaModeler Version and Namespace Configuration in Chapter 5 - Implementation.
   - **Include References:** Keep all references to preserve node relationships.
   - **XML Schema:** To make compatibility with OPC UA servers, use OPC UA XML schema.
4. **Save the XML File:** Save the XML file for each namespace. Example filenames:
   - Namespace 1 exported as "eulynx.generic.bl4r3.rev01.xml".
   - Namespace 2 exported as "eulynx.manufacturer.example.bl4r3.rev01.xml".

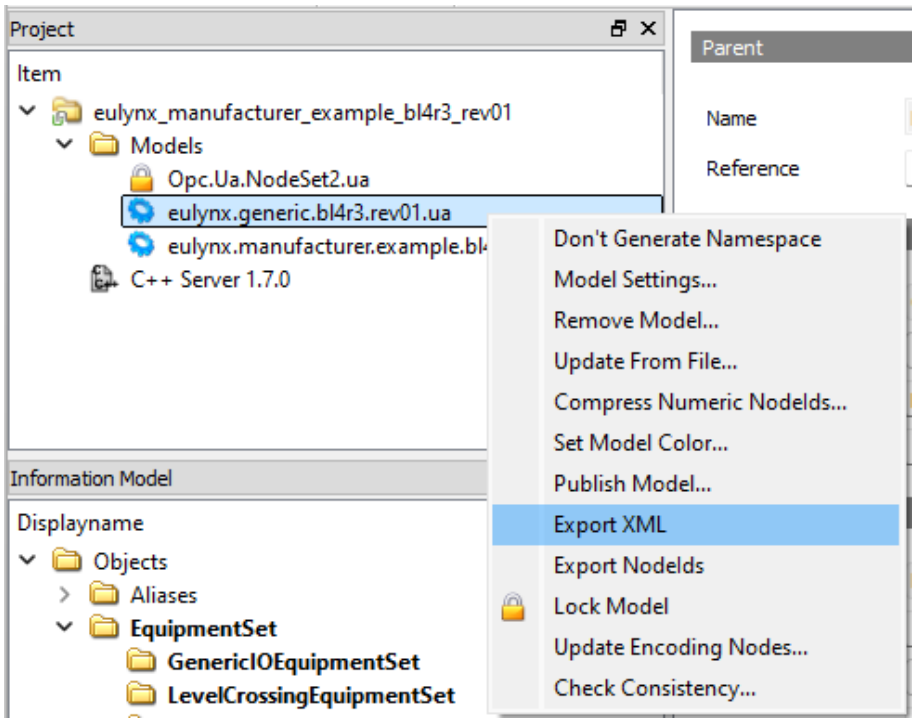


Figure 5.4.: UaModeler: Exporting Namespace as XML, own illustration

Structure of Exported XML Files

The exported XML files contain all the elements necessary to recreate all the OPC UA Information Model within the server. The XML structure contains:

- **NodeSet:** The element that contains all nodes including ObjectTypes, Variables and References.
- **Node:** Represented by a Node element, that has attributes of NodeId, BrowseName, and DisplayName.
- **Reference:** Relationships between nodes are described by this (e.g. IsImplementedBy)
- **DataType:** Within DataTypes, custom data types such as enumerations are defined for consistent interpretation of diagnostic values.

The only difference between Namespace 1 (Generic) and Namespace 2 (Manufacturer Example) is the amount of detail. Namespace 1 offers a core, generic structure and Namespace 2 enhances this with manufacturer specific detail, including specialized ObjectTypes and other redundancy related variables. With this, Namespace 2 is able to provide manufacturer specific diagnostics.

The following is the example exported XML of Namespace 1 (Generic):

```
<NodeSet xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
    <NamespaceUris>
        <Uri>http://twin.eulynx.eu/generic/</Uri>
    </NamespaceUris>
    <Node NodeId="ns=1;i=1001">
        <BrowseName>1:PointType</BrowseName>
        <DisplayName>PointType</DisplayName>
        <References>
            <Reference ReferenceType="HasComponent" IsForward="true">ns=1;i=1002</
                Reference>
        </References>
    </Node>
</NodeSet>
```

Namespace 2 (Manufacturer Example) adds additional references to specified types and components:

```
<NodeSet xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
    <NamespaceUris>
        <Uri>http://twin.eulynx.eu/manufacturerexample/</Uri>
        <Uri>http://twin.eulynx.eu/generic/</Uri>
    </NamespaceUris>
    <Node NodeId="ns=2;i=1501">
        <BrowseName>2:PointManufacturer_M1_ProductType</BrowseName>
        <DisplayName>PointManufacturer_M1_ProductType</DisplayName>
        <References>
            <Reference ReferenceType="HasComponent" IsForward="true">ns=2;i=1502</
                Reference>
        </References>
    </Node>
</NodeSet>
```

As a result, Namespace 2 is based on the generic foundation but adds manufacturer specific attributes and references.

After implementing OPC UA Information Model in UaModeler and exporting the implemented model as XML the next step is to set up scenario simulations to be used in Excel. These scenarios were built based on the NodeIDs configured in Namespace 1 and Namespace 2, mapping similarly to the structured model. This allows for verification and validation of the OPC UA server, that it is working as planned, has a correct hierarchy and stores proper event logs when actions happen that are relevant for use. The Excel file, furthermore, will be used to fulfill requirements validation as on through this process.

5.3. Scenario Simulation and Failure Handling Implementation

This section continues with the OPC UA Information Model design and XML exporting to also explain the implementation of scenario simulations and failure handling of the OPC UA server. Using Excel to configure scenario data for base case and expected values, as described in Section 4.3 - Scenario Simulation and Failure Handling Design in Chapter 4 - Detailed Design, the outlined design allowed system response to be tested under specific conditions.

5.3.1. Scenario Setup and NodeID Configuration

In initial implementation, the NodeIDs associated with Point S10 and Controller Object Points S6-S10 are organized according to critical and non-critical scenarios discussed in Subsection 4.3.1 - Motivation for Scenario Selection in Chapter 4 - Detailed Design. Key NodeIDs, nodes type and variables are placed in excel, focusing on Unit Computing Safe for controller side.

5.3.2. Excel Structure for Scenario States

The full tables for each scenario are available in the Appendix A.2.3 - Detailed Variable States for Scenario Simulation, which represent the variable states and NodeID configurations at each stage for further reference, as there are extensive details. There are three worksheets for different states in the Excel file:

1. **Default State**: For normal operation represents that all systems are working as expected.
2. **Scenario 1 - Critical Failure**: Sets up a failure where Point S10 fails to get to its end position (see Subsection 4.3.3 - Scenario 1: Point Switch Failure in Chapter 4 - Detailed Design).
3. **Scenario 2 - Non-Critical Failure**: Failure of Unit Computing Safe (see Subsection 4.3.4 - Scenario 2: Object Controller Failure in Chapter 4 - Detailed Design).

Variable changes between simulation stages are shown in the tables for verification. They are in the appendix for easy reference and main document clarity because of their size.

5.3.3. Integration of Excel Data into OPC UA Server

Simulation scenarios use Excel data as structured input. The server simulates both normal and failure events by loading NodeID and diagnostic data for each state, and then generates alerts, recovers, and tracks the recovery. This will verify server's responses to each scenario, retaining the hierarchical structure, logging. See Appendix A.2.3 - Detailed Variable States for Scenario Simulation for Scenario 1 (Figure A.11) and Scenario 2 (Figures A.12 and A.13).

5.3.4. Scenario Verification and Requirements Validation

The OPC UA server's functionality specified in UaModeler is validated using these Excel-based scenarios. Diagnostic responses were monitored using UaExpert to confirm that the server meets project requirements and that it continuously logs event related changes. The formatted Excel file is used as a structured reference to check these capabilities throughout testing.

Hence, This implementation of scenario simulations (see Appendix A.2.3 - Detailed Variable States for Scenario Simulation) verifies that the OPC UA Information Model effectively represents system behavior and handles failures as required in Section 4.3 - Scenario Simulation and Failure Handling Design in Chapter 4 - Detailed Design. A technical foundation and core components for the OPC UA server are described in the following section. The server is implemented in Python adopting real time data diagnostics and asynchronous operations to support the effective exchange of information between railway field elements and the processing of simulation scenarios.

5.4. OPC UA Server Implementation

The implementation of the OPC UA server is done using asyncua, asyncio, pandas, and os libraries. The XML files are loaded from UaModeler to deliver real time diagnostics data on railway field elements, according to the design principles of Section 4.4.1 - OPC UA Server Implementation Design in Chapter 4 - Detailed Design. Finally, the server processes scenario data from Excel files and dynamically updates node values during runtime, enabling efficient communication with the connected OPC UA clients through use of asynchronous operations. This chapter explains core tools and processes that are needed to implement the server. The complete implementation code is available in Appendix A.2.4.

5.4.1. Key Python Libraries and Tools

To support real-time data handling, concurrent tasks, and interaction with the OPC UA Information Model, the server integrates essential below libraries according to the modular approach in Subsection 4.4.2 - Core Libraries and Tools in Chapter 4 - Detailed Design.

```
import asyncio
import pandas as pd
from asyncua import Server, ua
import os
```

- **asyncio**: Supports asynchronous operations meaning that the server can work on multiple things at once without blocking real time updates or communication.
- **pandas**: For data manipulation and analysis, it loads node information and scenario steps from an Excel files using openpyxl to handle .xlsx files, facilitating dynamic diagnostic updates.
- **asyncua**: Provides real-time data communication and interaction with the Information Model through a fully asynchronous OPC UA protocol implementation in Python.
- **os**: Interacts with the operating system to set environment variables e.g server name, endpoint setting, port configuration.

By using these libraries these libraries are integrated into the server to perform non blocking, real time operations, updates values dynamically in the node, and communicate with the connected clients.

5.4.2. Loading the Information Model

The server loads generic and manufacturer namespace XML files to define the data structure, aligning with the modular design described in Subsection 4.4.3 - Server Initialization in Chapter 4 - Detailed Design.

```
server = Server()
await server.init()

# Set up the OPC UA server endpoint and server name
server_name = os.getenv("OPCUA_SERVER_NAME", "BL4R3 SDI OPC UA")
server_port = os.getenv("OPCUA_SERVER_PORT", 4840)
endpoint = os.getenv("OPCUA_ENDPOINT", f"opc.tcp://0.0.0.0:{server_port}/EULYNX")
server.set_endpoint(endpoint)
server.set_server_name(server_name)

# Import the generic and manufacturer namespaces
await server.import_xml("eulynx.generic.bl4r3.rev01.xml")
await server.import_xml("eulynx.manufacturer.example.bl4r3.rev01.xml")
await server.import_xml("mdm.rev01.xml")
```

The server is initialised and the XML files defining the Generic Information Model are loaded by this code. For flexibility, it configures server name and endpoint using environment variables. To load the instances created in UaModeler such as S1, S2 and ObjectControllerPoint_S6_S10, the server also imports the manufacturer namespace from the eulynx.manufacturer.example.bl4r3.rev01.xml file.

5.4.3. Scenario Loading and Node Updates

OPC UA server loads and processes Excel data during runtime with pandas to enable real time scenario simulation, and the updates of bulk nodes and an event generation during runtime, as described in Subsection 4.4.3 - Scenario Simulation in Chapter 4 - Detailed Design.

Loading Node Information from Excel

As presented in Subsection 4.4.3 - Node Data Loading in Chapter 4 - Detailed Design, pandas is used to read node information defined in an external Excel file into the server. The function load_node_info obtains a name, namespace and data type to update dynamically at runtime.

```python
# Load node information from Excel file
def load_node_info(file_path, sheet_name):
    """
    Parameters:
        file_path (str): The path to the Excel file.
        sheet_name (str): The name of the sheet to read from.
    Returns:
        list[dict]: A list of dictionaries containing node information.
    """
    df = pd.read_excel(file_path, sheet_name=sheet_name, engine='openpyxl')
    return [
        {
            "name": row["Name"],
            "datatype": row["DataType"].replace("(Enumeration)", "").strip(),
            "ns": int(row["ns"]),
            "i": int(row["i"]),
            "value": row.get("Value", None)
        }
        for _, row in df.iterrows()
        if row["NodeClass"] == "Variable"
]
```

This function reads node data from Excel sheet and stores it into a list of dictionaries to have flexible access to the node data during runtime.

Cleaning Values and Data Validation

The values are cleaned and converted using clean_value function before writing data to OPC UA nodes, protecting data integrity on nodes, especially for critical diagnostic variables such as position or fault status.

```python
# Clean values from the Excel file
def clean_value(value, datatype):
    """
    Parameters:
        value: The value to be cleaned.
        datatype (str): The datatype of the value.
    Returns:
        The cleaned value in its appropriate type or None if not applicable.
    """
    if pd.isnull(value):
        return None
    if "Int" in datatype:
        return int(value)
    elif "Boolean" in datatype:
        return bool(value)
    elif "Double" in datatype:
        return float(value)
    elif "String" in datatype:
        return str(value)
    return value
```

Loading Scenario Steps

Separate Excel sheets are created which define Scenario steps, and these are processed into a series of updates which the server applies to multiple nodes in parallel. The load_scenario_steps function reads and transforms this data so it can be input into scenario based simulations.

```
# Load scenario steps from Excel file
def load_scenario_steps(file_path, sheet_name):
    """
    Parameters:
        file_path (str): The path to the Excel file.
        sheet_name (str): The name of the sheet to read from.
    Returns:
        list[dict]: A list of dictionaries containing scenario step data.
    """
    df = pd.read_excel(file_path, sheet_name=sheet_name, engine='openpyxl')
    steps = []
    for step_column in df.columns:
        if step_column.startswith("Step"):
            step_data = {row["Name"]: clean_value(row[step_column], row["DataType"]) for _
                                                                    , row in df.iterrows()}
            steps.append(step_data)
    return steps
```

These steps are used by the server to update node values in real time, for instance, simulate different operational scenarios like system failure or normal operation.

5.4.4. Declaring Node IDs for Event Generation

In UaModeler Object instances (e.g., S1, S2, ObjectControllerPoint_S6_S10) are instantiated and saved in the manufacturer namespace XML file to support event driven architecture as specified in Subsection 4.4.3 - Event Handling in Chapter 4 - Detailed Design. However, the server still has to declare node IDs for these instances to enable event generation against them at runtime.

```
# Declare the node IDs for event generation
point_turn_event_type_node = server.get_node("ns=2;i=1123")
instance_objects = [server.get_node("ns=4;i=5057")]  # Example for Point S10
print(f"OPC UA Server '{server_name}' is running on endpoint: {endpoint}")
```

This code refer to the instance of S10 (ns=4;i=5057) that has been instanciated in UaModeler and imported from XML files. This node ID needs to be declared because the server generates events attached to it that happen during runtime like failure notifications or status updates.

5.4.5. Node Setup and Real-Time Data Updates

The OPC UA server dynamically updates node values during runtime, based on diagnostic variables defined within the Information Model (e.g., Position, Fault Status), mapped to object instances in the manufacturer namespace. This approach, detailed in Subsection 4.4.3 - Real-Time Data Updates in Chapter 4 - Detailed Design, enables real-time access and feedback for connected clients.

Single Node Value Update

The OPC UA server's single node value is updated by the update_node_value function. First, it finds the node out in terms of its namespace and identifier, then it reads what data type it expects for the input value to comply. The function converts it as needed, if the value is valid (for example, from timestamp to datetime object) and writes it to the node. In real time updates, this function allows an accurate reporting of status of the individual field elements.

```python
# Update the value of a specific node
async def update_node_value(server, node_info, value):
    """
    Parameters:
        server (Server): The OPC UA server instance.
        node_info (dict): Information about the node to be updated.
        value: The value to be written to the node.
    Returns:
        None
    """
    node = server.get_node(ua.NodeId(node_info["i"], node_info["ns"]))
    try:
        data_type = await node.read_data_type_as_variant_type()
        cleaned_value = clean_value(value, data_type.name)
        if cleaned_value is None and data_type != ua.VariantType.String:
            print(f"Skipping node: {node_info['name']} due to None value and incompatible type.")
            return

        if isinstance(cleaned_value, pd.Timestamp):
            cleaned_value = cleaned_value.to_pydatetime()

        variant_value = ua.Variant(cleaned_value, data_type)
        await node.write_value(variant_value)
        print(f"Updated {node_info['name']} (ns={node_info['ns']}; i={node_info['i']}) to {cleaned_value}")
    except Exception as e:
        print(f"Error writing to {node_info['name']} (ns={node_info['ns']}; i={node_info['i']}): {e}")
```

This function has been found to be very flexible to use in cases where individual node values need to be updated, for example, real-time diagnostic scenarios.

Asynchronous Bulk Node Updates

The update_data_access_view function searches all nodes and applies steps based on the given scenario. It is useful for simulations such as failure, or repair scenarios, that require concurrent node updates, and it uses update_node_value function to update each node individually.

```python
# Update Data Access View nodes for a scenario step
async def update_data_access_view(server, nodes, scenario_step=None):
    """
   Parameters:
        server (Server): The OPC UA server instance.
        nodes (list[dict]): A list of node information dictionaries.
        scenario_step (dict, optional): The data for the current scenario step.
    Returns:
        None
    """
    for node_info in nodes:
        if node_info["name"] in EVENT_SPECIFIC_NODES:
            continue
        value = scenario_step.get(node_info["name"], node_info.get("value")) if scenario_step else node_info.get("value")
        if value is not None:
            await update_node_value(server, node_info, value)
```

The function is important in order to update multiple nodes during scenario simulations involving operational scenarios (e.g., normal operations, failure modes) in real time. The system will process the updates efficiently in an asynchronous manner with the function itself meeting the real time performance requirements of the system.

Therefore, it is necessary to distinguish between these two functions for the reason of scope and purpose:

- **update_node_value**: This functions update single node, suited for real time data change.
- **update_data_access_view**: This function iterates over a list of nodes, updates them all concurrently by calling update_node_value for each. It is especially useful for scenario based simulations.

In summary, update_node_value function is designed to update single node whereas update_data_access_view function mainly updates bulk data across different nodes during the simulations of different scenarios.

5.4.6. Event Generation and Handling

As explained Subsection 4.2.9 - EventTypes in Chapter 4 - Detailed Design, to notify clients of significant railway field element state changes, real-time events are created by the OPC UA server. PointTurnEventType and BarrierTurnEventType will be triggered on Points and Barriers state change for accurate and instant monitoring.

Events are linked to object instances (e.g., Point S10) imported from the manufacturer namespace by the server. This triggers the event to send to clients and dynamically update event specifc nodes (e.g. isEndpositionReached, commandedPosition) based on scenario data. Below is Python code of how events are triggered for object instances:

```
# Trigger events for event-specific nodes
async def trigger_event(server, event_type_node, instance, step_data, nodes):
    """
    Parameters:
        server (Server): The OPC UA server instance.
        event_type_node (Node): The event type node.
        instance (Node): The instance node.
        step_data (dict): The data for the current scenario step.
        nodes (list[dict]): A list of dictionaries containing node information from the
                                                    Excel file.
    Returns:
        None
    """
    event = await server.get_event_generator(event_type_node, instance)
    event.event.Severity = 500
    event.event.Message = ua.LocalizedText("Event triggered with scenario values")

    for node_info in nodes:
        if node_info["name"] in EVENT_SPECIFIC_NODES:
            value = step_data.get(node_info["name"], False if node_info["datatype"] == "
                                                    Boolean" else 0)
            variant_type = ua.VariantType.Boolean if node_info["datatype"] == "Boolean"
                                                    else ua.VariantType.Int32
            setattr(event.event, node_info["name"].capitalize(), ua.Variant(value,
                                                    variant_type))

    await event.trigger()
```

In this function, the server creates events associated with instances such as Point S10. Scenario data updates the event specific nodes and triggers the event to inform clients of update.

This mechanism for event generation is critical to real time monitoring and diagnostics which allows clients to remain up to date as the system changes. It is designed to only generate events for instances of objects without alarm or fault notifications in line with the expected design concept in Subsection 4.4.1 - Design Principles in Chapter 4 - Detailed Design.

5.4.7. Scenario Simulation Loop

As described in Subsection 4.4.3 - Scenario Simulation in Chapter 4 - Detailed Design, the OPC UA server provides scenario based simulation capability by looping through predefined scenarios, dynamically changing node values and triggering event based on given scenario data.

```
scenarios = ["default", "1", "default", "2"]

async with server:
    while True:
        """
        Loop scenario playback
        """
        for scenario_choice in scenarios:
            print(f"Playing scenario: {scenario_choice}")
            nodes = load_node_info("BL4R3-rev01-Diagnostic_NodeID_List-scenario.xlsx",
                                                    sheet_name=scenario_choice)

            if scenario_choice == "default":
                await update_data_access_view(server, nodes)
            else:
                scenario_steps = load_scenario_steps("BL4R3-rev01-Diagnostic_NodeID_List-
                                                        scenario.xlsx", sheet_name=
                                                        scenario_choice)
                for step_data in scenario_steps:
                    await update_data_access_view(server, nodes, step_data)
                    for instance in instance_objects:
                        await trigger_event(server, point_turn_event_type_node, instance,
                                                                step_data, nodes)
                    await asyncio.sleep(5)

            print(f"Scenario {scenario_choice} completed.")
            await asyncio.sleep(15)

        print("Completed all scenarios, starting over...")
```

The above code provides an example of the scenario loop that the server uses to playback a series of operational scenarios defined in an external Excel file. Different system states, including normal operation or failure modes, can be simulated in each scenario and thus test the server's behaviour and performance.

As a result, the server conducts dynamic node updates and event handling to emulate actual scenarios for testing and diagnostic purposes. By bulk updating of nodes during scenario simulations and real time event generation, a robust and scalable framework for managing railway field elements is provided.

5.4.8. Flowchart Summary of OPC UA Server Implementation

Figure 5.5 provides a flowchart of the main steps and functions used to provide a high level overview of the implementation OPC UA server. The structure for the whole project is illustrated in this diagram, from the initialization of modules and constants to the main asynchronous loop and the handling of simulations for scenarios.

The detailed code explanation is complemented with this flowchart which visualizes the sequential set of actions taken by the server. It begins with import of necessary modules and constants, then helper functions, the server initialization, importing XML files, and finally the scenario loop and event triggering functions. As a result, it gives readers a high level view of the server core processes and helps their understanding of the whole implementation.

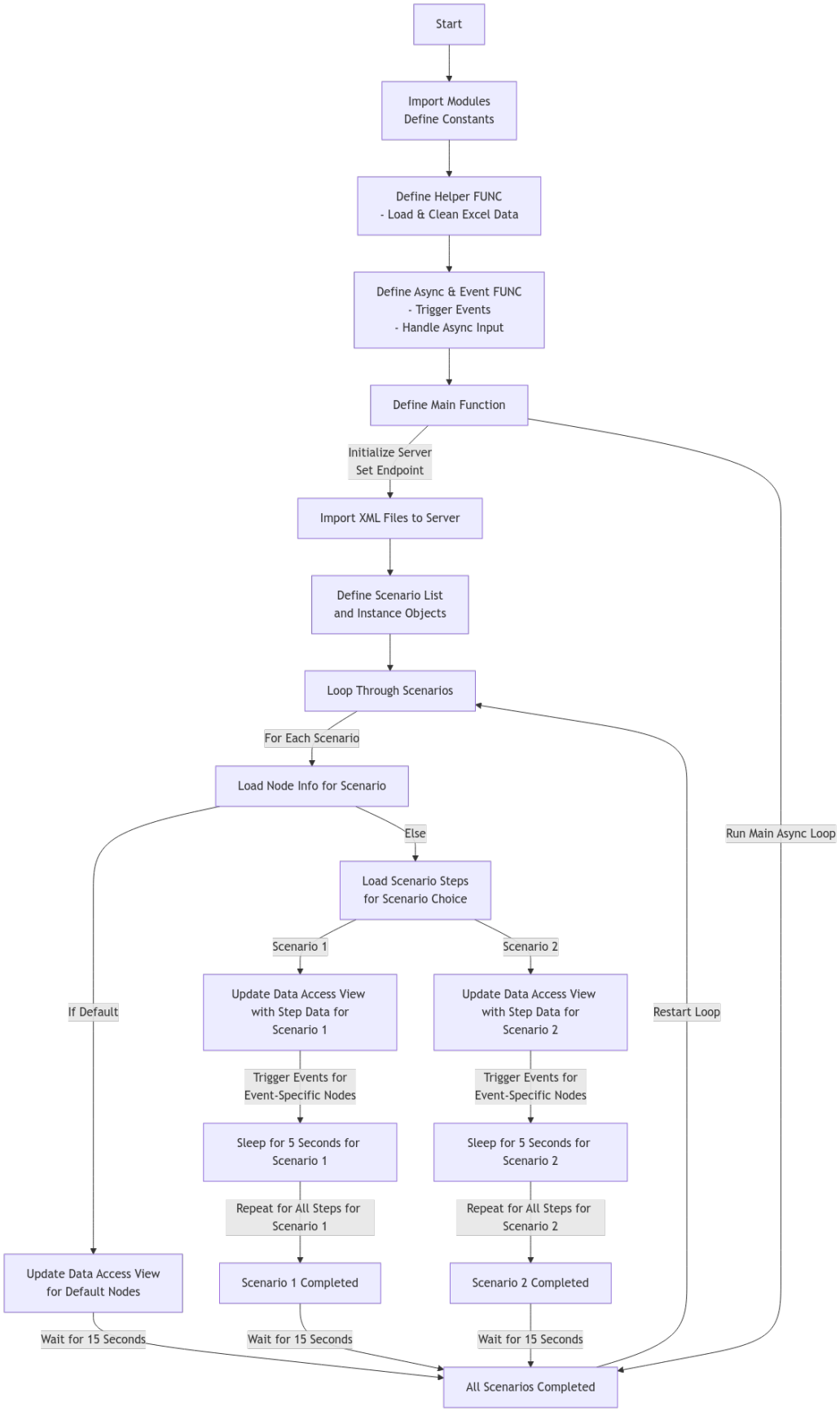


Figure 5.5.: Flowchart summarizing the OPC UA server implementation process, own illustration

The next phase involves making the core OPC UA server that processes the real time data ready for deployment. A repository is structured in order to store, administer and deploy the server along with its configurations. The following section explains the configuration and structure of repository, explaining how each part contributes to a the deployment process.

5.5. Repository Structure and Setup Configuration

For efficient deployment, configuration, and automation, a diagnostics system based on OPC UA is implemented with a repository structured into four main folders. With this implementation adhering to a structured approach, each folder is oriented toward a particular set of configurations used in real-time operation. These configurations are detailed within the folder structure, as shown in Figure 5.6, conformed to architectural specifications in Section 4.5 - Deployment Architecture in Chapter 4 - Detailed Design.

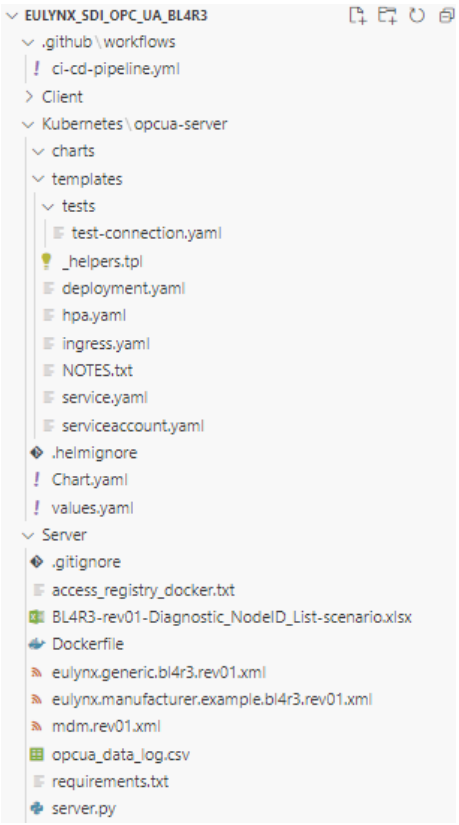


Figure 5.6.: Repository Structure for OPC UA Server Deployment, own illustration

The repository includes the four following directories:

- **.github**: As described in Subsection 4.5.7 - CI/CD Pipeline Integration in Chapter 4 - Detailed Design, this directory has "workflows" folder with the "ci-cd-pipeline.yml" file, containing instructions for the test, build and deployment of application using GitHub Actions.

  Regarding the design phase, the pipeline stages are building, pushing to the registry, and deploying and testing within a sandboxed Kubernetes environment. Appendix A.2.5 - CI/CD Pipeline Configuration (ci-cd-pipeline.yml), contains the implementation configurations for this pipeline.

- **Client**: Although present in the repository, this directory will be neglected, as it was included solely due to a colleague's involvement.

- **Kubernetes\opcua-server**: This directory describes the primary Kubernetes configurations and Helm charts for deploying the OPC UA servers (see Subsection 4.5.5 - Kubernetes Orchestration and Subsection 4.5.6 - Helm for Configuration Management in Chapter 4 - Detailed Design). In this directory key files include:
  - **Chart.yaml**: Defines the metadata for the Helm chart and implementation configurations as given in Appendix A.2.5 - Chart Metadata (Chart.yaml).
  - **values.yaml**: Provides configuration values for Helm, enabling parameterization of deployments, such as replica count and update strategy,

    Explained in Subsection 4.5.6 - Centralized Configuration with values.yaml in Chapter 4 - Detailed Design. Detailed implementation can be seen in Appendix A.2.5 - Values File (values.yaml).
  - **templates**: Contains templates files for Kubernetes resources, as outlined in Subsection 4.5.6 - Templated Deployment and Services in Chapter 4 - Detailed Design:
    - **deployment.yaml** – Specifies the deployment of OPC UA server pods.

      Outlined in Subsection 4.5.5 - Pod Management and Replica Control in Chapter 4 - Detailed Design, with detailed implementation in Appendix A.2.5 - Deployment Configuration (deployment.yaml).
    - **service.yaml** – Configures the NodePort service for external access.

      Detailed in Subsection 4.5.5 - Networking and Service Exposure in Chapter 4 - Detailed Design, with detailed implementation in Appendix A.2.5 - Service Configuration (service.yaml).
    - **hpa.yaml** – Configures horizontal pod autoscaling based on resource usage.

      Covered in Subsection 4.5.5 - Horizontal Pod Autoscaling (HPA) in Chapter 4 - Detailed Design), with detailed implementation in Appendix A.2.5 - Horizontal Pod Autoscaling (HPA) Configuration (hpa.yaml).
    - **test-connection.yaml** – Test file for verifying successful deployment and connectivity, implementation details can be seen in Appendix A.2.5 - Test Connection Configuration (test-connection.yaml).
  - **`helmignore`** – Specifies the patterns from which to exclude files within the Helm chart packaging, described in more detail in Appendix A.2.5 - Helm Ignore File (.helmignore).
- **Server**: Contains the OPC UA server's source code and dependencies. Key files in this directory are:
  - **Dockerfile**: Defines the steps to build the OPC UA server container.

    Outlined in Subsection 4.5.1 - Containerization with Dockerfile in Chapter 4 - Detailed Design, with implementation details in Appendix A.2.5 - Docker Configuration (Dockerfile).
  - **access_registry_docker.txt**: Provides access credentials for Docker registry.

    Detailed in Subsection 4.5.4 - Docker Registry Access in Chapter 4 - Detailed Design, with implementation details in Appendix A.2.5 - Docker Registry Access (access_registry_docker.txt).
  - **BL4R3-rev01-Diagnostic_NodeID_List-scenario.xlsx**: Excel file used for scenario simulations.

    Covered in Section 5.3 - Scenario Simulation and Failure Handling Implementation in Chapter 5 - Implementation), with implementation details in Appendix A.2.3 - Detailed Variable States for Scenario Simulation.
  - XML files such as **eulynx.generic.bl4r3.rev01.xml**, which define the OPC UA Information Model (see Subsection 5.2.6 - Exporting the Generic and Manufacturer Example Namespace as XML in Chapter 5 - Implementation).
  - **server.py**: The main Python script for the OPC UA server implementation.

    Explained in Section 5.4 - OPC UA Server Implementation in Chapter 5 - Implementation), with implementation details in Appendix A.2.4 - Complete OPC UA Server Implementation Code.

– **.gitignore**: Specifies files and directories to be excluded from version control.

  Outlined in Section 4.5.2 - Version Control and Dependency Management in Chapter 4 - Detailed Design, with implementation details in Appendix A.2.5 - Git Ignore File (.gitignore).

This structured layout allows for easy scalability and states that each of the deployment, configuration, and automation are logically laid out. Appendix A.2.5 - Detailed Configuration Files, shows complete configurations for all the YAML and related files.

Pushing the Repository to GitHub

Once repository is structured and all files are ordered, next implementation is to push the repository in GitHub for version control, collaboration and for integration with the CI/CD pipeline.

GitHub makes it possible to push a local repository to GitHub in many ways, command line Git commands, or using a graphical interface. This thesis will be based on using of Git Source Control in Visual Studio Code. This approach takes advantage of VS Code's built-in Source Control feature, allowing the user to work with Git through a user graphical interface.

Using Visual Studio Code Git Source Control

The following steps outline the process of pushing the local repository to GitHub using Visual Studio Code.

- **Open the Project in VS Code:** Open the project folder with a Visual Studio Code launch.
- **Initialize Git:** If Git isn't already initialized, click on "Initialize Repository" in the Source Control tab (it's usually depicted as a branch icon on the sidebar).
- **Stage Changes:** Click the "+" icon under the Source Control panel next to each file to stage changes, or use the "+" icon next to the "Changes" label to stage all files.
- **Commit Changes:** After staging, type a commit message in the input box at the top of the Source Control panel and click the checkmark icon to the right to commit.
- **Add Remote Repository:** Open up the command palette (Cmd+Shift+P for Mac, Ctrl+Shift+P for PC) and type 'Git: add remote.' Then press enter on it, and add in the GitHub repository URL 'https://github.com/loukjeab/EULYNX_SDI_OPC_UA_BL4R3'.
- **Push to GitHub:** Click on the Icon for "... (More Actions)" under the Source Control panel and select "Push." Doing this will take those changes and push them into the GitHub repository.

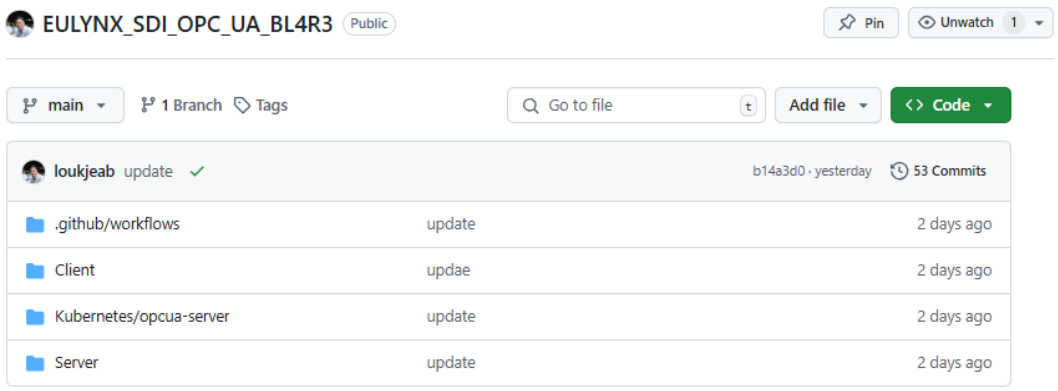


Figure 5.7.: Final GitHub Repository Structure with organized folders and commits, own illustration

As illustrated in Figure 5.7, it is possible to use the Git Source Control feature inside VS Code for an easier Git operation management. After that, all project files in the repository are pushed securely to GitHub, and can be integrated with the CI/CD pipeline or used for collaborative development.

5.6. CI/CD Pipeline Validation

Once the repository is pushed to GitHub the GitHub Actions configured CI/CD pipeline is triggered. This pipeline, as described in Detailed Design Chapter Subsection 4.5.7 - CI/CD Pipeline Integration in Chapter 4 - Detailed Design and Appendix A.2.5 - CI/CD Pipeline Configuration (ci-cd-pipeline.yml), uses kind (Kubernetes in Docker) to validate the build and containerization of the OPC UA server and how it is deployed in a local testing environment. The pipeline includes three main jobs:

- **Build**: Installs dependencies, performs initial checks and verifies the code is compatible with required libraries, only to verify the OPC UA server code, before containerizing.
- **Docker**: Builds a Docker container image of the OPC UA server, tags and pushes the image to the GitHub Container Registry (GHCR) where it is available and stored for deployment.
- **Deploy**: Uses the 'kind' to set up a local K8s environment, deploys the Docker image using Helm and validates the configuration in a sandboxed environment, ensuring the server works as expected in K8s.

Monitoring the CI/CD Pipeline

Each CI/CD Pipeline workflow status is observable in GitHub Actions Interface (accessed under the repository's 'Actions' tab). Figure 5.8 shows the view, includes both successful and failed runs, which facilitates troubleshooting and provides for iteration to successive improvements until a successful run is realized.

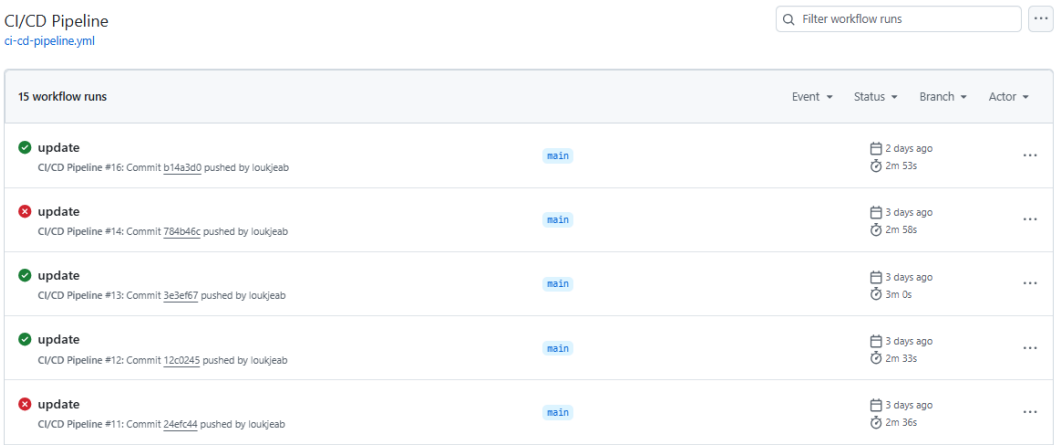


Figure 5.8.: Overview of CI/CD Pipeline Runs with Successful and Failed Executions, own illustration

After successful execution of each job shows a green check mark to confirm that the CI/CD process works fine. If there's a failure in GitHub Actions, it offers error logs for each job step in troubleshooting. An example of a fully successful pipeline run with all checks passing a shown in Figure 5.9.

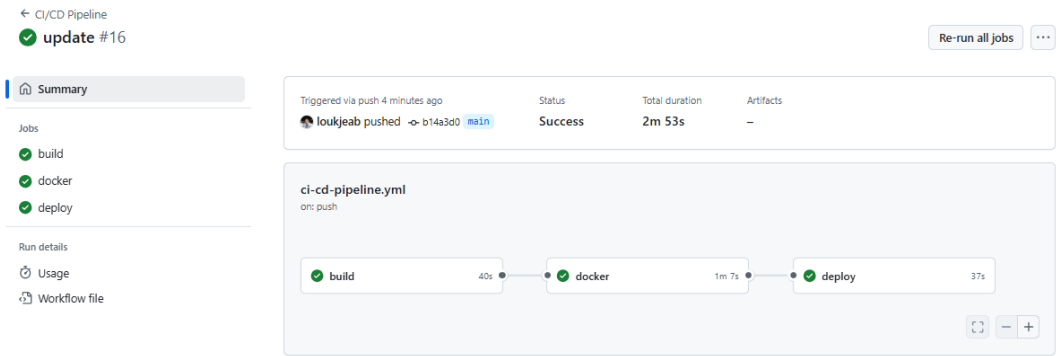
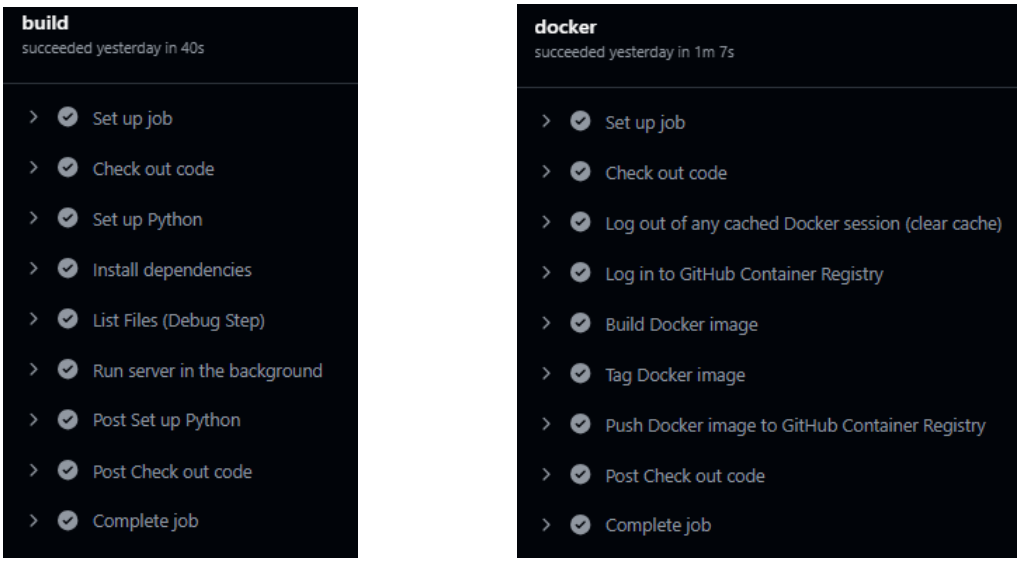


Figure 5.9.: Successful CI/CD Pipeline Execution with All Checks Passed, own illustration

The main jobs break down into detailed steps below for a total validation process:

- **Build Job**: This is a build job, with steps to set up the environment, check out the code, install dependencies, and running the server in the background, to test. Figure 5.10a describes the sequence of what gets executed in this job.
- **Docker Job**: The Docker job logs out of any existing Docker session, logs into the GitHub Container Registry, builds the Docker image, tags it, and finally pushes it to GHCR. Figure 5.10b shows this sequence.



(a) Sequential Steps in the Build Job

(b) Sequential Steps in the Docker Job

Figure 5.10.: Overview of Steps in Build and Docker Jobs, own illustration

- **Deploy Job**: The deploy job defines a ‘kind‘ cluster, sets up kubectl context, add required helm repositories, deploys the Docker image using Helm and finally checks the deployment status. The sequence of steps in this job is shown in Figure 5.11.

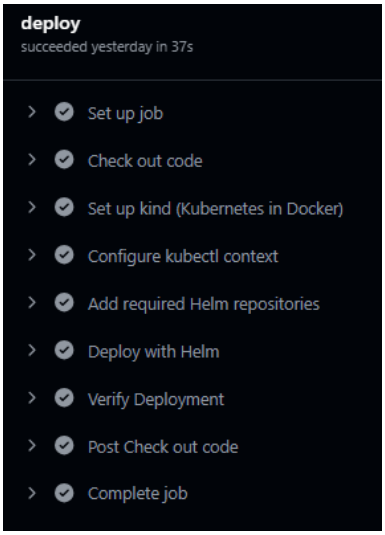


Figure 5.11.: Detailed Steps in the Deploy Job, own illustration

In summary, all CI/CD jobs guarantee OPC UA server reliability, scalability and deployment. Code is first built by verify, Docker prepares the image and Deploy finally deploys code on Kubernetes.

Limitations of the ‘kind‘ Environment

The workflow for CI/CD in GitHub Actions using ‘kind‘ (Kubernetes in Docker) provides a temporary, a sandboxed environment primarily for ensuring that our deployment pipeline works correctly. Below item being able to done by this setup:

- **Test Deployment Pipeline:** Confirm Docker images, K8s, and Helm charts configs are applied correctly.
- **Catch Issues Early:** Avoid any potential errors before going to produce deployment.
- **Run Automated Tests:** Run unit tests to ensure the application fulfils purpose.

Since the environment gets reset with CI/CD run, it doesn't share or sync with local or even production setups and is thus perfect for isolated, disposable testing. For a continuous access or production deployment though, the Docker image is decided to be manually deploy onto a separate, persistent Kubernetes cluster, which will be discussed in the following containerization and deployment sections.

5.7. Containerization with Docker

Containerization was chosen to allow the deployment of the OPC UA server in a Kubernetes environment via its features of portability and scalability. The Docker image is built, tagged and pushed to a GitHub repository using Docker for the ability to control the version and purpose.

Building the Docker Image

A preconfigured Dockerfile within the project's Server directory was used to build the Docker image for the OPC UA Server. This directory contains source files needed for the OPC UA server as described in Implementation Section 5.5 - Repository Structure and Setup Configuration. With the Server directory, the build process was initiated by the following command.

```
docker build -t opcua-server .
```

This command made Docker to construct an image with the instructions contained in the Dockerfile. It installed gcc, libffi-dev and libssl-dev as essential dependencies, and then pulled proper base images. After that, the resulting image was tagged as opcua-server.

Tagging the Docker Image

In order to perform version control, a version tag was assigned to the Docker image using the following command:

```
docker tag opcua-server ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v1
```

The following applied the v1 tag to the image and also specified the GitHub Container Registry (GHCR) location so that future deployments will fetch this version if needed.

Figure 5.12 demonstrates that Docker image was created properly and tagged. opcua-server image is shown with tag latest and ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v1 with tag v1. This verifies that the image and its version tag are configured correctly in order to be pushed to GitHub Container Registry.

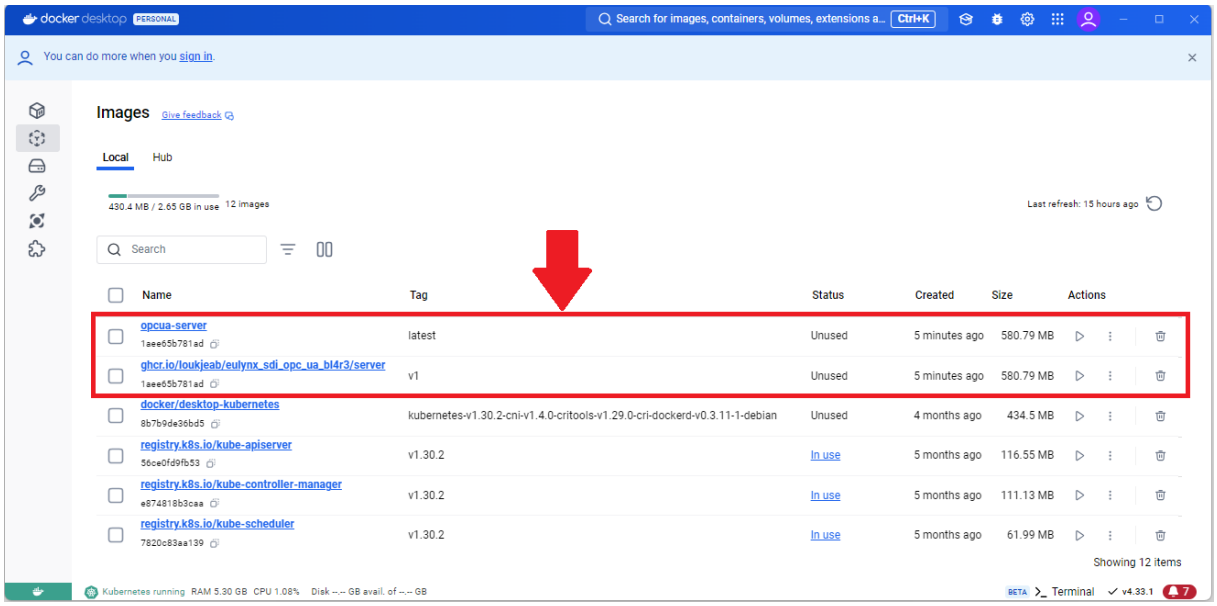


Figure 5.12.: Docker Desktop interface showing OPC UA server images, own illustration

Login and Permissions for Pushing the Image

A login using a Personal Access Token (PAT) will be required to push in case of permission issues when pushing the image to GHCR. Below is the command for authentication:

```
docker login ghcr.io -u GITHUB_USERNAME -p YOUR_NEW_TOKEN
```

It uses the GitHub username and token to authenticate and authorize push the image to the GHCR.

Pushing the Docker Image to GitHub Packages

Once authentication is successful the next step is to push the Docker image, tagged with the name mentioned above, to the GitHub Container Registry using the following command:

```
docker push ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v1
```

This command uploads the image into the GitHub Container Registry so that Kubernetes is able to deploy it. Once successfully pushed, the Docker image would be listed under the repository in the GHCR, accessible in the Packages tab of the GitHub repository, as shown in the Figure 5.13.

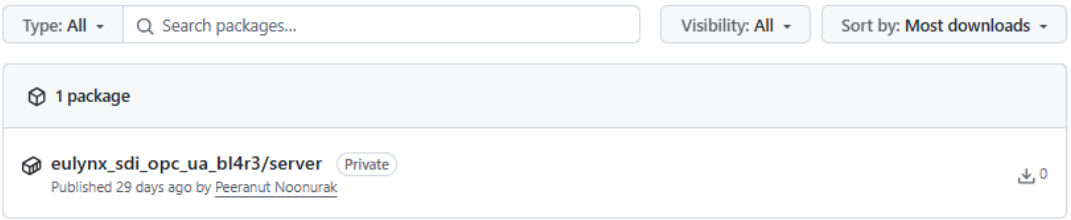


Figure 5.13.: Uploaded Docker image in GHCR Packages tab, own illustration

With this, the steps to build, tag, and push the Docker image to GHCR is completed and now the Docker image is ready to deploy in Kubernetes environment. In the next section this prepared Docker image will be deployed inside Kubernetes cluster using Helm for efficient orchestration and management of multiple OPC UA server instances.

5.8. Deploying the OPC UA Server in Kubernetes using Helm

Having built and pushed the Docker image, the step is then to deploy the image into the Kubernetes cluster using Helm. Kubernetes management is simplified by Helm, since it takes care of configurations and updates.

Helm Installation and Deployment

To deploy the OPC UA server using Helm, navigate to the directory containing the Helm chart configuration. The following command was executed to install or upgrade the Helm release:

```
helm upgrade --install opcua-server .
```

This commands uses the current directory's Helm chart to deploy the OPC UA server. Helm will install the release opcua-server if it doesn't already exist.

Observing the Helm Deployment Status

Once the Helm deployment is done, status information can be observed for the deployment. For this follow is an example output:

```
Release "opcua-server" does not exist. Installing it now.
NAME: opcua-server
LAST DEPLOYED: Wed Oct 30 14:28:36 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
   export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].
       nodePort}" services opcua-server)
   export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.
       addresses[0].address}")
   echo http://$NODE_IP:$NODE_PORT
```

The Helm release name opcua-server has been successfully deployed in the default namespace and the deployment status is deployed.

Observing the Deployed Pod and Containers

To confirm the running pod, which is having multiple containers, can be done with the following command:

```
kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
opcua-servers-578bb49d9c-xlf58    6/6     Running   0          7s
```

Above output ensures a single pod with six containers that are all in a Running state has been deployed. Consequently, to make sure verify the service configuration it can be done by using this follow command.

```
kubectl get svc
```

The output below confirms that the OPC UA server service is exposed via a NodePort port that it can be accessed from the outside.

```
NAME                  TYPE       CLUSTER-IP      PORT(S)                      AGE
kubernetes            ClusterIP  10.96.0.1       443/TCP                      7d13h
opcua-servers-service NodePort   10.105.176.107  4840:30040-4845:30045/TCP    12s
```

The above implemented confirms that the OPC UA server is running and accessible in the Kubernetes environment but can be accessed only by the machine which is hosting the cluster. However, since we are using Docker Desktop to setup the environment, external access is not possible. To allow access from outside of local environment, the Kubernetes cluster has to be hosted on a cloud provider with external IP allocation, or with a LoadBalancer.

Running OPC UA Server Containers in Docker Desktop

Once the OPC UA server is deployed, the deployed containers can be shown within the Docker Desktop as depicted in Figure 5.14. Each one of these containers is an instance of the OPC UA server which, all together, are running inside a single pod.

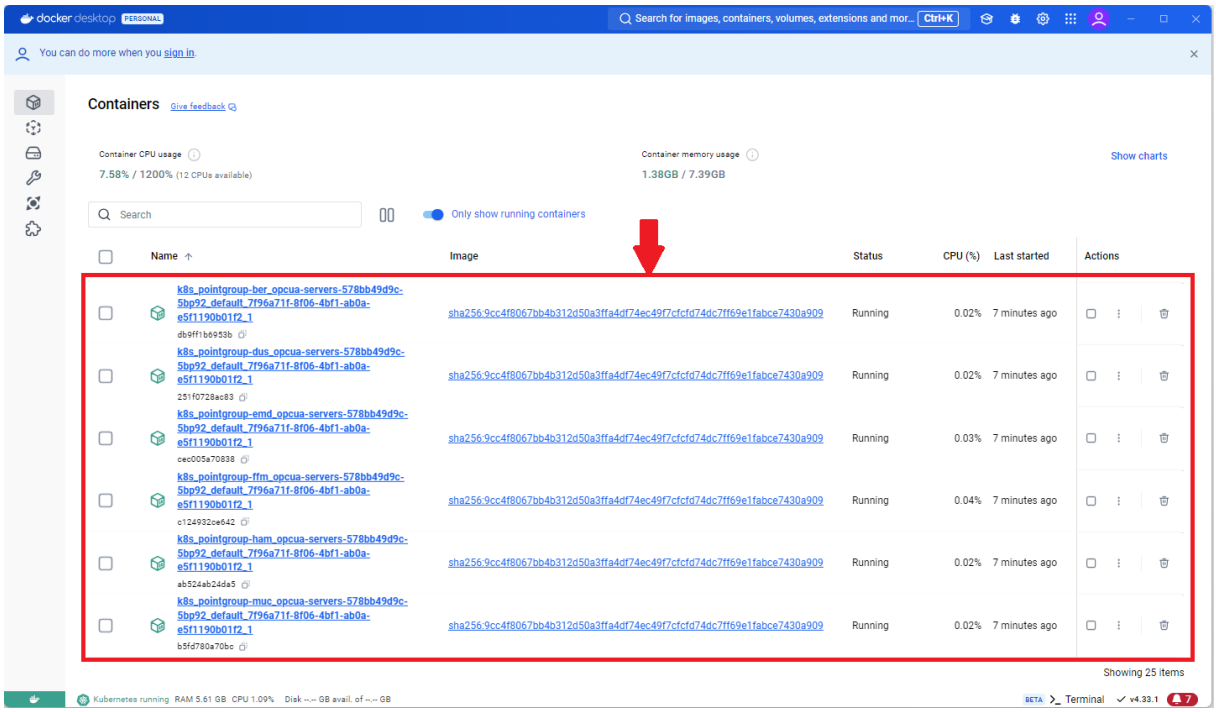


Figure 5.14.: OPC UA server containers in a single Kubernetes pod on Docker Desktop, own illustration

Figure 5.14 above shows that multiple OPC UA server containers are running successfully inside Docker Desktop in a single Kubernetes pod, which provides scalability and resource optimization in the system.

The chapter following the Implementation chapter details the Test & Validation chapter. That phase will involve the integration, tests and verification of each and every component. It will also cover system verification and validation concerning the demonstration that the OPC UA based diagnostics system is fully compliant with all required specifications and works reliably in its intended operational environment.

6. Test & Validation

In this chapter, the validation and verification processes performed to confirm that the standard diagnostics system based on OPC UA meets the specified requirements are described. Every test focuses on functional core, interoperability, and performance to verify the system is ready for introduction as an example of reliable operation to convince other railway stakeholder to adopt this approach.

6.1. Integration, Test, and Verification

Building from the detail in Chapter 5 - Implementation, the Project Management specifications drive the selection of the key test items. In this section, tests are presented to verify the functionality, interoperability, and reliability of diagnostic system employing OPC UA. The results documented will assist Section 6.2 - Requirement Validation, to determine if the project requirements have been fulfilled.

The following Table 6.1 correlates the test items with the corresponding specifications to facilitate full coverage of functional, nonfunctional, technical, and operating requirements.

Test Category	Detailed Test	Matched Specifications
OPC UA Server Connectivity Testing	Accessing OPC UA Server Endpoints	OS-01, FS-02, FS-04, TS-02
	Endpoint Access Test with UaExpert	FS-05, FS-02
	Response Time Measurement Using Log Data	NFS-01, FS-01
	Data Security Testing	NFS-02, OS-02
	Access Restrictions Validation	FS-04, NFS-03, NFS-04, TS-03
	OPC UA Connectivity & Security Testing Summary	FS-01, FS-02, FS-05, NFS-01, NFS-02, OS-01
OPC UA Information Model Verification	ObjectTypes Verification	TS-01, TS-02, FS-03, FS-05
	Enumeration DataTypes Verification	TS-01, FS-03, FS-05
	EventTypes Verification	TS-01, FS-03, FS-05
	Object Mapping Verification	TS-01, FS-03, FS-05, NFS-05
	Model Compliance Summary	TS-01, FS-03, NFS-05
Object Instantiation and Scenario Testing	Object Instantiation in Scenario Plans Verification	TS-02, TS-04, FS-01
	Scenario Simulation and Verification	FS-01, FS-02, FS-03, NFS-03
	Integration with MDM Systems	FS-02, OS-01
	Object Instantiation & Scenario Testing Summary	TS-02, FS-01, FS-02, FS-03, OS-01
Performance and Resource Utilization	Response Time and Latency Measurement	NFS-01, FS-01
	Resource Utilization and Efficiency	NFS-01, FS-04, NFS-03
	CI/CD Pipeline Execution Verification	TS-06
	Kubernetes and Helm Implementation Verification	TS-06, OS-01, NFS-01
	Performance, Response Time & Latency Summary	NFS-01, FS-01, NFS-03

Table 6.1.: Mapping of Test Items to Specifications

6.1.1. OPC UA Server Connectivity Testing

Tests of endpoint accessibility, acceptable response times, and proper enforcement of access restrictions are covered by OPC UA server connectivity testing. This includes connectivity checks, as well as response time measures using log data, UaExpert endpoint validation, limitation in access, and data security verification.

Accessing OPC UA Server Endpoints

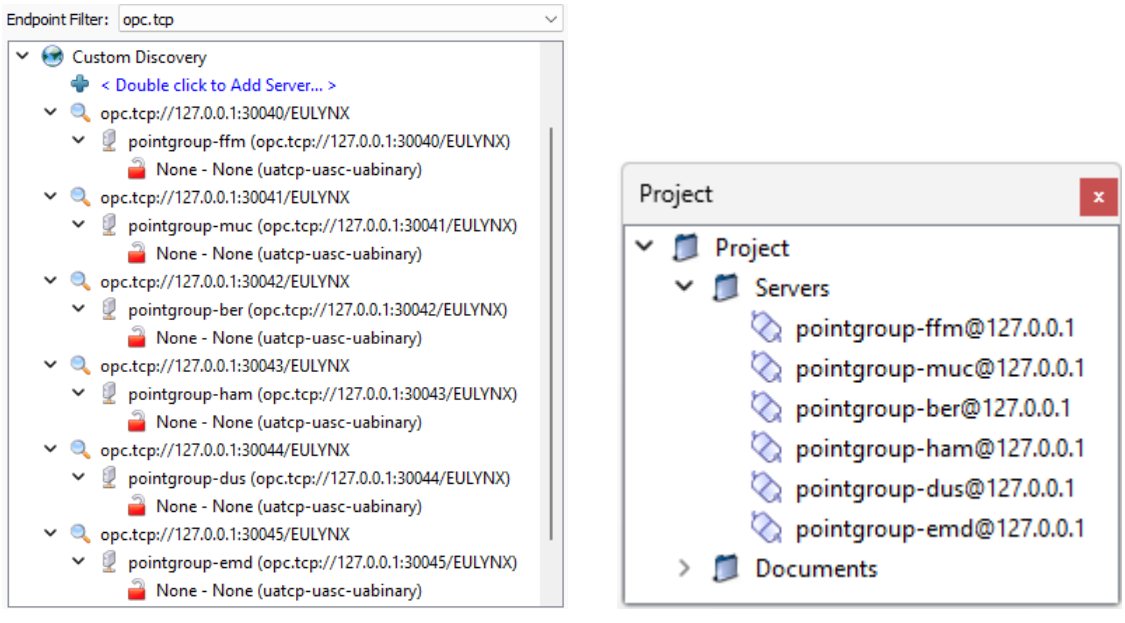
Section 5.8 - Observing the Deployed Pod and Containers in Chapter 5 - Implementation describes the OPC UA server (accessible locally through a Kubernetes NodePort service) used for testing. In this setup, Kubernetes node ports are mapped to server internal ports allowing access within the host machine. Due to the Docker Desktop environment, we cannot get external access because we need either cloud hosting for an external IP or a LoadBalancer service to expose the server to the internet.

The endpoint URIs for each OPC UA server instance are defined in the Helm values.yaml file by specifying the /EULYNX suffix for each endpoint URL (see Appendix A.2.5 - Values File). A client connection to an EULYNX diagnostic interface dictates this suffix, so all client connections know which data model and structure to work with. As such, these URLs are local URLs map to the OPC UA server on NodePort:

```
opc.tcp://127.0.0.1:30040/EULYNX
opc.tcp://127.0.0.1:30041/EULYNX
opc.tcp://127.0.0.1:30042/EULYNX
opc.tcp://127.0.0.1:30043/EULYNX
opc.tcp://127.0.0.1:30044/EULYNX
opc.tcp://127.0.0.1:30045/EULYNX
```

Endpoint Access Test with UaExpert

For endpoint confirmation and configuration testing, OPC UA client (UaExpert), was used with anonymous authentication. The graphical interface of UaExpert helps to connect to OPC UA servers, validate server endpoints connection. Figure 6.1 shows the UaExpert configuration and project windows with each server endpoint correctly recognized and accessible.



(a) Configuration for OPC UA Server Endpoint Access Test (b) Project Window Showing OPC UA Server Endpoints

Figure 6.1.: UaExpert: Endpoint Access Test, own illustration

Response Time Measurement Using Log Data

Log data, captures timestamps on both connection initiation and connectivity status, was used to calculate the response time of each endpoint. With this approach it is possible to accurately measure how long each connection takes to establish. Figure 6.2 shows a pointgroup-emd log entry with derivation of response time.

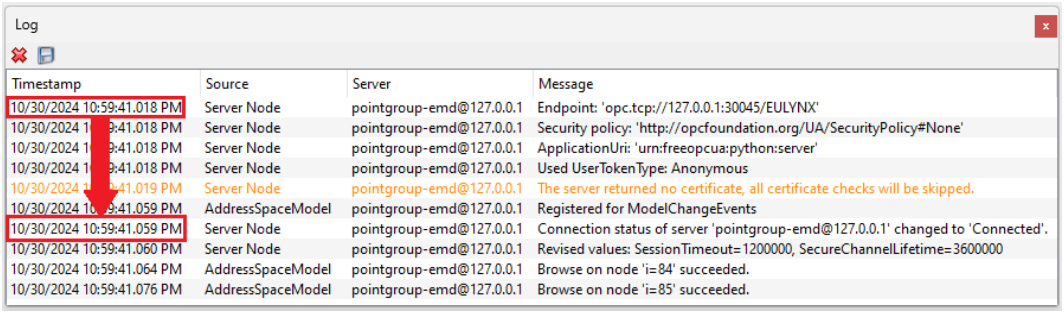


Figure 6.2.: Example Response Time Log (pointgroup-emd), own illustration

Figure 6.2 shows response time calculation for pointgroup-emd, being the time difference between 10:59:41.018 PM connection initiator contacted and finally reaches 10:59:41.059 PM in successful connection status which is 41 milliseconds.

Data Security Testing

OPC UA server connectivity to test for secure data handling validation was performed using UaExpert. Figure 6.3 shows that since authentication using username-password or certificate was not configured, only anonymous authentication was used. Hence, all other endpoints didn't process any encryption or security.

Figure 6.3.: UaExpert Configuration for Data Security Testing, own illustration

Log Figure 6.4 confims this configuration as messages such as "The server returned no certificate, all certificate checks will be skipped" and "Security policy: http://opcfoundation.org/UA/SecurityPolicyNone" are shown as "Used UserTokenType: Anonymous". Which confirms that encryption nor user authentication was applied in the current implementation.

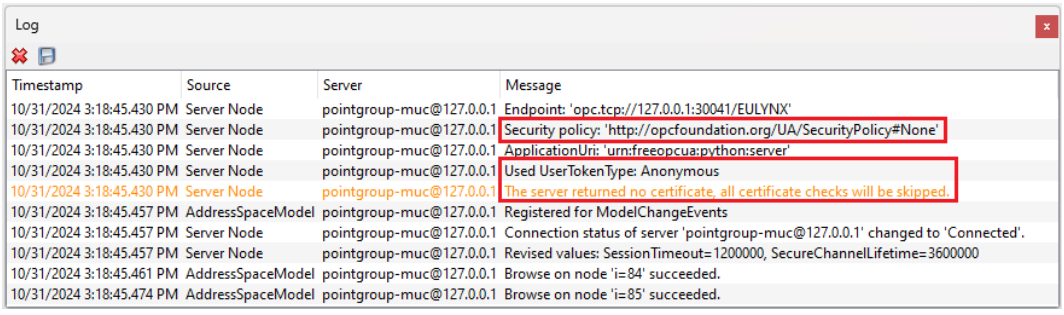


Figure 6.4.: Example Response Lack of Security Log (pointgroup-muc), own illustration

The test results show that some security requirements were not met:

- **Username-Password Authentication**: No username-password authentication was configured and information entered in channel could be accessed by clients anonymously, potentially exposing diagnostic data to unauthorized clients.
- **Message Encryption (AES-256)**: When the security policy is set to "None," then the data is unencrypted, the transmission of data between the client and the server fails to meet the AES-256 encryption.

Access Restrictions Validation

At this time, the OPC UA server can only be accessed within the local machine hosting the Kubernetes cluster because it defaults to being blocked from external access due to the Docker Desktop setup. To enable external connectivity additional configurations would be required on the LoadBalancer or cloud deployment and this is noted as an outlook for potential future work.

OPC UA Server Connectivity and Security Testing Summary

This test ensures all OPC UA server endpoints are accessible, measures response time for each connection, verifies security policies were applied and verifies access restrictions. Table 6.2 summarizes results.

Server	Endpoint	Access Status	Response Time (ms)	Security Policy
pointgroup-ffm	opc.tcp://127.0.0.1:30040/EULYNX	Success	40	No
pointgroup-muc	opc.tcp://127.0.0.1:30041/EULYNX	Success	44	No
pointgroup-ber	opc.tcp://127.0.0.1:30042/EULYNX	Success	47	No
pointgroup-ham	opc.tcp://127.0.0.1:30043/EULYNX	Success	47	No
pointgroup-dus	opc.tcp://127.0.0.1:30044/EULYNX	Success	51	No
pointgroup-emd	opc.tcp://127.0.0.1:30045/EULYNX	Success	49	No

Table 6.2.: Connectivity Test Results for OPC UA Server Endpoints

The response times for each server endpoint ranged from 40 to 51 milliseconds, which verfies that each server endpoint has been successfully accessed and meets to the expected connectivity performance as shown on the Table 6.2. But in the "Security Policy" column all entries have "No" in them, indicating that there were no security policies (encryption and authentication) applied.

Overall, OPC UA server endpoints can be observed with a consistent connectivity and response time within the local Kubernetes environment. Diagnostic data, however, is not created securely and therefore, exposed to unauthorized access. The access is currently limited locally and meets the internal operational requirements but does not allow to access it broadly. In Chapter System Verification and Validation, future improvements to security and broader accessibility shall be addressed.

6.1.2. OPC UA Information Model Mapping Verification

The OPC UA Information Model Mapping Verification validates that all core elements, such as node types, data points, attributes, enumerations and event objects, corresponding to a UML model and conforming to specifications of the EU-Rail and EULYNX Interface Specification SDI (SDI P, SDI-TDS, SDI-LC, etc.). It makes sure that each part of the information model corresponds to the expected configurations set out in the specification, thus guaranteeing uniformity, availability and compatibility within the system.

ObjectType Verification

The verification confirms that each implemented ObjectType conforms to the EU-Rail and EULYNX Interface specifications SDI, all required attributes have been implemented in Namespace 1 (refer to Subsection 5.2.1 - UaModeler Version and Namespace Configuration in Chapter 5 - Implementation). On top of Namespace 0, each ObjectType was modelled following the specification in UaModeler to support its categories such as SDI-Generic, SDI-P, and SDI-IO.

All 111 ObjectTypes for the model is contained in Apendix A.3.1 - List of ObjectTypes, as a verification record. It covers both general types as well as individual components (e.g. Points, Train Detection Systems, Level Crossings) amongst others. Tables A.7 to A.11 of this appendix, document the Name, Abstraction Status, ReferenceType and Target for each ObjectType. ObjectTypes are summarized by SDI specifications, verified as compliant to defined attributes and relationships in the Table 6.3 below:

ObjectType No.	Example Item	SDI-XX Cetagory	Actual Attributes	Verification Status
1 - 57	SubsystemType InterfaceType EquipmentType etc.	SDI-Generic	Matches	Verified
58 - 62	PointType PointMachineType PointTurnEventType etc.	SDI-P	Matches	Verified
63 - 71	GenericIOType LogicalChannelType PhysicalChannelConnectionType etc.	SDI-IO	Matches	Verified
72 - 80	LightSignalType LightPointType etc.	SDI-LS	Matches	Verified
81 - 98	DetectionDeviceType TrainDetectionSystemType TvpSectionType etc.	SDI-TDS	Matches	Verified
99 - 111	LevelCrossingType BarrierType BarrierTurnEventType etc.	SDI-LC	Matches	Verified

Table 6.3.: Verification Summary of ObjectTypes against SDI Specifications

As summarized in Table 6.3, the verification confirms that all implemented ObjectTypes comply with SDI specifications. The appendix provides a complete record of the verified ObjectTypes, serving as a reference for detailed implementation and compliance documentation.

Enumeration DataType Verification

The OPC UA Information Model verification process ascertains that all Enumeration DataTypes are correct and within compliance to EU-Rail and EULYNX Interface Specification SDI. The enumeration data types were validated against the attributes DataType, Range and Enumerated values as per the SDI documentation. Appendix A.2.1 - Example of Enumeration DataType Creation in UaModele, were followed to model them in UaModeler.

86 Enumeration DataTypes were implemented, all relating to Operational States and Conditions within SDI-XX categories like SDI-Generic, SDI-P, SDI-IO, and SDI-LC. Tables A.12 to A.14 are contained in Appendix A.3.1 - List of Enumeration DataTypes, which serves as a verification record. Each of these tables details every enumeration by SDI-XX category.

Enumeration No.	**Example Item**	**SDI-XX Category**	**Actual Attributes**	**Verification Status**
1 - 31	StatusTechnical, ConnectionStatus VoltageStatus etc.	SDI-Generic	Matches	Verified
32 - 39	PointPosition PointTurnFailureReason etc.	SDI-P	Matches	Verified
40 - 46	LogicalInputValue ValenceType LogicalOutputValue etc.	SDI-IO	Matches	Verified
47 - 53	IndicatorStatus LampWireRole etc.	SDI-LS	Matches	Verified
54 - 73	OccupancyStatus SweepingStatus DirectionOfPassing etc.	SDI-TDS	Matches	Verified
74 - 86	BarrierStatus ObstacleDetectorStatus TimeoutStatus etc.	SDI-LC	Matches	Verified

Table 6.4.: Verification Summary of Enumeration DataTypes against SDI Specifications

Thus, the enumerations are summarized in Table 6.4, with example items for each SDI-XX category, and verified compliance with EU-Rail and EULYNX Interface Specifications SDI. This confirms the alignment with the associated attributes, provides standard values for diagnostics and operational monitoring, and meets all data type representation and interoperability requirements.

Event Type Verification

The verification validates that all EventTypes in the OPC UA Information Model correspond to EU-Rail and EULYNX Interface Specification SDI. In total, 11 EventTypes were developed that represent different types of system events, which are based on specification and extend properly from BaseEventType, as described in Subsection 5.2.2 - Structure of the OPC UA Information Model in Chapter 5 - Implementation.

The detailed verification record of all implemented EventTypes is included in Appendix A.3.1 - List of EventType. As presented in Table 6.5, they confirm that the attributes of their interfaces comply with EU-Rail and EULYNX Interface Specification SDI to support the defined system behaviour and their functionality.

Event Type No.	Example Item	SDI-XX Category	Actual Attributes	Verification Status
1 - 4	LogEventType LogEventInterfaceType LogEventSubsystemType etc.	SDI-Generic	Matches	Verified
5	PointTurnEventType	SDI-P	Matches	Verified
6-8	LogEventDetection DeviceLinearTrack CircuitOccupancyType etc.	SDI-IO	Matches	Verified
9 - 11	BarrierTurnEventType LogEventLocalHandoverType etc.	SDI-LC	Matches	Verified

Table 6.5.: Verification Summary of EventTypes against SDI Specifications

Table 6.5 confirms that relevant EventType satisfy SDI specification and supports diagnostics, monitoring, and alerting functions. Coverage of both operational and state change events is guaranteed and the OPC UA Information Model is aligned with SDI-defined behaviors.

Object Mapping Verification

This verification is to check if each ObjectType as well as its child placeholders has been implemented as per EU-Rail and EULYNX Interface Specification SDI with parent–child relationships as per UML Class diagram in the specification. In hierarchical structure placeholders in ObjectTypes are essential for the railway system design.

The example of verofication is summarized in Table 6.6, which shows how object placeholders are instantiated and validated at the level of their parent ObjectTypes, using EU-Rail and EULYNX Interface Specifications SDI. By this alignment, it verifies that the implementation supports interoperability and traceability of the OPC UA Information Model. Compliance with SDI defined structures is shown for example in Controller within EquipmentType and LightPointsSingleLED within LightSignalType.

Placeholder Object	Parent ObjectType	Verification Status
<Controller>	EquipmentType	Verified
<LightPointsSingleLED>	LightSignalType	Verified
<DetectionElement>	LevelCrossingType	Verified
<PhysicalAnalogInput>	EquipmentType	Verified
<ObstacleDetectorRadar>	LevelCrossingProtectionFacilityType	Verified

Table 6.6.: Representative Object Mapping Verification in OPC UA Information Model

Also, the verification of the inner variable structure for the ObjectTypes' key attributes, e.g. DataType, Modeling Rule, and Reference Type for types such as EquipmentType, was conducted. An example of the organization and validation of these variables in the model is shown in Table 6.7. Attributes important to satisfy SDI-Generic and SDI-dependent requirements are rendered by this table.

NodeClass	**Name**	**DataType**	**Modeling Rule**	**Verification Status**
Variable	IsTimeSynchronised	Boolean	Optional	Verified
Variable	HardwareRevision	String	Mandatory	Verified
Variable	Manufacturer	String	Mandatory	Verified
Variable	ManufacturerModel	String	Mandatory	Verified
Variable	ManufacturingDateTime	DateTime	Mandatory	Verified
Variable	ReplacementIndication	Equipment ReplaceabilityStatus (Enumeration)	Mandatory	Verified
Variable	SerialNumber	String	Mandatory	Verified
Variable	SoftwareRevision	String	Mandatory	Verified
Variable	StatusTechnical	Enumeration	Mandatory	Verified
Variable	StatusTechnicalManufacturer SpecificMessage	MultiState DiscreteType Supplier (UInt16)	Optional	Verified
Variable	DefaultInstanceBrowseName	QualifiedName	Mandatory	Verified
Variable	Label	String	Optional	Verified
Placeholder Object	<Controller>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalAnalogInput>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalAnalogOutput>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalDigitalInput>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalDigitalOutput>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalNetworkInterface>	-	OptionalPlaceholder	Verified
Placeholder Object	<PowerSupply>	-	OptionalPlaceholder	Verified
Placeholder Object	<StorageMediumFlashMemory>	-	OptionalPlaceholder	Verified
Placeholder Object	<SubEquipment>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalSeparatedOutput>	-	OptionalPlaceholder	Verified
Placeholder Object	<PhysicalSeparatedInput>	-	OptionalPlaceholder	Verified
Placeholder Object	<InputSwitch>	-	OptionalPlaceholder	Verified

Table 6.7.: Inner Structure Example of 'EquipmentType' Verification in OPC UA Model

Object Mapping Verification confirmed that all of the ObjectTypes and their child nodes conform to the EU-Rail and EULYNX Interface Specification SDI and that the parent-child relationship is maintained, as defined in Table 6.6. Additionally, the inner structure of EquipmentType (Table 6.7) was validated to match with SDI defined attributes. This includes more than 600 variables, thus complete ObjectType details and structures is provided in a supplemental Excel document, which is considered a database for this thesis.

Model Compliance Summary

This verification confirms that Information Objects (including ObjectTypes, Enumeration DataTypes and EventTypes) and object mappings of the OPC UA Information Model comply with specifications. Table 6.3, 6.4 and 6.5 show compliance with defined SDI attributes and requirements respectively.

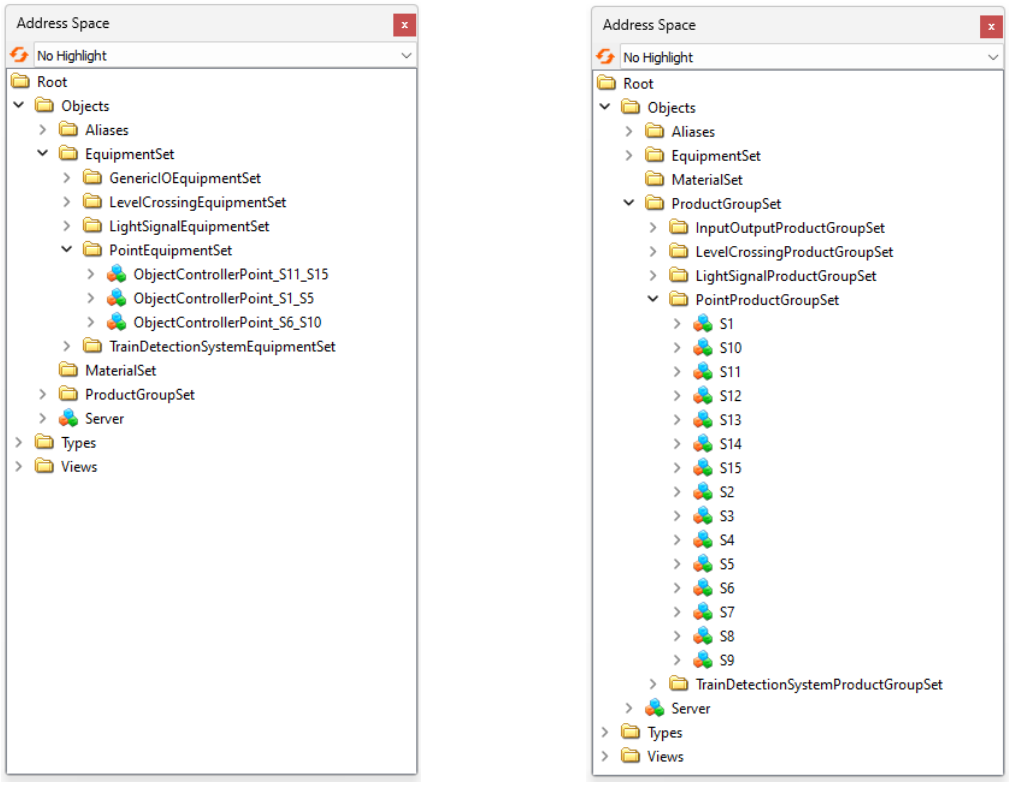
Table 6.6 confirms relationships between parent and child ObjectTypes and Table 6.7 verifies the children of EquipmentType are properly constructed as per SDI attribute definitions for their DataType, Modeling Rule, and Reference Type. A detailed record of over 600 variables and structure definitions is provided in the supplementary Excel document, assuring traceability, interoperability and full alignment to SDI specifications.

6.1.3. Object Instantiation and Scenario Testing

Implementation of object instantiation and scenario simulation has been performed, as described in Implementation Section 5.3 - Scenario Simulation and Failure Handling Implementation. The purpose of this verification subsection is to verify whether the configurations are properly implemented, additionally ensuring that the performance is correct under simulated conditions and evaluate the integration with MDM system.

Object Instantiation in Scenario Plans Verification

This verification guarantees that the instantiated objects in Namespace 2 of the OPC UA (Points and Object Controllers) are compliant with the intended structure described in Implementation Subsection 5.2.4 - Manufacturer Example Model - Namespace 2, and that the instantiated objects are in accordance with the pre-configured Excel file for scenario testing (see Implementation Section 5.3 - Scenario Simulation and Failure Handling Implementation); namely Point S10, and Object Controller Points S6-S10.



(a) PointEquipmentSet

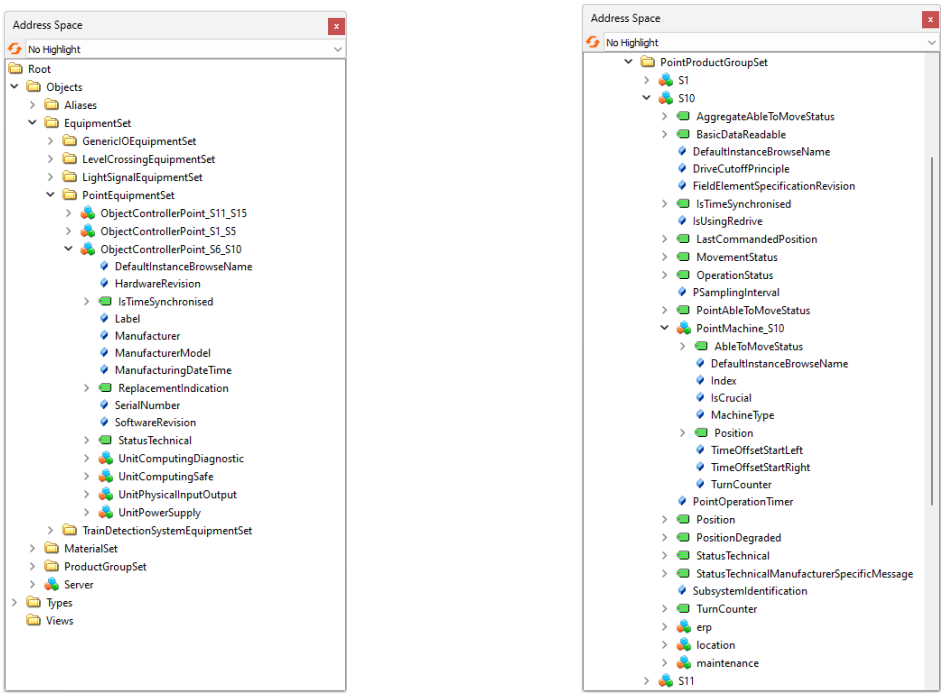
(b) PointProductGroupSet

Figure 6.5.: UaExpert: Address Space Window, own illustration

Figures 6.5a and 6.5b thus serve as visual proof that the instantiated objects (Points and Object Controllers) are now visible in the server's address space, and hence the instantiation was done correctly.

PointEquipmentSet As explained in Chapter 5 - Implementation, Subsection 5.2.4 - Instantiation of Object Controllers Points, each Object Controller is comprised of four main units: These units are shown in Figure 6.6a, which verifies that the attributes and structure concur with the planned instantiation. Appendix A.3.2 - PointEquipmentSet Unit Details, provides details on each unit of the Object Controller Point.

PointProductGroupSet Subsection 5.2.4 - Instantiation of Point (S1–S15) and Subsection 5.2.4 - Instantiation of Point Machine (Point Machine S1 – Point Machine S15) in Chapter 5 - Implementation, outlined that the PointProductGroupSet contains the Points and their Point Machines. Figure 6.6b illustrates that Points and their associated attributes including object are instantiated accurately following scenario plans, according to the structure of PointProductGroupSet.



(a) Object Controller Point S6-S10 (b) Point S10

Figure 6.6.: UaExpert: Overall Point Equipment and Product group, own illustration

The Figure 6.6 above indicate that each unit of EquipmentSet and ProductGroupSet is instantiating correctly, with structures and attributes mapped to the expected scenario and manufacturer specific configurations.

Object	Expected Attributes	Actual Attributes	Verification Status
Point S1 - S15	Position, Status, Detection Circuits, etc.	Matches	Verified
Point Machine S1 - S15	TimeOff, Turn Counter, etc.	Matches	Verified
Object Controller S1-S5	ID, Status, Redundancy Group, Units, etc.	Matches	Verified
Object Controller S6-S10	ID, Status, Redundancy Group, Units, etc.	Matches	Verified
Object Controller S11-S15	ID, Status, Redundancy Group, Units, etc.	Matches	Verified

Table 6.8.: Expanded Object Instantiation Verification with Manufacturer-Specific Attributes

Hence, table 6.8 confirm that all instantiated objects conform to planned configurations (as described in Subsection 5.2.4 - Manufacturer Example Model in Chapter 5 - Implementation) needed for this scenario simulation. Key attributes including Position, Status, Redundancy Group, Operational Status, match the pre configured excel data for which the OPC UA model is prepared for accurate monitoring and diagnostics.

Scenario Simulation and verification

The scenarios were configured in Excel as explained in Section 5.3 - Scenario Simulation and Failure Handling Implementation in Chapter 5 - Implementation, to verify the diagnostic response of the OPC UA server to critical and non-critical failures.

The real time diagnostic messages to the client was monitored by using UaExpert, to make sure that correct response behaviors were observed. Table 6.9 presents a summary of the results where we verified that the diagnostic responses of the OPC UA model matched expected results.

Scenario	**Expected Diagnostic Messages/Status**	**Observed Diagnostic Messages/Status**	**Verification Status**
Critical Failure at Point S10	Point S10 Failure Detected	Matches expected message	Verified
	Position set to UnintendedPosition	Position confirmed as UnintendedPosition	
	Status updated to FailureCritical	Status accurately set to FailureCritical	
Non-Critical Failure in OCP S6-S10	Subsystem Status: Non-Critical Issue	Matches expected message	Verified
	Warnings issued for: - StatusTechnical - CoolingFanStatus - TemperatureStatus	Variables updated as: - StatusTechnical: Warning - CoolingFanStatus: Failure - TemperatureStatus: TooHigh	
	Failure isolated to subsystem without broader system impact	No impact on overall system status	

Table 6.9.: Scenario Simulation Verification Results

The detailed analysis of each scenario is presented below:

- **Scenario 1: Critical Failure at Point S10**

  The critical failure at Point S10 simulated does not reach its position. Data for simulation was configured in the Excel worksheet (see Appendix A.2.3 - Scenario 1: Critical Failure – Point Switch Failure).

  UaExpert's Data Access View (Appendix A.3.2 - Data Access View in Scenario 1) and Event View (Appendix A.3.2 - Event View in Scenario 1) were used to validate the server's response, including failure alerts and status updates. This confirms that variable states could be monitored in real-time and that timestamps synchronized to indicate immediate updates.

- **Scenario 2: Non-Critical Failure in Object Controller**

  In the Object Controller for Points S6-S10, the Unit of Computing Safe was simulated for a non-critical failure with data from respective Excel worksheet as detailed in Appendix A.2.2 - Scenario 2: Non-Critical Failure – Object Controller Failure.

  UaExpert's Data Access View (Appendix A.3.2 - Data Access View in Scenario 2) was used to observe the server's diagnostic response in real time, and it showed that only the effected sub-equipment displayed changes, without impacting the overall system status.

As the outcomes confirm, real time monitoring, logging and simulation are all running as expected, providing system visibility. It effectively separates between critical and non critical failures, which support robust monitoring as well as diagnostics.

Integration Verification with MDM System

A custom OPC UA Client was built to test interoperability, data retrieval and real time monitoring functionalities between the OPC UA server and the Maintenance and Data Management (MDM) system. Sooryanarayanan Garudath implemented this client as part of his parallel thesis project dealing with an MDM solution for railway diagnostics under the EULYNX standard. His work on the OPC UA client can support simlutaneous connections to multiple servers and namespace aggregation, as well as node subscription and dynamic monitoring that is based on the EULYNX diagnostics framework specifications [43].

Integration testing consisted of establishing a connection between the MDM system's OPC UA Client and the OPC UA Server instances created for this thesis. The client, built with asyncua Python library supported asynchronous communication with the servers, and enabled the real time diagnostics data flow with no interruptions even when multiple servers were connected. Operational data, including Point S10's status and diagnostic updates for an Object Controller Point, were retrieved by the MDM system, and proved to show integration conforming with the architecture EULYNX intended for diagnostics.

Below were the Primary focus areas for verification:

- **Connection Stability**: Stable and consistent connectivity of the OPC UA Client with multiple server instances was observed for monitoring multiple Points.
- **Namespace Aggregation and Data Consistency**: Diagnostic data from OPC UA Server was verified to match the expected data structure with MDM system.
- **Dynamic Value Updates**: Accuracy of the updates from OPC UA server for real-time monitoring and diagnostics was confirmed using the Excel configured scenarios.

Test Item	Expected Outcome	Verification Status
Connection Stability	Stable Connection	Verified
Namespace Aggregation	Consistent Data Structure	Verified
Dynamic Updates	Real-time Diagnostics	Verified

Table 6.10.: Integration with MDM System Verification Results

Shown in Table 6.10 above, The OPC UA server was successfully integrated into MDM system's OPC UA Client and all specified connectivity, data accuracy and real-time monitoring goals were achieved. The EULYNX compliant diagnostic framework validated and supported by this collaboration facilitated monitoring, diagnostics and scenario handling in both thesis projects.

Object Instantiation and Scenario Testing Summary

As a result, this subsection confirms that the OPC UA Information Model components are correctly instantiated, scenarios simulated as intended, and are well integrated with external systems. Verification demonstrated that all objects satisfy planned configuration, which allows to structurally monitor Points and Object Controllers.

Validation of the server's real-time diagnostic response was performed via scenario simulation and verified proper logging and failure differentiation for both critical and non critical failures. Also, integration with the MDM system's OPC UA Client demonstrated successful interoperability, reliable data retrieval and real time diagnostics. The result of these show that the EU-Rail and EULYNX Interface Specification SDI has been supported in terms of comprehensive diagnostics, simulation and interoperability through the thesis' implementation.

6.1.4. Performance and Resource Utilization Verification

In this section, OPC UA server performance is verified by measuring response time, latency, CPU and memory usage and stability of CI/CD pipeline. The results of these tests confirm that the system can effectively process real time diagnostics with minimum resource expenditure and also with reliable automated deployment.

CI/CD Pipeline Execution Verification

In this subsection, the execution and performance of the CI/CD pipeline implemented in the GitHub Actions is verified. The pipeline was designed to automate building, testing and deployment of the OPC UA server thus ensuring consistent and reliable deployments. As described in Detailed in Section 5.6 - CI/CD Pipeline Validation in Chapter 5 - Implementation, the pipeline comprises of Build, Docker and Deploy jobs running inside of a sandboxed ‘kind‘ (Kubernetes in Docker). These are tests to validate whether the pipeline functions as intended, without impact to production environment.

The CI/CD pipeline was verified for the following objectives:

- **Pipeline Stability:** Confirming the pipeline runs reliably across multiple runs, each job finish w/o error.
- **Automated Build, Test, and Deployment:** Ensures that the complete pipeline (from build, containerization, to deployment) is working and a Docker image is pushed into the GitHub Container Registry (GHCR) and deployed in kind.
- **Resource Efficiency:** Verified minimal resource consumption, such as cpu and memory, while executing pipeline to ensure scalability and efficiency of automated deployments.

The verification results, as shown in Table 6.11, validate the pipeline being reliable and efficient at each stage.

Pipeline Stage	Execution Status	Resource Impact
Build	Successful	Minimal CPU/memory usage
Docker	Successful	Minimal CPU/memory usage
Deploy	Successful	Minimal CPU/memory usage

Table 6.11.: CI/CD Pipeline Execution Verification Results

The results support that the CI/CD pipeline runs correctly, with all stages ensuring functional accuracy and resource utilization. Through the use of the kind environment, the pipeline provides a safe and temporary place to test out the entire deployment process. Though kind is a non-persistent testing environment, to achieve long term operational stability and reliability, it needs to be manually implemented in a persistent Kubernetes cluster for production level deployment.

Kubernetes and Helm Implementation Verification

This verification, described in more detail in Implementation Section 5.8 - Deploying the OPC UA Server in K8s using Helm, verifies that the OPC UA Server was implemented in K8s using Helm charts successfully. Maintainability is supported by allowing the infrastructure to be updated efficiently via the CI/CD pipeline with the rolling update strategy ensuring continuous, non disruptive maintenance. By using this setup, K8s and Helm simplify initial deployment and ongoing updates for the system in more reliable way.

Response Time and Latency Verification

In the OPC UA Server Endpoint Connectivity Test, refer to Section 6.1.1, response time and latency were measured. Records from Table 6.2 shows that the response time fell between 40 to 51 milliseconds for the server endpoints. The response time here is low which indicates that communication between local kubernetes environment is efficient.

Resource Utilization and Efficiency Verification

Docker Desktop was utilized to monitor the resource utilization of the OPC UA server containers focusing on average CPU and memory usage over time. The trend of resource usage (CPU and memory) in Figure 6.7 shows metrics for OPC UA server containers.

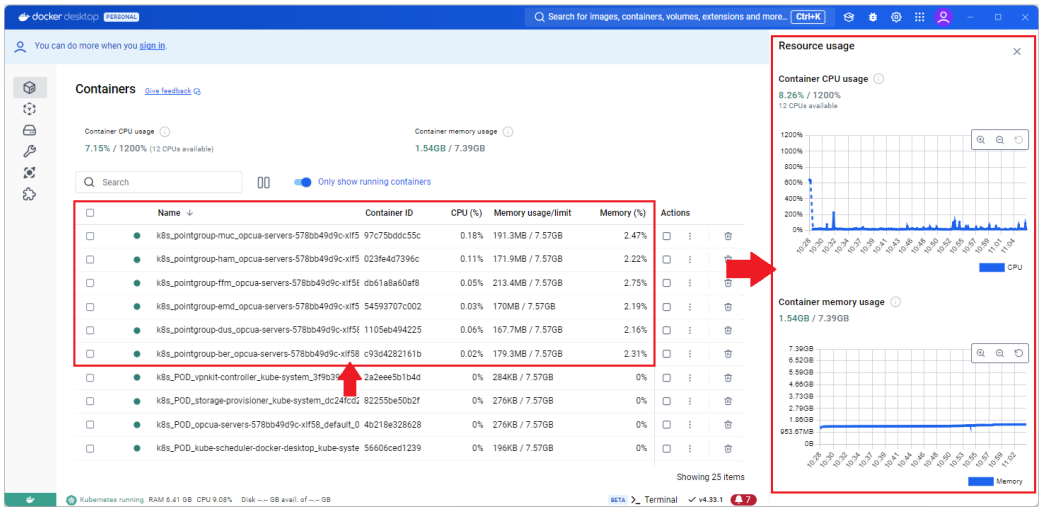


Figure 6.7.: Resource Trends and Docker Metrics, highlighting OPC UA server containers, own illustration

As can be seen in the Figure 6.7, CPU usage across all containers is kept low, while memory usage stays well below the total system capacity of 7.57 GB. This implies that this system utilizes an efficient resource utilization without causing undue stress to the host environment. A summary of mentioned item for each container, with minimal impact on system resource, is shown in Table 6.12.

Container	CPU Usage (%)	Memory Usage (MiB)	Memory Usage (%)
pointgroup-muc	0.18%	191.3 MB	2.47%
pointgroup-ham	0.11%	171.9 MB	2.22%
pointgroup-ffm	0.05%	213.4 MB	2.75%
pointgroup-emd	0.05%	170 MB	2.19%
pointgroup-dus	0.06%	167.7 MB	2.16%
pointgroup-ber	0.02%	179.3 MB	2.33%

Table 6.12.: Resource Utilization Results for OPC UA Server Containers

Table 6.12 verifies that each OPC UA server container is low on CPU as well as memory usage maintaining the high operational efficiency without causing intense usage of system resource.

Performance Summary

In conclusion, the performance tests proved that the OPC UA server based diagnostics system provides stable, low latency communication with response times varying between 40 – 51 milliseconds per endpoint allowing for real time diagnostics.

Resource monitoring shown low CPU and memory consumption (from 2 to 3% of system capacity), which proves the efficient use of resources and the scalability for the systems. Also, the CI/CD pipeline delivered consistent and dependable execution, automation of build, test, and deployment processes in resource efficient ways in the sandboxed environment. These results together suggest that both the system and its deployment processes are ready for responsive, scalable operations with minimal latency and sufficient automation.

6.1.5. System Workflow Overview

Figure 6.8 gives an overview of a system which combines modeling, implementation and verification phases. UaModeler is used to map UML based diagnostic models to an OPC UA Information Model, and Python and various libraries are used to implement server. After that, the implementation is containerized with Docker, then deployed on Kubernetes using Helm charts in order to be able to scale and be reliable.

A CI/CD pipeline has been designed to have the OPC UA servers built, tested and deployed automatically using GitHub Actions. Additionally, scenarios are imported from Excel to drive the diagnostic condition simulation, while UaExpert also validates server endpoints to provide real time diagnostics. The interoperability across multiple servers is further tested with a custom Maintenance and Data Management (MDM) system.

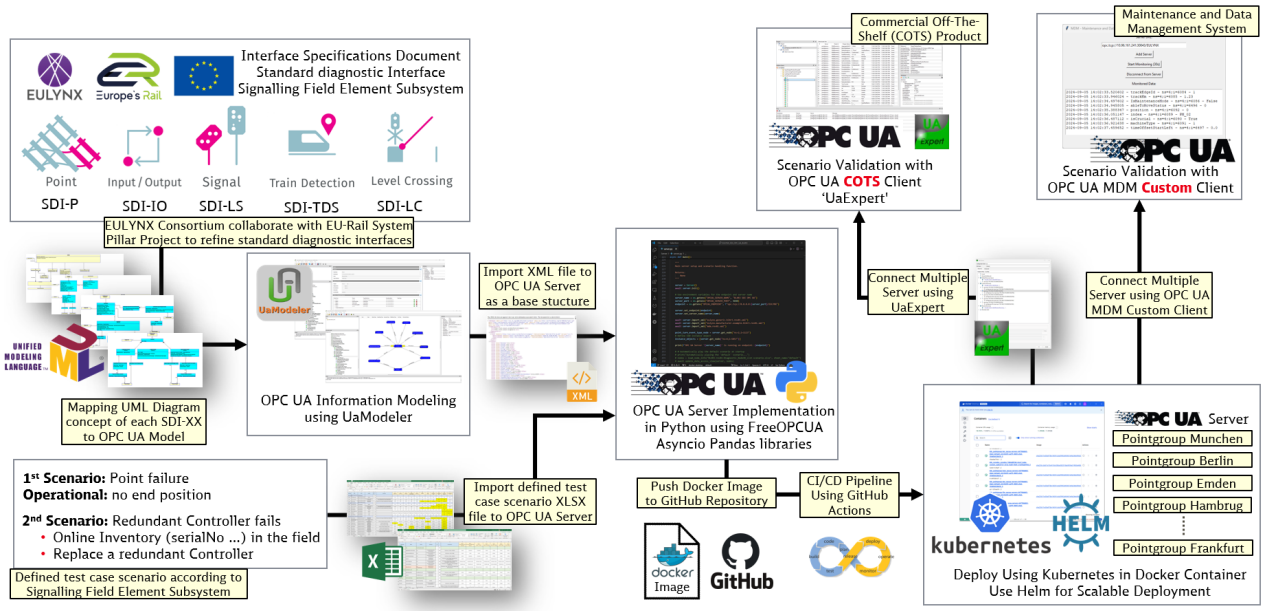


Figure 6.8.: System Workflow Overview, from modeling to verification, own illustration.

This represents a standardized and efficient approach to achieving the diagnostics, aligned with EU-Rail and EULYNX Interface Specification SDI, and further automates the process to make that scalable and maintainable way to deploy the system.

6.2. Requirement Validation

This section evaluates the OPC UA based EU-Rail and EULYNX Interface Specification Standard Diagnostics Interface (SDI) for the Command and Control Signaling (CCS) system in terms of meeting the requirements in Section 3.1 - Requirements of Chapter 3 - Project Management. Validation of each requirement category is carried out through specific test items, with clear mapping between a test to its requirement to ensure that the project requirement is in line with the specifications.

6.2.1. Functional Requirements (FR) Validation

1. **FR-001-01: Real-time Data Collection from Field Elements**
   - **Test Item:** Response Time Measurement Using Log Data (Subsection 6.1.1).
   - **Result:** Met - Real-time diagnostic data collected from the field elements are confirmed through response time logging, fulfilling the requirement for timely data collection.

2. **FR-001-02: Data Access for Maintenance**
   - **Test Item:** Endpoint Access Test with UaExpert (Subsection 6.1.1).
   - **Result:** Met - Successful endpoint access testing with UaExpert confirmed that Maintenance and Data Management system have access to the Data.
3. **FR-002-01: System Interoperability**
   - **Test Item:** Integration Verification with MDM System (Subsection 6.1.3).
   - **Result:** Met - The interoperability is confirmed by MDM integration, adherence to specification SDI and compatibility with external systems.
4. **FR-003-01: Flexible and Scalable System**
   - **Test Item:** Resource Utilization and Efficiency Verification (Subsection 6.1.4).
   - **Result:** Met - The system scales well with more field elements and is supported by flexibility, with its effective utilization of resources in testing.
5. **FR-004-01: User Interface for Data Visualization**
   - **Test Item:** Endpoint Access Test with UaExpert (Subsection 6.1.1).
   - **Result:** Partially Met - UaExpert provides data visualization, but does not include a dedicated HMI meeting part of the visualization requirements.
6. **FR-005-01: OPC UA Information Model Compliance**
   - **Test Item:** OPC UA Information Model Mapping Verification (Subsection 6.1.2).
   - **Result:** Met - Field elements are accurately represented in the OPC UA Information Model and are verified with model mapping, which conforms to standards.

6.2.2. Non-Functional Requirements (NFR) Validation

1. **NFR-001-01: System Performance**
   - **Test Item:** Performance and Resource Utilization Verification (Subsection 6.1.4).
   - **Result:** Met - The scalability and the reliability of automated deployment are validated through response times (40-51 ms) and efficient allocation of resources in testing.
2. **NFR-002-01: Data Security**
   - **Test Item:** Data Security Testing (Subsection 6.1.1).
   - **Result:** Partially Met - Data was exposed only to the anonymous access. Data access needs to be fully secured with additional authentication and encryption.
3. **NFR-003-01: High Availability and Fault Tolerance**
   - **Test Item:** Kubernetes and Helm Implementation Verification (Subsection 6.1.4).
   - **Result:** Met - The pod failure is managed by Kubernetes and redundancy is maintained, CI/CD provides reliable updates. Verification also demonstrates the infrastructure ready for continued operation.
4. **NFR-004-01: Maintainability**
   - **Test Item:** Kubernetes and Helm Implementation Verification (Subsection 6.1.4).
   - **Result:** Met - K8s and Helm support maintainability and updates are achieved with CI/CD. The system's maintainability meets requirements and is confirmed through efficient server management.
5. **NFR-005-01: Compliance with Industry Standards**
   - **Test Item:** Model Compliance Summary (Subsection 6.1.2).
   - **Result:** Met - Verification of compliance to EU-Rail and EULYNX Interface Specification SDI proved ObjectTypes, enumerations, and events mapping accurate, fully matching industry standard.

6.2.3. Technical Requirements (TR) Validation

1. **TR-001-01: Implementation of OPC UA Server for at least One Subsystem**
   - **Test Item:** Object Instantiation in Scenario Plans Verification (Subsection 6.1.3).
   - **Result:** Met - Implementation and validation of the OPC UA server with instantiated objects including Points and Object Controllers aligned to the designed structure for Namespace 2.
2. **TR-002-01: Containerized Deployment of OPC UA Servers**
   - **Test Item:** CI/CD Pipeline Execution Verification (Subsection 6.1.4)
   - **Result:** Met - Docker stage validates containerized OPC UA server deployment via CI/CD pipeline, ensuring orchestration and scalability in K8s, besides meeting the deployment requirement.
3. **TR-003-01: Dynamic Simulation of Field Element Changes**
   - **Test Item:** Scenario Simulation and Verification (Subsection 6.1.3).
   - **Result:** Met - Implemented and validated by scenario testing the dynamic simulation. It showed correct diagnostic responses to both critical and non critical field element changes.
4. **TR-004-01: Real-Time Data Synchronization**
   - **Test Item:** Integration Verification with MDM System (Subsection 6.1.3).
   - **Result:** Met - Real-time update integration testing between the OPC UA server and MDM system showed the real-time updates between them are reliable and accurate.
5. **TR-005-01: UML Mapping to OPC UA Information Model**
   - **Test Item:** OPC UA Information Model Mapping Verification (Subsection 6.1.2).
   - **Result:** Met - Verification showed that all core elements were mapped according to UML mappings with consistent ObjectType, enumeration, and event structures in accordance with SDI standards.
6. **TR-006-01: CI/CD Pipeline Integration**
   - **Test Item:** CI/CD Pipeline Execution Verification (Subsection 6.1.4).
   - **Result:** Met - GitHub Actions CI/CD pipeline, automate build, test then deploy into a sandboxed kind environment. Support for seamless deployment, efficient resource usage and reliable automation are provided.

6.2.4. Operational Requirements (OR) Validation

1. **OR-001-01: Simulated Deployment Environment**
   - **Test Item:** Kubernetes and Helm Implementation Verification (Section 6.1.4).
   - **Result:** Met - Deploying an OPC UA server in Kubernetes with Helm proved operational readiness, while updating from CI/CD provided ease of use and dependedably update deployed items in a stable environment, matching the simulated operational requirements.
2. **OR-002-01: Monitoring and Logging System**
   - **Test Item:** Resource Utilization and Efficiency (Section 6.1.4).
   - **Result:** Partially Met - Docker Desktop confirms that logging is available, but continuous monitoring is limited. Prometheus and Grafana integration would improve real time monitoring, making operational requirements complete.
3. **OR-003-01: Backup and Disaster Recovery Plan**
   - **Test Item:** N/A
   - **Result:** Unmet - Because there are no cloud services, backup and disaster recovery were not implemented, and this is an area for future improvement if the external storage is intergrated.

4. **OR-004-01: Regular System Maintenance**
    - **Test Item:** Kubernetes and Helm Implementation Verification (Section 6.1.4).
    - **Result:** Met - Kubernetes and Helm support rolling updates, allowing continuous, non-disruptive maintenance, and ensures that system updates are rolled in with no downtime.

6.2.5. Summary of Validation Results

Verification and validation have confirmed that the implementation has fulfilled almost all the requirements. Summary of verification results for each of the requirements is given in following Table 6.13.

Mapping Between Requirements, Specifications, and Validation Status

Requirement	Specification	Validation
GRQ-001: Cross-layer collaboration	FS-01, FS-03, NFS-02, TS-01	Met
GRQ-002: Real-time Performance	FS-01, FS-05, NFS-01, TS-05	Met
GRQ-003: Compliance with Industry Standards	FS-03, NFS-05, TS-02	Met
GRQ-004: Flexibility and Scalability	FS-04, NFR-004-01, TS-03	Met
GRQ-005: System Monitoring and Maintenance	FS-02, TS-04, TS-06, OS-02, OS-04	Partially Met
FR-001-01: Real-time Data Collection	FS-01, NFS-01, TS-04, TS-05	Met
FR-001-02: Data Access for Maintenance	FS-01, FS-02	Met
FR-002-01: System Interoperability	FS-03, TS-01	Met
FR-003-01: Flexible and Scalable System	FS-04, TS-02	Met
FR-004-01: User Interface for Data Visualization	FS-05, NFS-01	Partially Met
FR-005-01: OPC UA Information Model Compliance	NFS-05	Met
NFR-001-01: System Performance	NFS-01, TS-05	Met
NFR-002-01: Data Security	NFS-02	Partially Met
NFR-003-01: High Availability and Fault Tolerance	NFS-03, TS-02	Met
NFR-004-01: Maintainability	NFS-04, TS-02, TS-06	Met
NFR-005-01: Compliance with Industry Standards	NFS-05	Met
TR-001-01: OPC UA Server Implementation	TS-02, TS-04	Met
TR-002-01: Containerized Deployment	TS-03, NFS-03	Met
TR-003-01: Dynamic Simulation of Field Element Changes	TS-04	Met
TR-004-01: Real-Time Data Synchronization	TS-05	Met
TR-005-01: UML Mapping to OPC UA Information Model	TS-01	Met
TR-006-01: CI/CD Pipeline Integration	TS-06	Met
OR-001-01: Simulated Deployment Environment	OS-01	Met
OR-002-01: Monitoring and Logging System	OS-02	Partially Met
OR-003-01: Backup and Disaster Recovery Plan	OS-03	Unmet
OR-004-01: Regular System Maintenance	OS-04	Met

Table 6.13.: Mapping Between Requirements, Specifications, and Validation Status

This mapping in Table 6.13 gives a clear way to align requirements with project goals. The majority of requirements are met, however, future work could include extending connectivity beyond the local machine, increased security, additional HMI visualization, sustained monitoring, and a disaster recovery plan.

7. Conclusion & Outlook

7.1. Conclusion

In this thesis, a detailed implementation and implementation of an OPC UA based EU-Rail and EULYNX Interface Specification Standard Diagnostics Interface (SDI) for the Command Control and Signaling (CCS) system is presented. The integration of OPC UA Server in the system is successful, managing the diagnostic data obtained from railway infrastructure, specifically from Point Subsystem. Kubernetes and Helm provides a deployment architecture of scalability, high availability and reliable performance and a continuous integration and continuous development (CI/CD) pipeline supports efficient updates and maintenance.

OPC UA Information Model is used for standardization of the diagnostic interface across railway infrastructure subsystems, as specified in the SDI specification. Testing and validation of the system were conducted and verified the system's capability supporting real time data collection, connectivity and efficient resource utilization to achieve the majority of functional, non functional, technical and operational requirements. Furthermore, the bridging of two complex frameworks (the translation from UML model to OPC UA Information Model) can be regarded as an achievement on its own. This translation enriches the diagnostic structure in terms of interoperability and seamless integration with standard field elements of railway components.

7.1.1. Core Contributions

Several contributions to the railway diagnostics and CCS System are brought by the project. These include:

- **Standardized Diagnostic Interface**: Utilization of OPC UA allows for consistent and standardized management and monitoring of diagnostic data, so as to provide interoperability for different railroad components and subsystems.
- **Scalable Architecture**: The containerization of OPC UA servers and their orchestration through Kubernetes and Helm support simple deployment. The system architecture supports the seamless bringup of subsystems or field elements for future system expansion.
- **Automated Deployment and Maintenance**: CI/CD pipeline improves the development and operation, such that deployments, updates and fixes are performed efficiently to improve system maintainability.
- **Simulation-Based Scenario Testing**: Since actual trackside equipment is unavailable, the project uses an Excel-based scenario as input to simulate the data for testing. It shows how the system can be used to diagnose simulated scenarios, preparatory work for future integration with real equipment.

7.1.2. Challenges and Limitations

The project encountered several challenges:

- **Security Limitations**: The present implementation is able to support basic connectivity but it does not have a robust support for security features such as encryption and authentication, which make it subject to unauthorized accesses.
- **Limited External Access**: For the diagnostic data to be accessed, deployment in a local Kubernetes cluster limits the access and more configuration or cloud deployment is needed for broader access.
- **Testing Data Limitation**: Although Excel based simulation does offer enough data to test and validate the system functionally, real world diagnostic data measured from actual trackside equipment would allow for a more dynamic and complete evaluation.
- **Absence of Backup and Disaster Recovery**: This feature could not be implemented, as it is currently unsupported by local deployment limits, but is a crucial necessity in the duty of protecting data and system resilience in real world operations.

7.2. Outlook

While this project constitutes a foundational implementation establishing the proof of concept for EU-Rail and EULYNX' Standard Diagnostic Interface (SDI), many opportunities to improve the system beyond proof of concept to production worthy standards exist with respect to robustness, scalability, and long term reliability.

7.2.1. Cloud Hosting with LoadBalancer for External Access

Although this current implementation runs in a local environment, moving the OPC UA server to a cloud platform, for instance, GKE or Amazon EKS or Azure AKS is recommended as a next step. Along with this comes scalability, automated load balancing, and the ability to be externally accessed, being production ready and increasing reliability and accessibility to external clients.

The LoadBalancer service type is recommended to facilitate external access in cloud enironments. It's configurable in the Helm chart. By default, an external IP is assigned automatically giving access to the service from outside the K8s cluster. The YAML configuration for the Helm chart of a LoadBalancer is as follows:

```
service:
  type: LoadBalancer
```

Code Listing 7.1: Helm chart configuration for LoadBalancer

With this configuration, this OPC UA server can be accessed through external networks, which is compatible to cloud hosted applications and increases flexibility of operations. This approach allows future deployments to be more accessible and meet the expectations of cloud based infrastructures presently.

7.2.2. Improved Security Measures

While the current implementation puts priority on functionality, future work is proposed in order to introduce more advanced security measures so that diagnostic data is protected, and the overall system integrity is robust. These improvements will overcome limitations in the current system and bring it in line with better practices for secure industrial applications. It includes some potential improvements:

- **Authentication and Authorization**: The system could implement beyond anonymous access to employ username–password based access control, ensuring only authorized users will access or modify data.
- **Data Encryption**: Using SSL/TLS encryption to implement secure communication channels will provide confidentiality and integrity protection of data in transit, and improve current data security.
- **Certificate Upload for Enhanced Security**: Data safety can be improved using certificate based authentication particularly as the system is expanded to include more critical trackside equipment.
- **Logging and Auditing**: Logging and monitoring can be added to track system activity, address unauthorized access, and notify security problem.

These suggested security measures complement future scalability and operational requirements while ensuring system capability of handling sensitive diagnostic data securely and in a reliable manner.

7.2.3. Continuous Monitoring and HMI Integration

Future work could add Prometheus and Grafana to make real time metrics visualization and gain insights about resource utilization and performance in the system. Also, it would be beneficial to develop Human Machine Interface (HMI) specifically dedicated to Standard Diagnostic Interface (SDI) as an alternative from UaExpert for diagnostics monitoring.

7.2.4. Future Integration with Actual Trackside Equipment

Current system without real lifetime point machines runs based on simulated data through the use of excel inputs, however in future, the live data feeds from operational equipment can be integrated which opens a big opportunity. Simulation dependencies would be replaced with real world diagnostic data, which would give more dynamic, accurate, and realistic diagnostics. This would improve the system's practical application and reliability for real time monitoring applications.

7.2.5. Maintaining Alignment with Evolving SDI Specification

Since EU-Rail and EULYNX Interface Specification will continue to evolve, future work should include updating the OPC UA Information Model in relation to the latest standards. Updates to the model on a regular basis will ensure ongoing continued compatibility and compliance with current industry standards, enabling incremental uptake within the system of new functionality and to accommodate specification changes.

7.2.6. Backup and Disaster Recovery Plan

The future developments should focus on the implementation of complete backup and disaster recovery plan so that data saved and system flexibility is developed in the operational environment. Key include:

- **Automated Backups**: Automated and regular schedule snapshoting of OPC UA server data and configurations in physical or cloud based repositories should be provided, to secure data protection against loss.
- **Disaster Recovery Testing**: Development and periodic verification of restoration procedures to ensure system readiness and reliability at unexpected failures event.

Further incorporating these measures should help make the system more robust and provide continuity when deploying the system in a complex operational deployment.

7.3. Final Remarks

This thesis introduces an OPC UA-based diagnostics interface that is a major step towards standardized and scalable diagnostics for railway infrastructure. As a proof of concept implementation, it represents a successful demonstration of how the EU-Rail and EULYNX Interface Specification Standard Diagnostic Interface (SDI) can be used to provide a reliable, "single source of truth" for the real time monitoring of field assets.

A part of this work was displayed at InnoTrans 2024 (September 23-27, Messe Berlin), under the theme of "End-to-End Data Process for a Digital Railway", as part of the Europe's Rail Joint Undertaking System Pillar demonstration. The thesis supported the Operation & Diagnosis phase of the demo by contributing to the OPC UA Server implementation of the Point subsystem relative to the SDI Base Line 4 Release 3 (BL4R3). It was shown that the system is able to handle critical and non-critical failure scenarios with the demonstrated real-time diagnostic capabilities, which are aligned with European railway standards [44].

Although the current implementation meets key milestones, its limitations will be addressed and proposed future enhancements integrated to elevate the system to production level maturity. Further developing its impact will be the integration of actual trackside equipment, improved security measures, and updating of the OPC UA Information Model with evolving specifications. These advances can serve as a benchmark for interoperable, reliable and scalable diagnostic solutions that will help digitalize railway operations.

Being recognized at InnoTrans 2024 shows that this work is important and aligns with the wider ambition of a Single European Railway Area (SERA). Additionally, it highlights academic research and industry innovation collaboration potential in driving advances in railway diagnosis and standardization.

Bibliography

[1] European Union Agency for Railways. *Interoperability Overview 2023*. Accessed: 2024-05-26. Valenciennes Cedex, France, July 2023. URL: https://www.era.europa.eu/content/era-railway-factsheets.

[2] European Union Agency for Railways. *Report on Railway Safety and Interoperability in the EU 2022*. Accessed: 2024-05-26. Luxembourg, May 2022. DOI: 10.2821/060591. URL: https://www.era.europa.eu/content/report-railway-safety-and-interoperability-eu-2022_en.

[3] EULYNX Partners. *Interface Definition SDI*. Version 3.0 (1.A). Document number: Eu.Doc.77, Accessed: 2024-05-26. June 2023. URL: https://rail-research.europa.eu/wp-content/uploads/2023/06/20230627-Interface-definition-SDI-Eu.Doc.77-v3.0-1.A.pdf.

[4] Europe's Rail Joint Undertaking. *System Pillar*. Accessed: 2024-06-30. Brussels, Belgium, Oct. 2023. URL: https://rail-research.europa.eu/system_pillar/.

[5] Armando W. Colombo. *Industrial Cyber-Physical Systems (ICPS)*. Lecture Series: MII - ICPS, SS23, HS Emden Leer. Mar. 1, 2023.

[6] Armando W. Colombo. *Digitalization of ICPS: The Digitalization Transformation (DICPS)*. Lecture Series: MII-DICPS, SS23, HS Emden Leer. Mar. 1, 2023.

[7] Olaf Bergmann. *Industrial Data Transport Technologies (IDTT)*. Lecture Series: MII-IDTT, SS23, HS Emden Leer. Mar. 1, 2023.

[8] Deutsche Bahn AG. *DB InfraGO*. Accessed: 2024-06-30. Berlin, Germany, June 2024. URL: https://ir.deutschebahn.com/en/db-group/business-units/db-infrago/.

[9] Deutsche Bahn AG. *DB InfraGO AG*. Accessed: 2024-06-30. Berlin, Germany, June 2023. URL: https://www.deutschebahn.com/de/konzern/konzernprofil/Konzernunternehmen/DB-InfraGO-AG-12598696.

[10] Europe's Rail Joint Undertaking. *Work Programme 2024*. Adopted by the EU-Rail Governing Board on 5 December 2023. This document outlines the operational activities, governance structure, and budget for EU-Rail in 2024. Brussels, Belgium, Dec. 2023. URL: https://rail-research.europa.eu/wp-content/uploads/2023/12/GB-Decision_16_Annex_WP2024.pdf.

[11] Deutsche Bahn AG. *Digitale Schiene Deutschland*. Accessed: 2024-06-30. Berlin, Germany, Aug. 2023. URL: https://www.digitale-schiene-deutschland.de.

[12] Ralph Müller. "Digitale Stellwerke tragen die Digitalisierung der Bahn". In: *EIK - Ausrüstung & Betrieb | Funktionsinnovation* 2021 (2021). This article discusses the advancements and challenges in digital interlocking systems and their role in railway digitalization., pp. 180–201. URL: https://digitale-schiene-deutschland.de/Downloads/EIK_2021_Mueller.pdf.

[13] Deutsche Bahn AG. *Digitalisierungsprojekte und -konzepte im Fokus*. Accessed: 2024-06-30. Berlin, Germany, Dec. 2022. URL: https://ibir.deutschebahn.com/2022/de/konzernlagebericht/produktqualitaet-und-digitalisierung/digitalisierung/digitalisierungsprojekte-und-konzepte-im-fokus/.

[14] DB Netz AG. *Die Zukunft der Mobilität: Digitale Schiene Deutschland*. Ed. by Julia Rott and Axel-Björn Hüper. This publication highlights the key technologies, projects, and future outlook for railway digitization in Germany. Leverkusen, Germany: GRT Global Rail Academy and Media GmbH, Sept. 2023. ISBN: 978-3-96245-260-5.

[15] Digitale Schiene Deutschland. *Technologies*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://digitale-schiene-deutschland.de/en/technologies.

[16] Deutsche Bahn AG. *The future of mobility lies in the tracks*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://www.deutschebahn.com/en/Digitalization/digitalrail-6935138.

[17] European Commission. *History of ERTMS*. Accessed: 2024-06-30. Brussels, Belgium, Aug. 2023. URL: https://transport.ec.europa.eu/transport-modes/rail/ertms/history-ertms_en.

[18] European Union Agency for Railways. *Control Command and Signalling TSI*. Accessed: 2024-06-30. Valenciennes Cedex, France, Aug. 2023. URL: https://www.era.europa.eu/domains/technical-specifications-interoperability/control-command-and-signalling-tsi_en.

[19] European Commission. *Commission Implementing Regulation (EU) 2023/1695 of 10 August 2023 on the technical specification for interoperability relating to the control-command and signalling subsystems of the rail system in the European Union and repealing Regulation (EU) 2016/919*. Accessed: 2024-06-30. Brussels, Belgium, Aug. 2023. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32023R1695.

[20] Digitale Schiene Deutschland. *ETCS (European Train Control System)*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://digitale-schiene-deutschland.de/de/technologien/ETCS.

[21] International Union of Railways (UIC). *Future Railway Mobile Communication System (FRMCS)*. Accessed: 2024-06-30. Paris, France, Sept. 2023. URL: https://uic.org/rail-system/telecoms-signalling/frmcs.

[22] Digitale Schiene Deutschland. *FRMCS/5G-Datenkommunikation*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://digitale-schiene-deutschland.de/FRMCS-5G-Datenkommunikation.

[23] Digitale Schiene Deutschland. *Digitale Stellwerke (DSTW)*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://digitale-schiene-deutschland.de/de/technologien/DSTW.

[24] Digitale Schiene Deutschland. *Automatic Train Operation (ATO)*. Accessed: 2024-06-30. Berlin, Germany, July 2024. URL: https://digitale-schiene-deutschland.de/Automatic-Train-Operation.

[25] European Parliament. *Report on railway safety and signalling: assessing the state of play of the European Rail Traffic Management System (ERTMS) deployment*. Accessed: 2024-06-30. Brussels, Belgium, June 2021. URL: https://www.europarl.europa.eu/doceo/document/A-9-2021-0181_EN.html.

[26] Europe's Rail Joint Undertaking. *Europe's Rail Joint Undertaking proposed by the European Commission to the Council and Parliament*. Accessed: 2024-06-30. Brussels, Belgium, Feb. 2021. URL: https://rail-research.europa.eu/news/europes-rail-joint-undertaking-proposed-by-the-european-commission-to-the-european-council-and-parliament/.

[27] Europe's Rail Joint Undertaking. *Europe's Rail Mission and Objectives*. Accessed: 2024-06-30. Brussels, Belgium, Sept. 2023. URL: https://rail-research.europa.eu/about-europes-rail/europes-rail-mission-and-objectives/.

[28] Europe's Rail Joint Undertaking. *Governance Organisation and Working Arrangement of the System Pillar*. This document defines the governance, organizational structure, and working arrangements of the System Pillar within the EU-Rail framework. Brussels, Belgium, June 2022. URL: https://rail-research.europa.eu/wp-content/uploads/2022/07/20220603-System-Pillar-Governance-and-Working-Arrangements.pdf.

[29] Miguel Villeta Espada, Benedikt Wenzel, and Harish Narayanan. *TCCS D3.1 System Definition*. Sept. 2024.

[30] EULYNX Partners. *About Us*. Accessed: 2024-06-30. Europe, July 2024. URL: https://eulynx.eu/about-us/.

[31] Europe's Rail Joint Undertaking. *Trackside Assets Specifications*. Accessed: 2024-06-30. Brussels, Belgium, July 2024. URL: https://rail-research.europa.eu/system_pillar/system-pillar-outputs/trackside-assets-specifications/.

[32] EULYNX Partners. *EULYNX Domain Knowledge*. Version 1.16 (0.A). Document number: Eu.Doc.10, Accessed: 2024-06-30. Europe, June 2023. URL: https://eulynx.eu/storage/simple-file-list/General-Documents/Baseline-set-4-Release-3/20240618-EULYNX-Domain-knowledge-Eu_Doc_10-v1_18-0_A.pdf.

[33] Antonio J. Sala, Jesus Felez, and Juan David Cano-Moreno. "Efficient Railway Turnout Design: Leveraging TRIZ-Based Approaches". In: *Applied Sciences* 13 (Aug. 2023), p. 9531. DOI: 10.3390/app13179531. URL: https://www.mdpi.com/2076-3417/13/17/9531.

[34] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. 2nd. Boston, MA, USA: Addison-Wesley, 2005. ISBN: 0-321-26797-4. URL: https://www.researchgate.net/publication/234785986.

[35] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-540-68898-3. DOI: 10.1007/978-3-540-68899-0.

[36] GitHub, Inc. *What is CI/CD?* Accessed: 2024-06-30. San Francisco, CA, USA, July 2024. URL: https://github.com/resources/articles/devops/ci-cd.

[37] Docker, Inc. *What is Docker?* Accessed: 2024-06-30. San Francisco, CA, USA, July 2024. URL: https://docs.docker.com/get-started/docker-overview/.

[38] Kubernetes. *Kubernetes Documentation*. Accessed: 2024-06-30. San Francisco, CA, USA, July 2024. URL: https://kubernetes.io/docs/home/.

[39] Helm Authors. *Helm Documentation*. Accessed: 2024-06-30. San Francisco, CA, USA, July 2024. URL: https://helm.sh/docs/.

[40] Plattform Industrie 4.0 et al. *Reference Architecture Model Industrie 4.0 (RAMI4.0)*. DIN SPEC 91345. English translation of DIN SPEC 91345:2016-04,accessed: 2023-06-06. 2016. URL: https://www.beuth.de/en/technical-rule/din-spec-91345/2482458.

[41] Karsten Schweichhart. *Reference Architectural Model Industrie 4.0 (RAMI 4.0): An Introduction*. 2016. URL: https://ec.europa.eu/futurium/en/system/files/ged/a2-schweichhart-reference_architectural_model_industrie_4.0_rami_4.0.pdf.

[42] Armando W. Colombo. *Engineering Industrial Cyber-Physical Systems: Life Cycle Engineering of Industrial Cyber-Physical Systems (ICPS)*. Lecture Series: MII-ICPS, WS22/23, HS Emden Leer. Sept. 1, 2022.

[43] Sooryanarayanan Garudath. *Design and Implementation of OPC UA Client for Railway CCS : EULYNX-Based Standard Diagnostics Interface (SDI) within Maintenance and Data Management System (MDM)*. Bachelor's Thesis. Schweinfurt, Germany, Oct. 2024.

[44] Europe's Rail Joint Undertaking. *InnoTrans 2024*. Accessed: 2024-06-30. Berlin, Germany, Sept. 2024. URL: https://rail-research.europa.eu/calendar/innotrans-2024/.

A. Annex

A.1. Detailed Design

A.1.1. UML to OPC UA Mapping Diagrams

This appendix consists of a collection of diagrams to demonstrate how UML concepts and their relations are mapped to the OPC UA Information Model using the UaModeler tool.

Figure A.1 demonstrates how different subsystems and field elements Classes, i.e. Points, Light Signals, Train Detection Systems, are mapped into OPC UA Information Model and their relations.

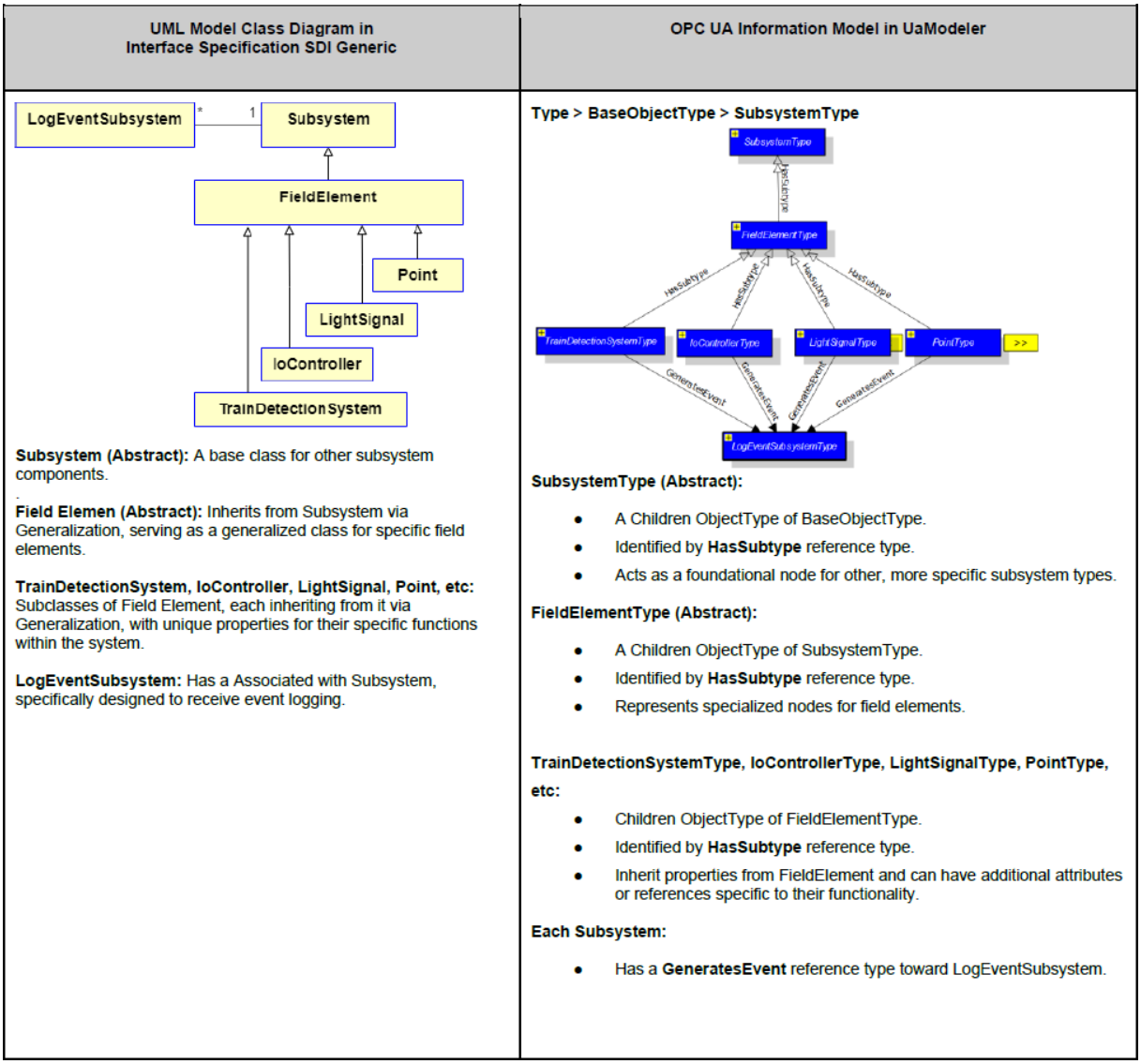


Figure A.1.: UML Class vs. OPC UA Model - Subsystem Mapping, own illustration

Figure A.2 depicts how distinct diagnostic interfaces UML class of each subsystem, e.g., SDI_IO, SDI_P, and SDI_LS, are represented in UML class diagrams and mapped into OPC UA ObjectTypes in the UaModeler. It shows the relationships between these subsystems and their corresponding OPC UA entities, e.g., the use of HasSubtype to model inheritance and GeneratesEvent to express event relationships with the LogEventInterface.

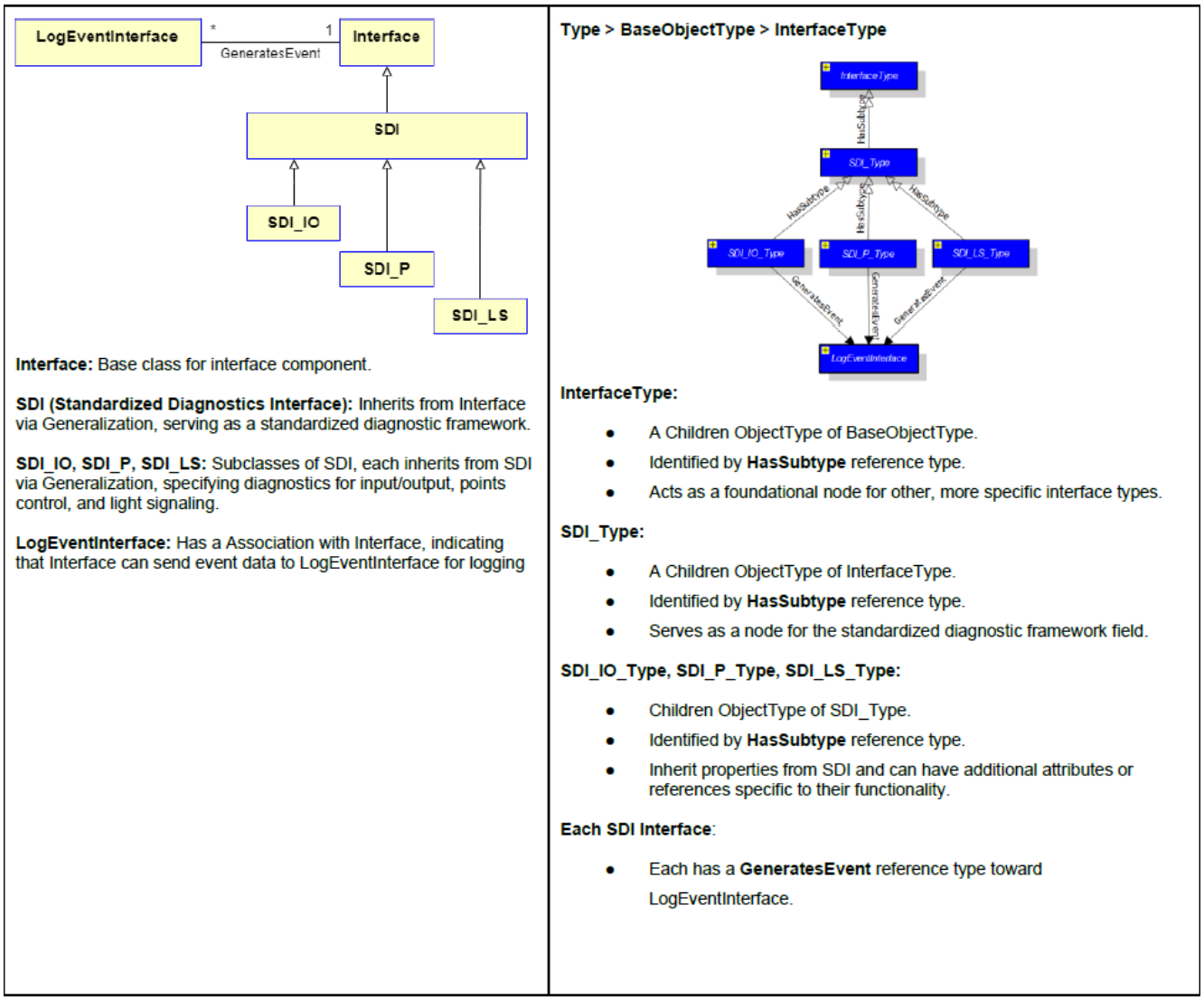


Figure A.2.: UML Class vs. OPC UA Model - Interface Mapping, own illustration

Figure A.3 shows how the OPC UA Server and SCP Classes are mapped into OPC UA Information Model.

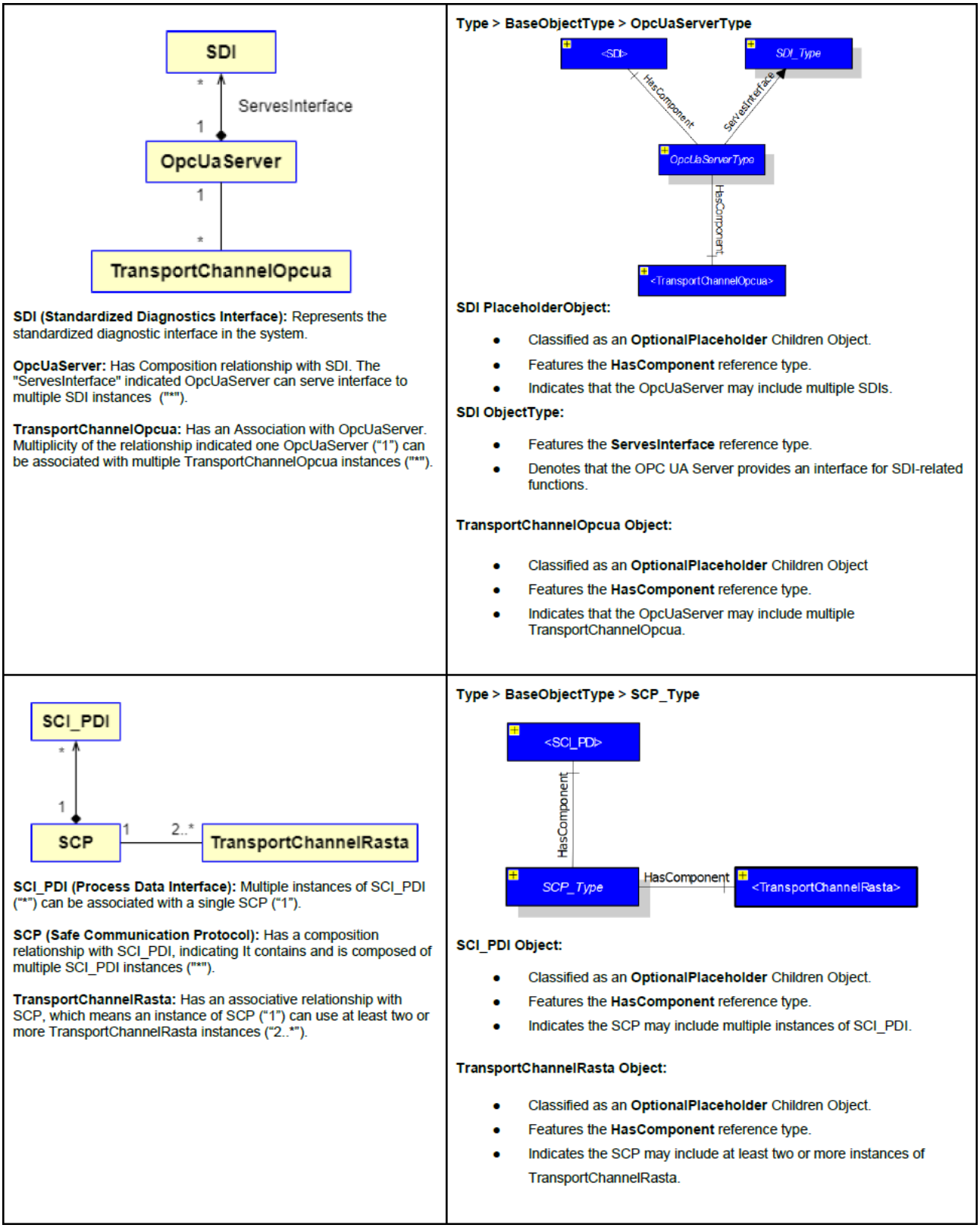


Figure A.3.: UML Class vs. OPC UA Model - OpcUaServer and SCP Mapping, own illustration

Two mappings are shown in Figure A.4. The UML association TransportChannel and TLSCertificate Class represented in OPC UA using the IsTlsCertificateOf reference type is shown in the top section.

The bottom section shows the composition of Equipment and the related PhysicalInput/Output in UML Class which is reflected in OPC UA Information Model via HasComponent reference with several optional placeholders for physical inputs and outputs.

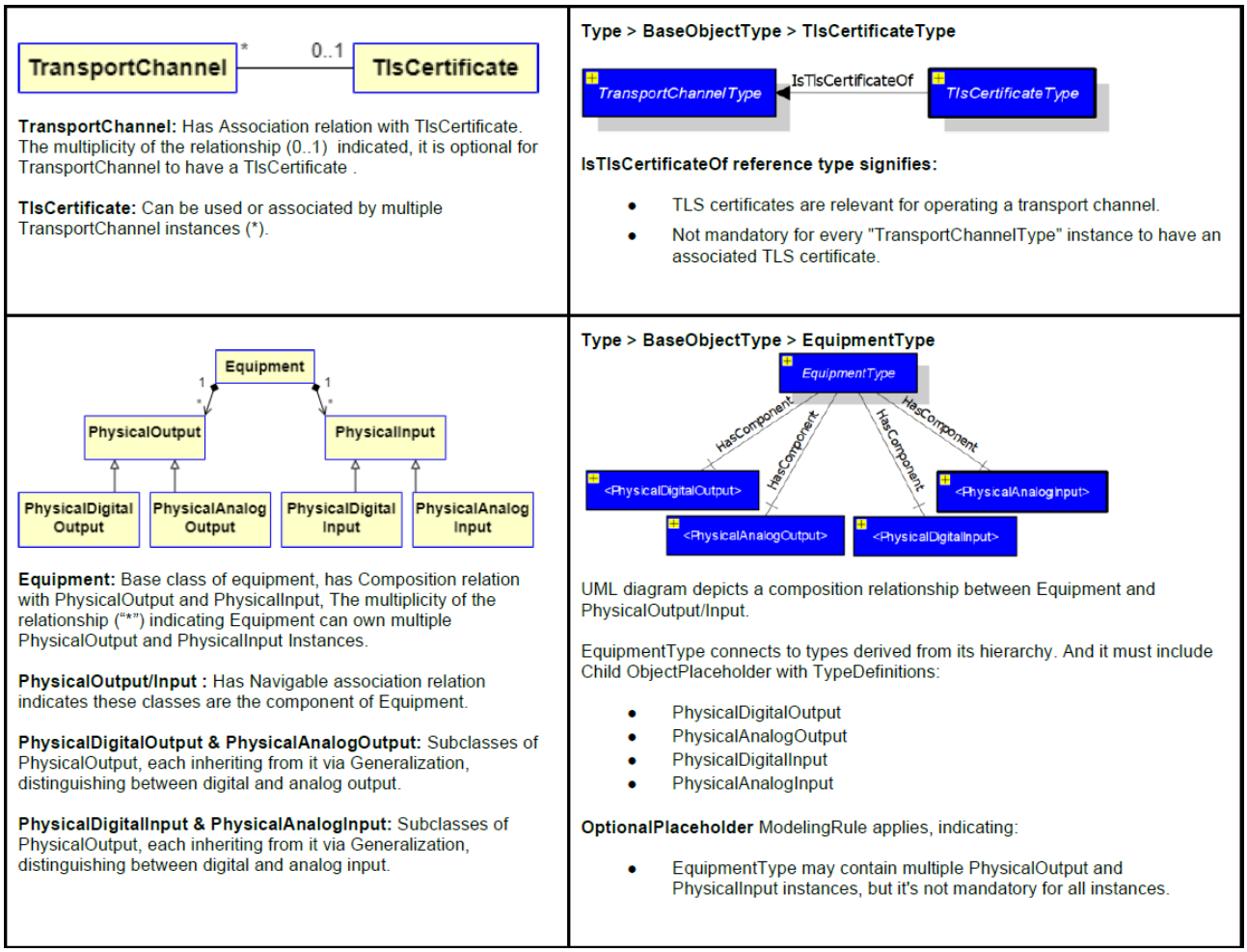


Figure A.4.: UML Class vs. OPC UA Model - TransportChannel and Equipment Mapping, own illustration

Figure A.5 shows the mapping of the BaseEvent UML Class and its derived classes to the OPC UA Information Model. BaseEvent is inherited by the LogEvent, LogEventSubsystem, and LogEventInterface and represented in OPC UA using HasSubtype to represent relationships between these event types. LogEventInterfacePdiEvent inherited from LogEventInterface and has special attributes for the Pdi events.

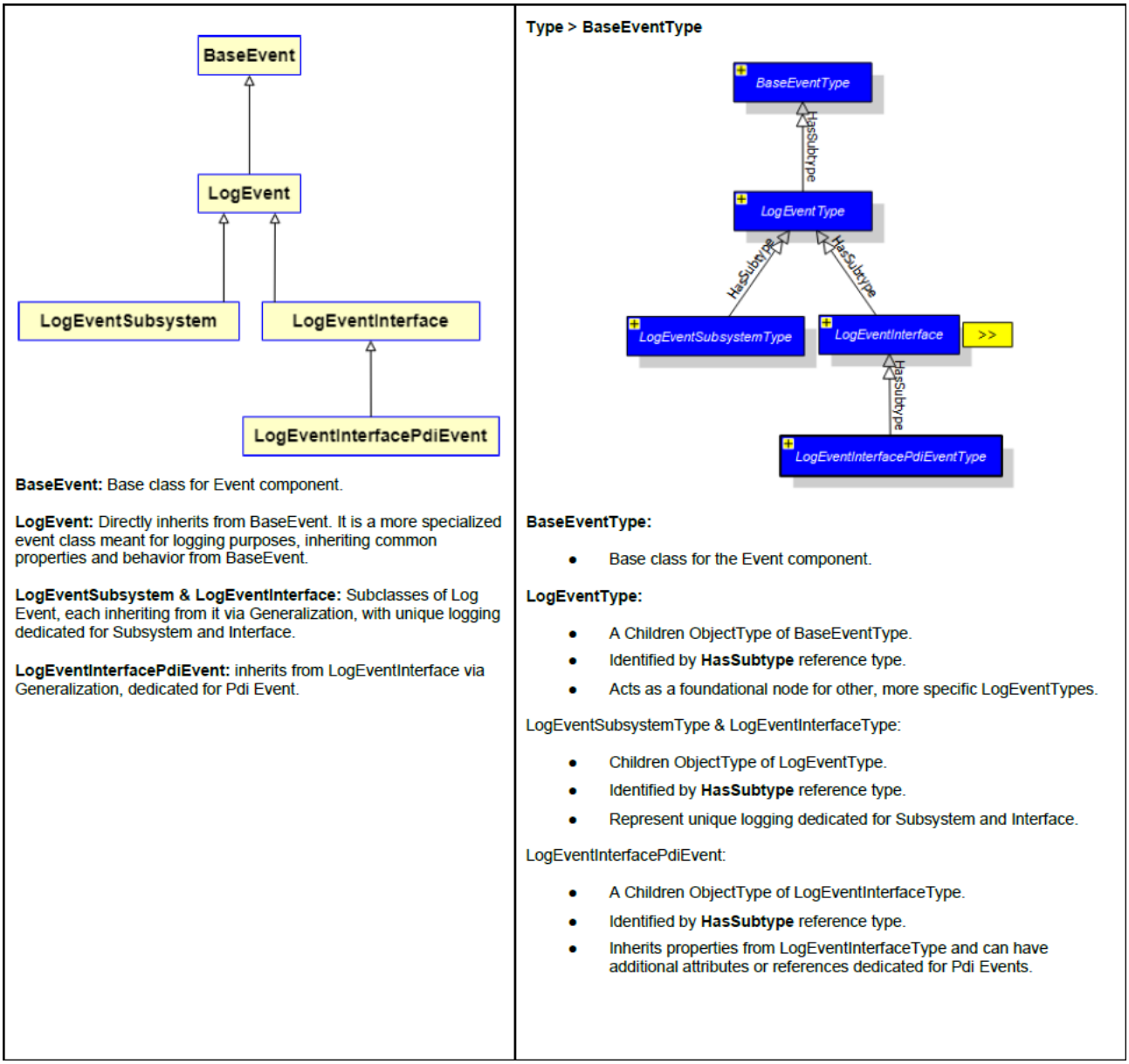


Figure A.5.: UML Class vs. OPC UA Model - BaseEvent Mapping, own illustration

A. Annex

The mapping between the UML Model Class and the OPC UA Information Model for LightSignal subsystem as shown on Figure A.6. LightSignalType maps abstract LightPoint class and its subclasses to FieldElementType via HasSubtype. Interactions with physical output devices are defined using HasPhysicalReferenceChannel to connect PhysicalOutputType with IndicatorType.

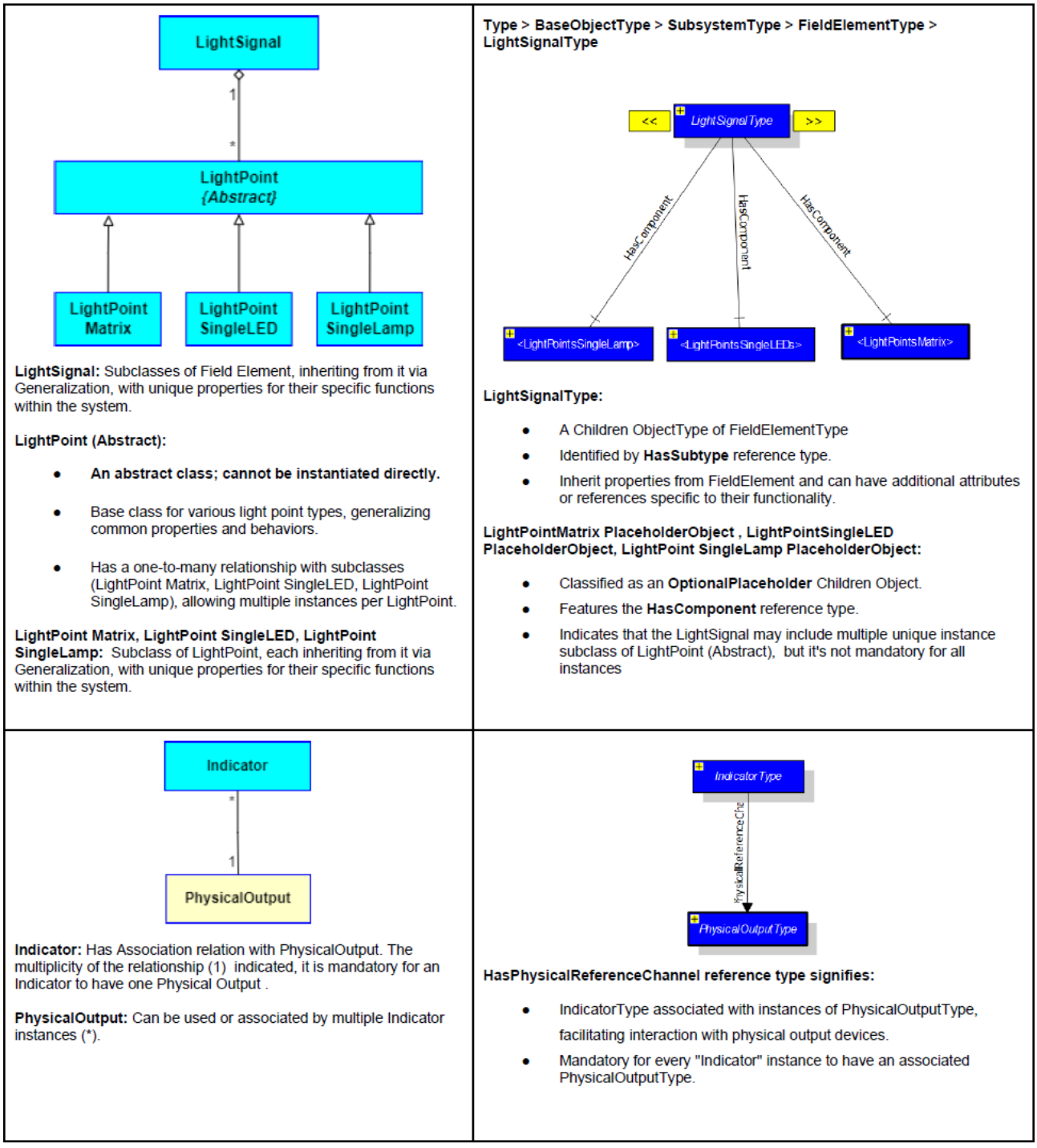


Figure A.6.: UML Class vs. OPC UA Model - LightSignal and Indicator Mapping, own illustration

A.2. Implementation

A.2.1. UaModeler Examples

In this section an example of the modelling of ObjectTypes, Variables and Enumeration DataTypes is shown, using the UaModeler with EU-Rail and EULYNX Interface Specification SDI, as an example. The explanations in the main text are complemented by these examples.

Example of ObjectType Creation in UaModeler

In UaModeler, ObjectTypes define attributes, variables and references. The following Figure A.7 shows how the SubsystemType ObjectType was created according to the SDI Specification.

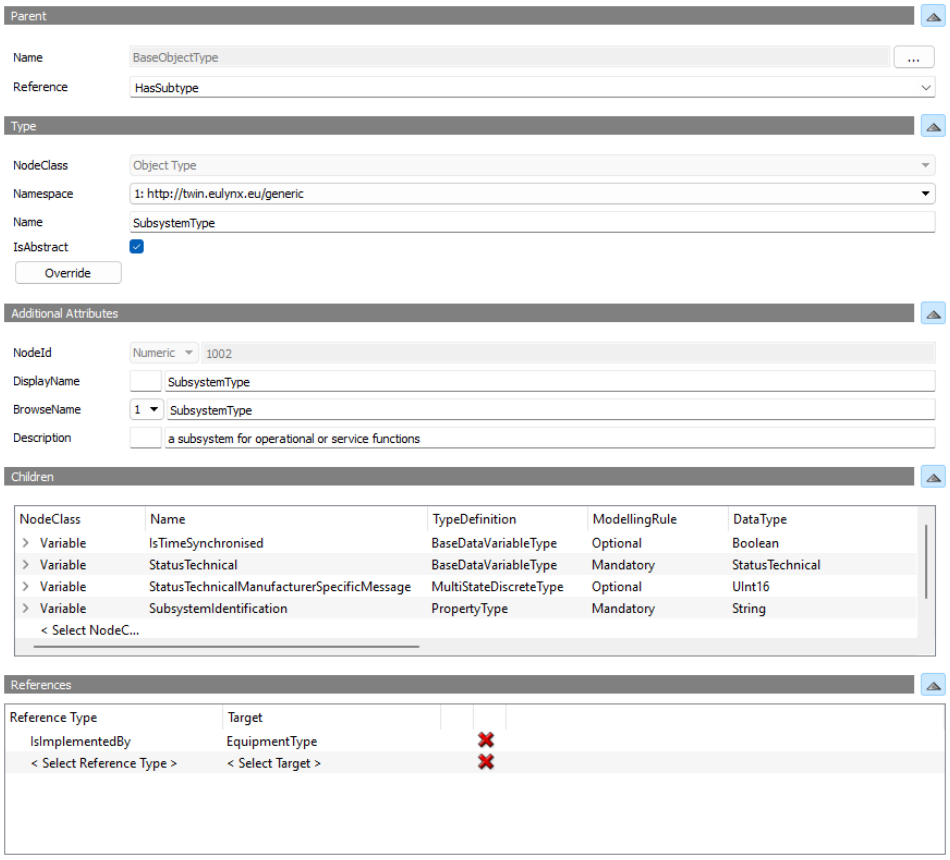


Figure A.7.: SubsystemType ObjectType Creation in UaModeler, own illustration

Key components of the SubsystemType ObjectType include:

- **NodeClass:** ObjectType, meaning a particular operational or service function.
- **Parent:** BaseObjectType is the parent node which is where its hierarchical relationship comes from.
- **Attributes:** Node ID includes DisplayName and a description as to what the system is.
- **Children:** Child nodes of the SubsystemType are variables such as IsTimeSynchronized, StatusTechnical and SubsystemIdentification.
- **ReferenceType:** Creates relationships between ObjectTypes, for example IsImplementedBy connecting SubsystemType to the EquipmentType to which it belongs.

This example is based on a structured approach for maintaining compliance with the SDI Specification.

Example of Variable Creation in UaModeler

Attributes of a variable in UaModeler are defined to describe the variable's role in the Information Model. The variable LastCommandedPosition is shown being created in Figure A.8 below.

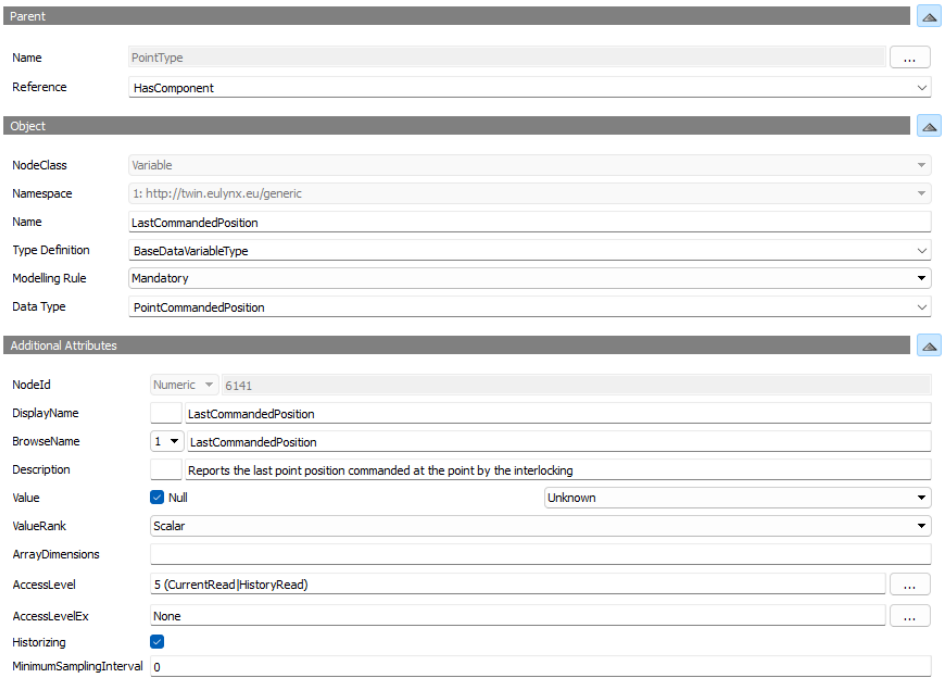


Figure A.8.: Creating the LastCommandedPosition Variable in UaModeler, own illustration

Attributes of the LastCommandedPosition variable include:

- **NodeClass:** Variable, Denoting a diagnostic data point of the point system.
- **Parent:** Belongs to PointType by using the HasComponent reference.
- **Attributes:**
  - **Namespace:** 1: http://twin.eulynx.eu/generic, meaning it is within the generic namespace.
  - **Name:** LastCommandedPosition, the last commanded point position.
  - **NodeId:** This variable is assigned a unique identifier 6141.
- **Type Definition:** BaseDataVariableType, This node is defined as a basic data variable.
- **Modelling Rule:** Mandatory, This variable is required in the context of all instances of PointType.
- **Data Type:** PointCommandedPosition, An enumeration indicating commanded position of the point.
- **Additional Attributes:**
  - **ValueRank:** Scalar, meaning that the variable contains only one value rather than multiple.
  - **AccessLevel:** 5 (CurrentRead/HistoryRead), the variable can be read and historical values can be stored for that variable.
  - **Historizing:** True, allow the access to the historical records to the data.
  - **Description:** "Reports the last point position commanded at the point by the interlocking system."

This variable is necessary to capture and interpret diagnostic data in the point system of railway infrastructure.

Example of Enumeration DataType Creation in UaModeler

DataTypes Enumeration allows the controlled representation of pre defined values. The below Figure A.9 shows how the PointPosition Enumeration DataType was defined in UaModeler.

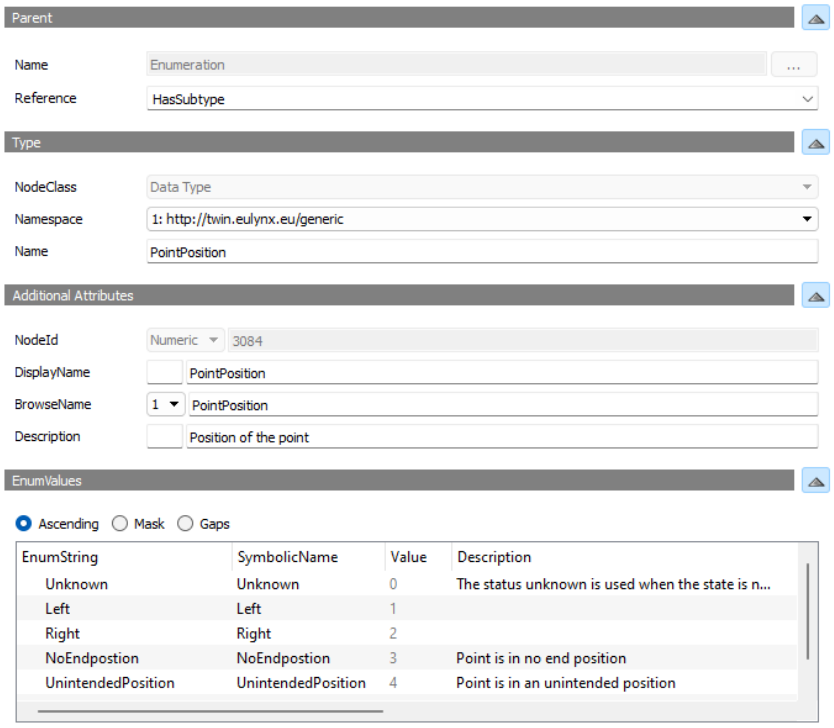


Figure A.9.: Creating the PointPosition Enumeration DataType in UaModeler, own illustration

Key attributes and values of the PointPosition DataType include:

- **NodeClass:** DataType (Enumeration), a list of predefined values represented by DataType.
- **Namespace:**1: http://twin.eulynx.eu/generic, part of the generic namespace indicating.
- **Name:** PointPosition, represents the possible positions of the point.
- **NodeId:** Numeric=3084, unique identifier is assigned to this DataType.
- **EnumValues:** The possible values of the PointPosition enumeration are included in this list:
  - **Unknown (0):** The point status is unknown.
  - **Left (1):** The point is positioned to the left.
  - **Right (2):** The point is positioned to the right.
  - **NoEndPosition (3):** The point has not reached an end position.
  - **UnintendedPosition (4):** The point is in an unintended position, likely due to a failure.

As is displayed in this example, enumeration types are created in UaModeler to maintain standardization of diagnostic data between implementations and to allow for flexibility of those implementations to support a specific operational state.

Example of ReferenceType Creation in UaModeler

UaModeler ReferenceTypes define the relationships between nodes inside the Information Model. The figure A.10, shown below, shows the creation of the IsImplementedBy ReferenceType used in UaModeler to represent an implementation relationship between subsystems and the equipment.

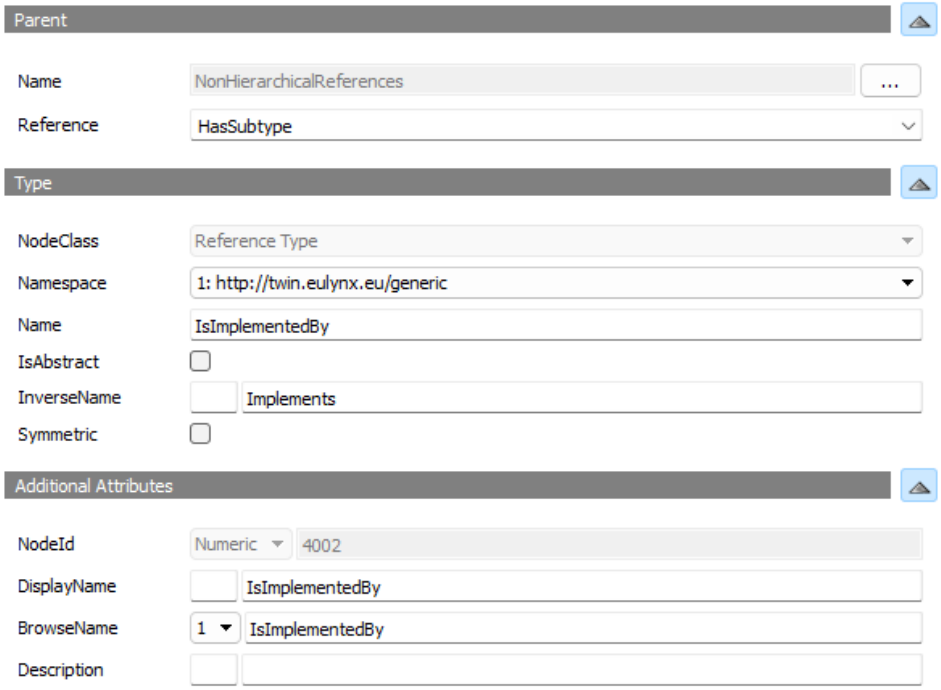


Figure A.10.: Creating the IsImplementedBy ReferenceType in UaModeler, own illustration

Key components of the *IsImplementedBy* ReferenceType include:

- **NodeClass:** ReferenceType, defining a relationship between two nodes.
- **Parent:** NonHierarchicalReferences, classification as a subtype of a non-hierarchical reference.
- **ReferenceType:** HasSubtype, specifying the type of reference it has associated.
- **Namespace:** 1: http://twin.eulynx.eu/generic, showing that it is in the generic namespace.
- **Name:** IsImplementedBy, indicating the implementation relationship between nodes.
- **InverseName:** Implements, showing the inverse reference name for the relationship.
- **NodeId:** Numeric=4002, unique identifier is assigned to this ReferenceType.
- **DisplayName:** IsImplementedBy, providing a human-readable name for this ReferenceType.
- **BrowseName:** IsImplementedBy, used in the OPC UA Information Model for browsing references.
- **Symmetric:** The reference is not symmetric, this checkbox is unchecked.

This example shows how ReferenceTypes are defined in UaModeler, to make it possible to create meaningful relationships between nodes within the OPC UA Information Model.

A.2.2. Variables and Enumeration for OPC UA Instances

Detailed examples of key variables and enumeration values for OPC UA instances for Points and Object Controller Points are provided in this appendix. These variables are important in monitoring and controlling the operation of some railway components. Examples are defined for variables that involve Point instances (e.g., S1) and Object Controller Point instances (e.g., S11–S15) with accompanying Enumeration values describing diagnostic and configuration statuses.

The presented examples serve as an extended discussion of the Information Model structure in the main body of the thesis (Implementation Subsection 5.2.4 - Variables for Point and Object Controller Instances). It deals with the derived variables from the generic and manufacturer specific namespaces, that extend the basic OPC UA model to provide railway component with specific diagnostic and operational attributes.

Variables for Point Instances

Point instances (S1–S15) include variables that provide key information about the point's operational status such as position, movement, and last commanded position. These are derived from the manufacturer specific namespace extending the generic namespace to accommodate manufacturer specific attributes. This structure enhances the abilities of the system for the real time monitoring and fault detection.

Table A.1 below provides an example of the key variables for Point S1 with their data types, node IDs, and the original ObjectType from which they are inherited.

Name	Parent	DataType	NodeId	AttributeType	Inherited From
LastCommanded Position	S1	PointCommandedPosition (Enumeration)	ns=2;i=6070	Diagnostic	PointType
PositionDegraded	S1	PointPositionDegraded (Enumeration)	ns=2;i=6074	Diagnostic	PointType
MovementStatus	S1	PointMovementStatus (Enumeration)	ns=2;i=6075	Configuration	PointType
PointOperationTimer	S1	Real (Float)	ns=2;i=6078	Configuration	PointType
StatusTechnical	S1	StatusTechnical (Enumeration)	ns=2;i=6080	Diagnostic	SubsystemType
IsTimeSynchronised	S1	Boolean	ns=2;i=6081	Diagnostic	SubsystemType
StatusTechnical Manufacturer SpecificMessage	S1	MultiState DiscreteTypeSupplier (UInt16)	ns=2;i=6082	Diagnostic	SubsystemType
SubsystemIdentification	S1	String	ns=2;i=6083	Configuration	SubsystemType
BasicDataReadable	S1	BasicDataReadable (Enumeration)	ns=2;i=6084	Diagnostic	FieldElementType
OperationStatus	S1	FieldElement OperationStatus (Enumeration)	ns=2;i=6086	Diagnostic	FieldElementType
FieldElement SpecificationRevision	S1	String	ns=2;i=6085	Configuration	FieldElementType

Table A.1.: Key Variables for Point S1 in the Manufacturer Namespace with Inheritance Information

The above variables give diagnostics and configuration information required for monitoring the operational status of Point S1. Efficient fault detection and real time system analysis is supported by these variable.

Variables for PointMachine Instances

Each switch (SXX) has a corresponding PointMachine which is responsible for physical operations such as switching, holding position. PointMachine variables enhance the point with precise state and performance information, making for accurate real time monitoring.

Attributes such as operational status, position or movement capabilities of the PointMachine instances corresponding to Point S1 are summarized in the following Table A.2.

Name	Parent	DataType	NodeId	AttributeType	Inherited From
AbleToMoveStatus	PointMachine_S1	PointAbleToMoveStatus (Enumeration)	ns=2;i=6020	Diagnostic	PointMachineType
Position	PointMachine_S1	PointPosition (Enumeration)	ns=2;i=6066	Diagnostic	PointMachineType
MachineType	PointMachine_S1	PointMachine_Type (Enumeration)	ns=2;i=6065	Diagnostic	PointMachineType
TimeOffsetStartLeft	PointMachine_S1	Real (Float)	ns=2;i=6021	Configuration	PointMachineType
TimeOffsetStartRight	PointMachine_S1	Real (Float)	ns=2;i=6022	Configuration	PointMachineType
TurnCounter	PointMachine_S1	Long (UInt64)	ns=2;i=6067	Counter	PointMachineType

Table A.2.: Key Variables for PointMachine S1 in the Manufacturer Namespace with Inheritance Information

These variables are critical diagnostic information for PointMachine S1, such as its movement, timers and machine type. With this breakdown, real time monitoring can be used to predict maintenance for more efficient railway operations.

Variables for Object Controllr Point Instances

For each Object Controller Point instance, key variables related to data describing diagnostic and configuration data necessary for monitoring and control are associated. These variables are similar to Point instances in that they are derived from the manufacturer specific namespace, which extends the generic namespace by adding manufacturer specific attributes. They support fault detection, diagnostics and operational control of Object Controller Points within the railway system.

Below is the Table A.3 with key variables of Object Controller Point S11–S15 with their data types and node IDs and their functional category (diagnostic or configuration).

Name	Parent	DataType	NodeId	AttributeType	Inherited From
IsTimeSynchronised	OCP_S11_S15	Boolean	ns=2;i=7285	Diagnosis	EquipmentType
Replacement Indication	OCP_S11_S15	Equipment ReplaceabilityStatus (Enumeration)	ns=2;i=7109	Diagnosis	EquipmentType
StatusTechnical	OCP_S11_S15	StatusTechnical (Enumeration)	ns=2;i=7112	Diagnosis	EquipmentType
ManufacturerModel	OCP_S11_S15	String	ns=2;i=7107	Configuration	EquipmentType

Table A.3.: Key Variables for Object Controller Point S11–S15 in the Manufacturer Namespace

These variables are critical to understanding the diagnostic and operational status of Object Controller Point S11–S15. This data supports real time monitoring, allowing for actual time intervention as well as increasing the reliability of the total system.

Variables for UnitComputingSafe

Each object controller is made up of four main units. The following table is an example of variables inside of the UnitComputingSafe, a unit in each instance of Object Controller Point that keeps information related to controllers, network interfaces, and storage systems, as from a safe operations perspective. The stability and redundancy of the Object Controller Points can be ensured with monitoring of these variables to help in case of system failures.

Examples of key variables from the UnitComputingSafe in OCP S11–S15 are given in the Table A.4 below.

Name	Parent	DataType	NodeId	AttributeType	Inherited From
IsTimeSynchronised	UnitComputingSafe	Boolean	ns=2;i=7289	Diagnosis	EquipmentType
StatusTechnical	UnitComputingSafe	StatusTechnical (Enumeration)	ns=2;i=7085	Diagnosis	EquipmentType
ManufacturingDateTime	UnitComputingSafe	DateTime	ns=2;i=7081	Configuration	EquipmentType
SerialNumber	UnitComputingSafe	String	ns=2;i=7083	Configuration	EquipmentType
CoolingFanStatus	Controller_S01	CoolingFanStatus (Enumeration)	ns=2;i=7379	Diagnosis	ControllerType
TemperatureStatus	Controller_S01	TemperatureStatus (Enumeration)	ns=2;i=7153	Diagnosis	ControllerType
CpuHealthStatus	Controller_S01	CpuHealthStatus (Enumeration)	ns=2;i=7134	Diagnosis	ControllerType
BootingLastReason	Controller_S01	ControllerResetReason (Enumeration)	ns=2;i=7370	Diagnosis	ControllerType
IsAvailable	Controller RedundancyGroup	Boolean	ns=2;i=7167	Diagnosis	Redundancy GroupType
MinimumAvailable	Controller RedundancyGroup	Ineger (UInt16)	ns=2;i=7169	Configuration	Redundancy GroupType
StatusTechnical	PhysicalNetwork Interface_S01	StatusTechnical (Enumeration)	ns=2;i=7178	Diagnosis	PhysicalNetwork InterfaceType
OperationStatus	PhysicalNetwork Interface_S01	PhysicalNetworkInterface OperationalStatus (Enumeration)	ns=2;i=7177	Diagnosis	PhysicalNetwork InterfaceType
MacAddress	PhysicalNetwork Interface_S01	String	ns=2;i=7175	Configuration	PhysicalNetwork InterfaceType
NominalBandwidth	PhysicalNetwork Interface_S01	Long (UInt64)	ns=2;i=7176	Configuration	PhysicalNetwork InterfaceType
StatusTechnical	UCS_Stroage	StatusTechnical (Enumeration)	ns=2;i=7194	Diagnosis	StorageMedium FlashMemoryType
WearStatus	UCS_Stroage	WearStatus (Enumeration)	ns=2;i=7195	Diagnosis	StorageMedium FlashMemoryType
MemorySize	UCS_Stroage	Integer (UInt64)	ns=2;i=7193	Configuration	StorageMedium FlashMemoryType

Table A.4.: Key Variables for UnitComputingSafe of OCP S11–S15 in the Manufacturer Namespace

These variables allow continuous monitoring, failure detection and also optimise reliable operation by minimizing downtime and supporting efficient railway management.

Enumeration Values for Point Instance's Variables

The operational status and behavior of both Point and PointMachine instances themselves are represented by predefined enumeration values used as diagnostic variables. These enumerations are important for understanding the point's and point machine's state, and to be able to assess functionality clearly, particularly in fault detection and failure management.

An example of enumeration values of Point S1 and associated PointMachine S1 is presented in the Table A.5.

Variable	Enumeration	Description
PointAbleToMoveStatus	0 = Unknown	Unknown status, possibly due to connection loss.
	1 = Able	Able to move, fully operational.
	2 = NotAble	Unable to move, indicating a fault or obstruction.
PointCommandedPosition	0 = Unknown	Commanded position is unclear or system disconnected.
	1 = Left	Commanded to move left.
	2 = Right	Commanded to move right.
PointPosition	0 = Unknown	Current position not established, or communication is lost.
	1 = Left	Positioned to the left.
	2 = Right	Positioned to the right.
	3 = NoEndPosition	No defined end position reached.
	4 = UnintendedPosition	In an unintended position, likely due to failure.
PointMovementStatus	0 = Unknown	Unknown status, possibly due to connection loss.
	1 = MovingToLeft	Point is moving to the left.
	2 = MovingToRight	Point is moving to the right.
	3 = NotMoving	Point is not moving.
StatusTechnical	0 = Unknown	Status unknown due to unestablished system state.
	1 = Ok	System fully operational with no detected issues.
	2 = Warning	Subsystems operational, unexpected behavior detected.
	3 = FailureNonCritical	Error detected, but all functions available.
	4 = FailureCritical	Function unavailable, operational consequences possible.
PointMachine_Type	0 = Unknown	Status unknown due to unestablished system state.
	1 = Drive	Point machine driving point's movable element.
	2 = Downholder	Point machine holding position of movable element.
	3 = DetectorOnly	Point machine detecting position of movable element.

Table A.5.: Explanation of Enumeration Values for Point S1 Variables

The point machine operational status and commanded actions are being measured by these enumeration values, which are essential for diagnostic information. Such configuration facilitates fault detection and system analysis for reliable performance, timely interventions when needed.

Enumeration Values for Object Controller Instance's Variables

For Object Controller Points, enumeration values are defined for predefined operational states of components such as controllers, network interfaces, and redundancy groups. These are important for understanding diagnostic information reported by each unit so as to perform real time system analysis and troubleshooting.

For example, enumeration values assigned to Object Controller Point S11–S15 and UnitComputing can be found in Table A.6 below.

Variable	Enumeration	Description
ControllerOperationStatus	0 = Unknown	Unknown status, connection possibly lost.
	1 = Booting	Controller is booting up and is not ready.
	2 = InOperation	Controller is in regular operation
	3 = Fallback	Controller is in the fallback mode
CoolingFanStatus	0 = Unknown	Unknown status, connection possibly lost.
	1 = Normal	Functioning according to specifications
	2 = Failure	Not functioning
TemperatureStatus	0 = Unknown	Unknown status, connection possibly lost..
	1 = Normal	CPU Temperature is normal
	2 = TooHigh	CPU Temperature is too high
PhysicalNetworkInterface	0 = Unknown	Unknown status, connection possibly lost.
OperationalStatus	1 = NotAvailableNotConnected	Network interface not available or connected.
	2 = AvailableNotConnected	Hardware ready but not connected.
	3 = Connected	The first 2 layers of PoS-Signalling are running
	4 = NotConnectedDisturbed	Network interface disturbed, connection lost.
ControllerResetReason	0 = Unknown	Unknown status, connection possibly lost.
	1 = OnSite	A reset by someone on-site
	2 = RemoteMdm	A remote reset by the MDM
	3 = InternalMaintenanceOk	Successful reset during configuration activation.
	4 = InternalMaintenanceFailure	Failed reset during configuration activation.
	5 = InternalFailure	Internal error caused controller reset.
WearStatus	0 = Unknown	Unknown status, connection possibly lost.
	1 = Ok	Flash memory wear status is normal.
	2 = Warning	Flash memory wear approaching warning levels.
	3 = Nok	Flash memory wear status not acceptable.

Table A.6.: Explanation of Enumeration Values for OCP S11-S15 Variables

These enumeration values are useful for diagnostic information about Object Controller Point's operational status and actions it performs inside of its components. This helps in detecting faults and analyzing the system in a reliable way, and helps intervene at the right time in case of requirement.

A.2.3. Detailed Variable States for Scenario Simulation

Detailed Scenario 1: Critical Failure – Point Switch Failure

This appendix presents the variables, NodeIDs, and the transitions of their states, for Scenario 1 (see Subsection 4.3.3 - Critical Failure – Point Switch Failure in Chaprter 4 - Detailed Design). Diagnostic variables for Point S10 and Object Controller Points S6–S10 during failure and recovery are shown in Table in Figure A.11.

Name	Parent	DataType	ns	i	Enumeration	Step 1: Initial Position and Command	Step 2: Point Fails to Reach End Position	Step 3: Repair Initiated	Step 4: Repair Completed	Step 5: Move Left to Right	Step 6: Reached Right	Step 7: Move Right to Left	Step 8: Reached Left
S10	PointProductGroupSet	-	4	5057	-								
aggregateAbleToMoveStatus	S10	PointAbleToMoveStatus (Enumeration)	4	6023	0=Unknown, 1=Able, 2=NotAble	1 (Able)	2 (NotAble)	2 (NotAble)	1 (Able)	1 (Able)	1 (Able)	1 (Able)	1 (Able)
lastCommandedPosition	S10	PointCommandedPosition (Enumeration)	4	6304	0=Unknown, 1=Left, 2=Right	1 (Left)	1 (Left)	1 (Left)	1 (Left)	2 (Right)	2 (Right)	1 (Left)	1 (Left)
pointAbleToMoveStatus	S10	PointAbleToMoveStatus (Enumeration)	4	6026	0=Unknown, 1=Able, 2=NotAble	1 (Able)	2 (NotAble)	2 (NotAble)	1 (Able)	1 (Able)	1 (Able)	1 (Able)	1 (Able)
position	S10	PointPosition (Enumeration)	4	6307	0=Unknown, 1=Left, 2=Right, 3=NoEndPosition, 4=UnintendedPosition	1 (Left)	4 (UnintendedPosition)	4 (UnintendedPosition)	1 (Left)	3 (NoEndPosition)	2 (Right)	3 (NoEndPosition)	1 (Left)
positionDegraded	S10	PointPositionDegraded (Enumeration)	4	6308	0=Unknown, 1=DegradedLeft, 2=DegradedRight, 3=NotDegrade, 4=NotApplicable	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)	4 (NotApplicable)
movementStatus	S10	PointMovementStatus (Enumeration)	4	6305	0=Unknown, 1=MovingToLeft, 2=MovingToRight, 3=NotMoving	3 (NotMoving)	3 (NotMoving)	3 (NotMoving)	3 (NotMoving)	2 (MovingToRight)	3 (NotMoving)	1 (MovingToLeft)	3 (NotMoving)
StatusTechnical	S10 (SubsystemType)	StatusTechnical (Enumeration)	4	6313	0=Unknown, 1=Ok, 2=Warning, 3=FailureNonCritical, 4=FailureCritical	1 (OK)	4 (FailureCritical)	4 (FailureCritical)	1 (OK)	1 (OK)	1 (OK)	1 (OK)	1 (OK)
PointMachine_S10	S10	-	4	5060	-								
ableToMoveStatus	PointMachine_S10	PointAbleToMoveStatus (Enumeration)	4	6030	0=Unknown, 1=Able, 2=NotAble	1 (Able)	2 (NotAble)	1 (Able)	1 (Able)	1 (Able)	1 (Able)	1 (Able)	1 (Able)
position	PointMachine_S10	PointPosition (Enumeration)	4	6300	0=Unknown, 1=Left, 2=Right, 3=NoEndPosition, 4=UnintendedPosition	1 (Left)	4 (UnintendedPosition)	4 (UnintendedPosition)	1 (Left)	3 (NoEndPosition)	2 (Right)	3 (NoEndPosition)	1 (Left)
PointTurnEventType	LogEventSubsystemType	-	2	1123	-								
IsEndpositionReached	PointTurnEventType	Boolean	2	6806		False	False	False	False	False	True	False	True
CommandedPosition	PointTurnEventType	PointCommandedPosition (Enumeration)	2	7608	0=Unknown, 1=Left, 2=Right	1 (Left)	1 (Left)	1 (Left)	1 (Left)	2 (Right)	2 (Right)	1 (Left)	1 (Left)
FailureReason	PointTurnEventType	PointTurnFailureStatus (Enumeration)	2	6808	0=None, 1=TimeOut, 2=UnsuccessfulStartOfMovement, 3=AllPmStoppedButCommandedPositionNotReached, 4=NoDrivePower, 5=Other	0 (None)	2 (UnsuccessfulStartOfMovement)	2 (UnsuccessfulStartOfMovement)	0 (None)	0 (None)	0 (None)	0 (None)	0 (None)

Figure A.11.: Detailed Scenario 1: Critical Failure – Point S10 Fails to Reach End Position, own illustration

Point S10 variables in Figure A.11 are tracked through each simulation phase from the initial command through recovery after failure in the table. And this validates the fact that OPC UA server can record and process the state transitions correctly.

Detailed Scenario 2: Non-Critical Failure – Object Controller Failure

In this appendix, the diagnostic variables and state transitions of Scenario 2 (see Subsection 4.3.4 - Non-Critical Failure – Object Controller Failure in Chaprter 4 - Detailed Design), concerning the Object Controller Point S6 S10 and its subcomponents, specifically Object Controller S1 and the Unit Computing Safe, are presented.

NodeClass	Name	Parent	DataType	ns	i	Enumeration	Step 1: CoolingFan Fail	Step 2: Temp Too High	Step 3: Ram Degrade	Step 4: cpu health degrade and cpu load high	Step 5: Status Warning	Step 6: Ram failure	Step 7: Cpu failure	Step 8: Controller Failure Critical	Step 9: Unit Computing Safe failure	Step 10: Unknown due to replacment	Step 11: After Replace Contoller
ObjectType	ObjectControllerPoint_S6_S10	PointEquipmentSet	-	3	5073	-											
Variable	isTimeSynchronised	ObjectControllerPoint_S6_S10	Boolean	3	7295		True	True	True	True	True	True	True	True	True	True	True
Variable	replacementIndication	ObjectControllerPoint_S6_S10	EquipmentReplaceabilityStatus (Enumeration)	3	6893	0=Unknown, 1=ReplacementNotNeeded, 2=ReplaceableAtOperation, 3=ReplaceableMaintenance, 4=ReplaceableRevalidation	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	2 (ReplacableAtOperation)	2 (ReplacableAtOperation)	1 (ReplacementNotNeeded)
Variable	statusTechnical	ObjectControllerPoint_S6_S10	StatusTechnical (Enumeration)	3	6891	0=Unknown, 1=Ok, 2=Warning, 3=FailureNonCritical, 4=FailureCritical	1(Ok)	1(Ok)	1(Ok)	1(Ok)	1(Ok)	1(Ok)	1(Ok)	1(Ok)	2 (Warning)	2 (Warning)	1(Ok)
ObjectType	UnitComputingSafe	ObjectControllerPoint_S6_S10	-	3	5089	-											
Variable	isTimeSynchronised	UnitComputingSafe	Boolean	3	7299		True	True	True	True	True	True	True	True	False	False	True
Variable	replacementIndication	UnitComputingSafe	EquipmentReplaceabilityStatus (Enumeration)	3	6781	0=Unknown, 1=ReplacementNotNeeded, 2=ReplaceableAtOperation, 3=ReplaceableMaintenance, 4=ReplaceableRevalidation	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	1 (ReplacementNotNeeded)	2 (ReplacableAtOperation)	2 (ReplacableAtOperation)	0 (Unknown)	1 (ReplacementNotNeeded)
Variable	statusTechnical	UnitComputingSafe	StatusTechnical (Enumeration)	3	6859	0=Unknown, 1=Ok, 2=Warning, 3=FailureNonCritical, 4=FailureCritical	0 (Unknown)	0 (Unknown)	0 (Unknown)	0 (Unknown)	2 (Warning)	0 (Unknown)	0 (Unknown)	2 (Warning)	3 (FailureNonCritical)	0 (Unknown)	1(Ok)
Variable	hardwareRevision	UnitComputingSafe	String	3	6271		v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	Null	v1.1
Variable	manufacturingDateTime	UnitComputingSafe	DateTime	3	6765		2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-08-12T15:32:17Z
Variable	serialNumber	UnitComputingSafe	String	3	6857		SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	SN_SC_P_2_V1	Null	SN_OC_P_2_V2
Variable	softwareRevision	UnitComputingSafe	String	3	6858		v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	v1.0	Null	v1.1

Figure A.12.: Detailed Scenario 2: Object Controller Point S6-S10 and Unit Computing Safe, own illustration

Breakdown helps the Object Controller Point S6-S10 and Unit Computing Safe to record and processing each state transition into the OPC UA server in case of non-critical failure of Object Controller.

The state transitions for Controller S1 within the Unit Computing safe of Object Controller Point S6-S10 can be seen in the following table in Figure A.13. Thus, this data gives detailed insights into the behavior of Controller S1 during any phase of failure and recovery.

NodeClass	Name	Parent	DataType	ns	i	Enumeration	Step 1: CoolingFan Fail	Step 2: Temp Too High	Step 3: Ram Degrade	Step 4: cpu health degrade and cpu load high	Step 5: Status Warning	Step 6: Ram failure	Step 7: Cpu failure	Step 8: Controller Failure Critical	Step 9: Unit Computing Safe failure	Step 10: Unknown due to replacment	Step 11: After Replace Controller
ObjectType	OCPS6S10 _Controller_S01	UnitComputingSafe	-	3	5096	-											
Variable	statusTechnical	OCPS6S10_Controller _S01	StatusTechnical (Enumeration)	3	6923	0=Unknown, 1=Ok, 2=Warning, 3=FailureNonCritical, 4=FailureCritical	0 (Unknown)	1 (Ok)	1 (Ok)	1 (Ok)	2 (Warning)	3 (FailureNonCritical)	3 (FailureNonCritical)	4 (FailureCritical)	4 (FailureCritical)	Null	1(Ok)
Variable	statusTechnicalManufacturerSpecificMessage	OCPS6S10_Controller _S01	MultiStateDiscrete TypeSupplier (UInt16)	3	7340		Null	Null	Null	Null	Null	Null	Null	Null	Null	Null	Null
Variable	operationStatus	OCPS6S10_Controller _S01	ControllerOperation Status (Enumeration)	3	6920	0=Unknown, 1=Booting, 2=InOperation, 3=Fallback	2 (InOperation)	2 (InOperation)	2 (InOperation)	2 (InOperation)	2 (InOperation)	2 (InOperation)	3 (Fallback)	3 (Fallback)	3 (Fallback)	0 (Unknown)	1 (Booting)
Variable	coolingFanStatus	OCPS6S10_Controller _S01	CoolingFanStatus (Enumeration)	3	7337	0=Unknown, 1=Normal, 2=Failure	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	2 (Failure)	0 (Unknown)	1 (Normal)
Variable	TemperatureStatus	OCPS6S10_Controller _S01	TemperatureStatus (Enumeration)	3	6924	0=Unknown, 1=Normal, 2=TooHigh	0 (Unknow)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	2 (TooHigh)	0 (Unknown)	1 (Normal)
Variable	cpuHealthStatus	OCPS6S10_Controller _S01	CpuHealthStatus (Enumeration)	3	6917	0=Unknown, 1=Normal, 2=Degraded, 3=Failure	1 (Normal)	1 (Normal)	1 (Normal)	2 (Degraded)	2 (Degraded)	2 (Degraded)	3 (Failure)	3 (Failure)	3 (Failure)	0 (Unknown)	1 (Normal)
Variable	cpuLoadStatus	OCPS6S10_Controller _S01	CpuLoadStatus (Enumeration)	3	6918	0=Unknown, 1=Normal, 2=High, 3=Critical	1 (Normal)	1 (Normal)	1 (Normal)	2 (High)	0 (Unknow)	0 (Unknow)	3 (Critical)	0 (Unknow)	0 (Unknow)	0 (Unknown)	1 (Normal)
Variable	ramHealthStatus	OCPS6S10_Controller _S01	RamHealthStatus (Enumeration)	3	6921	0=Unknown, 1=Normal, 2=Degraded, 3=Failure	1 (Normal)	1 (Normal)	2 (Degraded)	2 (Degraded)	2 (Degraded)	3 (Failure)	3 (Failure)	3 (Failure)	3 (Failure)	0 (Unknown)	1 (Normal)
Variable	bootingLastReason	OCPS6S10_Controller _S01	ControllerReset Reason (Enumeration)	3	7336	0=Unknown, 1=OnSite, 2=RemoteMdm, 3=InternalMaintenanceOk, 4=InternalMaintenanceFailure, 5=InternalFailure	0 (Unknow)	0 (Unknow)	0 (Unknow)	0 (Unknow)	0 (Unknow)	0 (Unknow)	0 (Unknow)	5(InternalFailure)	4 (InternalMaintenance Failure)	0 (Unknown)	3 (InternalMaintenance Ok)
Variable	operatingSystem	OCPS6S10_Controller _S01	String	3	7339		Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Linux Debian Bookworm	Null	Linux Debian Bullseye
Variable	bootingLastDateTime	OCPS6S10_Controller _S01	DateTime	3	7335		2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-07-15T10:32:17Z	2024-08-12T15:32:17Z

Figure A.13.: Detailed Scenario 2: Controller S1 within Unit Computing Safe, own illustration

To verify that the OPC UA Server logs and manages consistently each state of Controller S1, this comprehensive overview of Controller S1 status transitions helps the diagnostics and monitoring effectiveness.

A.2.4. Complete OPC UA Server Implementation Code

This chapter assembles the Python code for the OPC UA server, including setup, event handling and data updates, for scenario based diagnostics.

```python
import asyncio
import pandas as pd
from asyncua import Server, ua
import os
# Start
# Constants for event-specific node names
EVENT_SPECIFIC_NODES = ["isEndpositionReached", "commandedPosition", "failureReason"]
# Load node information from Excel file
def load_node_info(file_path, sheet_name):
    """
    Load node information from an Excel file.
    Parameters:
        file_path (str): The path to the Excel file.
        sheet_name (str): The name of the sheet to read from.
    Returns:
        list[dict]: A list of dictionaries containing node information.
    """
    df = pd.read_excel(file_path, sheet_name=sheet_name, engine='openpyxl')
    if sheet_name == 'default':
        return [
            {
                "name": row["Name"],
                "datatype": row["DataType"].replace("(Enumeration)", "").strip(),
                "ns": int(row["ns"]),
                "i": int(row["i"]),
                "value": row["Value"]
            }
            for _, row in df.iterrows()
            if row["NodeClass"] == "Variable"
        ]
    else:
        return [
            {
                "name": row["Name"],
                "datatype": row["DataType"].replace("(Enumeration)", "").strip(),
                "ns": int(row["ns"]),
                "i": int(row["i"])
            }
            for _, row in df.iterrows()
            if row["NodeClass"] == "Variable"
        ]
# Clean values from the Excel file
def clean_value(value, datatype):
    """
    Clean and convert the value from the Excel file based on its datatype.
    Parameters:
        value: The value to be cleaned.
        datatype (str): The datatype of the value.
    Returns:
        The cleaned value in its appropriate type or None if not applicable.
    """
    if pd.isnull(value) or value == "Null":
        return None
    if "Enumeration" in datatype or "Int32" in datatype or "UInt16" in datatype or "UInt64
                                                    " in datatype:
        try:
            return int(''.join(filter(str.isdigit, str(value).split()[0])))
        except ValueError:
            return None
    elif "Boolean" in datatype:
        return value.lower() == "true" if isinstance(value, str) else bool(value)
    elif "Double" in datatype or "Real" in datatype:
        return float(value)
    elif "String" in datatype:
```

```
        return str(value)
    elif "DateTime" in datatype:
        try:
            dt = pd.to_datetime(value)
            if dt.year < 1601:
                return None
            return dt
        except (ValueError, OverflowError, pd.errors.OutOfBoundsDatetime):
            return None
    return value
# Load scenario steps from Excel file
def load_scenario_steps(file_path, sheet_name):
    """
    Load scenario steps from an Excel file.
    Parameters:
        file_path (str): The path to the Excel file.
        sheet_name (str): The name of the sheet to read from.
    Returns:
        list[dict]: A list of dictionaries containing scenario step data.
    """
    df = pd.read_excel(file_path, sheet_name=sheet_name, engine='openpyxl')
    steps = []
    for step_column in df.columns:
        if step_column.startswith("Step"):
            step_data = {row["Name"]: clean_value(row[step_column], row["DataType"]) for _
                                                                    , row in df.iterrows()}
            steps.append(step_data)
    return steps
# Update Data Access View nodes
async def update_node_value(server, node_info, value):
    """
    Update the value of a specific node on the server.
    Parameters:
        server (Server): The OPC UA server instance.
        node_info (dict): Information about the node to be updated.
        value: The value to be written to the node.
    Returns:
        None
    """
    node = server.get_node(ua.NodeId(node_info["i"], node_info["ns"]))
    try:
        data_type = await node.read_data_type_as_variant_type()
        cleaned_value = clean_value(value, data_type.name)
        if cleaned_value is None and data_type != ua.VariantType.String:
            print(f"Skipping node: {node_info['name']} due to None value and incompatible
                                                                    type.")
            return
        if isinstance(cleaned_value, pd.Timestamp):
            cleaned_value = cleaned_value.to_pydatetime()
        variant_value = ua.Variant(cleaned_value, data_type)
        await node.write_value(variant_value)
        print(f"Updated {node_info['name']} (ns={node_info['ns']}; i={node_info['i']}) to
                                                            {cleaned_value}")
    except Exception as e:
        print(f"Error writing to {node_info['name']} (ns={node_info['ns']}; i={node_info['
                                                            i']}): {e}")
# Update Data Access View nodes for a scenario step
async def update_data_access_view(server, nodes, scenario_step=None):
    """
    Update Data Access View nodes for a given scenario step.
    Parameters:
        server (Server): The OPC UA server instance.
        nodes (list[dict]): A list of node information dictionaries.
        scenario_step (dict, optional): The data for the current scenario step.
    Returns:
        None
    """
    for node_info in nodes:
        if node_info["name"] in EVENT_SPECIFIC_NODES:
```

```
            continue
        value = scenario_step.get(node_info["name"], node_info.get("value")) if scenario_step else node_info.get("value")
        if value is not None:
            await update_node_value(server, node_info, value)
# Asynchronous input function
async def async_input(prompt: str) -> str:
    """
    Asynchronous input function to handle user input.
    Parameters:
        prompt (str): The prompt message to display.
    Returns:
        str: The user input.
    """
    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(None, input, prompt)
# Trigger events for event-specific nodes
async def trigger_event(server, event_type_node, instance, step_data, nodes):
    """
    Trigger events for event-specific nodes.
    Parameters:
        server (Server): The OPC UA server instance.
        event_type_node (Node): The event type node.
        instance (Node): The instance node.
        step_data (dict): The data for the current scenario step.
        nodes (list[dict]): A list of dictionaries containing node information from the Excel file.
    Returns:
        None
    """
    event = await server.get_event_generator(event_type_node, instance)
    event.event.Severity = 500
    event.event.Message = ua.LocalizedText("Event triggered with scenario values")
    for node_info in nodes:
        if node_info["name"] in EVENT_SPECIFIC_NODES:
            value = step_data.get(node_info["name"], False if node_info["datatype"] == "Boolean" else 0)
            variant_type = ua.VariantType.Boolean if node_info["datatype"] == "Boolean" else ua.VariantType.Int32
            setattr(event.event, node_info["name"].capitalize(), ua.Variant(value, variant_type))
            # Print the update in the desired format
            print(f"Updated {node_info['name']} (ns={node_info['ns']}; i={node_info['i']}) to {value}")
    await event.trigger()
# Main server setup and scenario handling
async def main():
    """
    Main server setup and scenario handling function.
    Returns:
        None
    """
    server = Server()
    await server.init()
    # Use environment variables for the endpoint and server name
    server_name = os.getenv("OPCUA_SERVER_NAME", "BL4R3 SDI OPC UA")
    server_port = os.getenv("OPCUA_SERVER_PORT", 4840)
    endpoint = os.getenv("OPCUA_ENDPOINT", f"opc.tcp://0.0.0.0:{server_port}/EULYNX")
    server.set_endpoint(endpoint)
    server.set_server_name(server_name)
    await server.import_xml("eulynx.generic.bl4r3.rev01.xml")
    await server.import_xml("eulynx.manufacturer.example.bl4r3.rev01.xml")
    await server.import_xml("mdm.rev01.xml")
    point_turn_event_type_node = server.get_node("ns=2;i=1123")
    # Define S10 instance object
    instance_objects = [server.get_node("ns=4;i=5057")]
    print(f"OPC UA Server '{server_name}' is running on endpoint: {endpoint}")
    """
```

```
    Define the scenarios to run in sequence
    """
    scenarios = ["default", "1", "default", "2"]
    async with server:
        while True:
            """
             Loop scenario
            """
            for scenario_choice in scenarios:
                print(f"Playing scenario: {scenario_choice}")
                nodes = load_node_info("BL4R3-rev01-Diagnostic_NodeID_List-scenario.xlsx",
                                                                    sheet_name=scenario_choice)
                if scenario_choice == "default":
                    await update_data_access_view(server, nodes)
                else:
                    scenario_steps = load_scenario_steps("BL4R3-rev01-
                                                           Diagnostic_NodeID_List-
                                                           scenario.xlsx",
                                                           sheet_name=
                                                           scenario_choice)
                    for step_data in scenario_steps:
                        await update_data_access_view(server, nodes, step_data)
                        for instance in instance_objects:
                            await trigger_event(server, point_turn_event_type_node,
                                                                        instance,
                                                                        step_data, nodes)
                        await asyncio.sleep(5)
                print(f"Scenario {scenario_choice} completed.")
                await asyncio.sleep(15)
            print("Completed all scenarios, starting over...")
if __name__ == "__main__":
    asyncio.run(main())
```

A.2.5. Detailed Configuration Files

Each of the key files within the repository used for the deployment, automation and setup of the OPC UA-based diagnostics system is provided with the complete configuration in this appendix.

.github\workflow Directory

CI/CD Pipeline Configuration (ci-cd-pipeline.yml) The CI/CD pipeline is defined using GitHub Action, this yml file is in .github/workflows directory.

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  # 1. Build Job: Installs dependencies, verifies server, and runs basic checks
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'

      - name: Install dependencies
        working-directory: Server
        run: pip install -r requirements.txt

      - name: List Files (Debug Step)
        run: ls -R  # Lists all files and directories for verification

      - name: Run server in the background
        working-directory: Server
        run: nohup python server.py &  # Starts server.py in the background

  # 2. Docker Job: Builds and pushes Docker image to GitHub Container Registry (GHCR)
  docker:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Log out of any cached Docker session (clear cache)
        run: docker logout ghcr.io || true

      - name: Log in to GitHub Container Registry
        env:
          GHCR_USERNAME: ${{ secrets.GHCR_USERNAME }}
          GHCR_TOKEN: ${{ secrets.GHCR_TOKEN }}
        run: echo "${GHCR_TOKEN}" | docker login ghcr.io -u "${GHCR_USERNAME}" --password-
            stdin
```

Code Listing A.1: ci-cd-pipeline.yml - 1/2

```
      - name: Build Docker image
        working-directory: Server
        run: docker build -t opcua-server -f Dockerfile .

      - name: Tag Docker image
        run: docker tag opcua-server ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v1

      - name: Push Docker image to GitHub Container Registry
        run: docker push ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v1

  # 3. Deploy Job: Deploys to a Kubernetes cluster using kind and Helm
  deploy:
    needs: docker
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set up kind (Kubernetes in Docker)
        run: |
          curl -Lo ./kind https://kind.sigs.k8s.io/dl/latest/kind-linux-amd64
          chmod +x ./kind
          sudo mv ./kind /usr/local/bin/kind
          kind create cluster

      - name: Configure kubectl context
        run: kubectl cluster-info

      - name: Add required Helm repositories
        run: |
          helm repo add bitnami https://charts.bitnami.com/bitnami
          helm repo add stable https://charts.helm.sh/stable
          helm repo update

      - name: Deploy with Helm
        # Adjusted path to point to the correct location for the Helm chart
        run: helm upgrade --install opcua-server ./Kubernetes/opcua-server -f ./Kubernetes
            /opcua-server/values.yaml --namespace default --debug

      - name: Verify Deployment
        run: |
          kubectl get pods -n default
          kubectl get svc -n default
```

Code Listing A.2: ci-cd-pipeline.yml - 2/2

Code Explanation

- **Pipeline Trigger:** Every push and pull request to the main branch will trigger it.
- **Build Job:** Python is set up, dependencies are installed, files are listed for verification, server.py is run in the background.
- **Docker Job:** Pushes the Docker image to GHCR.
  - **Authenticate**: Uses GitHub secrets to log in to GHCR.
  - **Build and Tag**: Takes the image and builds it, tagging it for GHCR.
  - **Push Image**: Push the image to GHCR registry.
- **Deploy Job:** Deployed to Kubernetes using kind and Helm.
  - **Set up kind**: Installs and Creates the Kubernetes cluster with Docker.
  - **Helm Deploy**: Makes installs the OPC UA server Helm chart into the Kubernetes.
  - **Verify**: Deploys a list of pods and services.

Kubernetes\opcua-server Directory

The Kubernetes configurations and Helm charts in this directory are for deploying OPC UA servers.

Helm Ignore File (.helmignore) This is an ignore file, which means to exclude files from packaging into the Helm chart.

```
# Patterns to ignore when building packages.
# This supports shell glob matching, relative path matching, and
# negation (prefixed with !). Only one pattern per line.
.DS_Store
# Common VCS dirs
.git/
.gitignore
.bzr/
.bzrignore
.hg/
.hgignore
.svn/
# Common backup files
*.swp
*.bak
*.tmp
*.orig
*~
# Various IDEs
.project
.idea/
*.tmproj
.vscode/
```

Code Listing A.3: helmignore

Code Explanation

- **General Purpose**: Excludes certain files and directories from Helm packages and thus reduces package size.
- **System Files**: Ignores OS-specific files like ‘.DS_Store‘.
- **Version Control**: Files and folders are ignored by common version control systems. (‘.git/‘, ‘.bzr/‘, ‘.hg/‘, etc.).
- **Backup Files**: Skips backup and temporary files (‘*.swp‘, ‘*.bak‘, ‘*.tmp‘, etc.).
- **IDE Configurations**: Exclude IDE-specific directories and project files (‘.idea/‘, ‘.vscode/‘, ‘.project‘).

Chart Metadata (Chart.yaml) Helm chart defines the metadata listed in the yaml file.

```yaml
apiVersion: v2
name: opcua-server
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
# Application charts are a collection of templates that can be packaged into versioned archives
# to be deployed.
# Library charts provide useful utilities or functions for the chart developer. They're included as
# a dependency of application charts to inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
# It is recommended to use it with quotes.
appVersion: "1.16.0"
```

Code Listing A.4: Chart.yaml

Code Explanation

- **apiVersion**: Indicates the format of the Helm chart version. (v2 for Helm 3 compatibility).
- **name**: Sets the chart's name as opcua-server.
- **description**: Describes the chart briefly as a Helm package for Kubernetes.
- **type**: declares the chart as application deployable (not a library).
- **version**: Defines the Helm chart version, updated with any chart/template changes (follows Semantic Versioning).
- **appVersion**: Updated for application changes, specifies the application's version. (1.16.0).

Values File (values.yaml) In this yaml file, the Helm deployment parameters are configurable.

```yaml
replicaCount: 2

updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1  # Allow a maximum of 1 pod to be unavailable during updates
    maxSurge: 1        # Allow 1 additional pod above the desired number during updates

image:
  repository: ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server
  pullPolicy: IfNotPresent
  tag: v1

service:
  type: NodePort  # Set service type to NodePort
  ports:
    - name: pointgroup-ffm
      port: 4840
      targetPort: 4840
      nodePort: 30040  # Explicit NodePort
    - name: pointgroup-muc
      port: 4841
      targetPort: 4841
      nodePort: 30041
    - name: pointgroup-ber
      port: 4842
      targetPort: 4842
      nodePort: 30042
    - name: pointgroup-ham
      port: 4843
      targetPort: 4843
      nodePort: 30043
    - name: pointgroup-dus
      port: 4844
      targetPort: 4844
      nodePort: 30044
    - name: pointgroup-emd
      port: 4845
      targetPort: 4845
      nodePort: 30045

servers:
  - name: pointgroup-ffm
    port: 4840
    endpoint: opc.tcp://0.0.0.0:4840/EULYNX
  - name: pointgroup-muc
    port: 4841
    endpoint: opc.tcp://0.0.0.0:4841/EULYNX
  - name: pointgroup-ber
    port: 4842
    endpoint: opc.tcp://0.0.0.0:4842/EULYNX
  - name: pointgroup-ham
    port: 4843
    endpoint: opc.tcp://0.0.0.0:4843/EULYNX
  - name: pointgroup-dus
    port: 4844
    endpoint: opc.tcp://0.0.0.0:4844/EULYNX
  - name: pointgroup-emd
    port: 4845
    endpoint: opc.tcp://0.0.0.0:4845/EULYNX

serviceAccount:
  create: true
  automount: true
  name: ""
```

Code Listing A.5: values.yaml - 1/2

```yaml
podAnnotations: {}
podLabels: {}

podSecurityContext: {}
securityContext: {}

ingress:
  enabled: false

resources: {}

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 100
  targetCPUUtilizationPercentage: 80

volumes: []
volumeMounts: []

nodeSelector: {}

tolerations: []

affinity: {}
```

Code Listing A.6: values.yaml - 2/2

Code Explanation

- **replicaCount**: Sets a default number of pod replicas (2).
- **updateStrategy**: Used to define rolling update strategy.
  - **type**: RollingUpdate is set for zero down time updates.
  - **maxUnavailable**: Enables up to 1 pod's unavailability for updates.
  - **maxSurge**: Allows an additional pod above the requested count to run while updating.
- **image**:
  - **repository**: Indicates the Docker image location.
  - **pullPolicy**: Pulls image if not pulled locally.
  - **tag**: Sets the image version to v1.
- **service**: Sets the service type and the ports.
  - **type**: Configures service as NodePort.
  - **ports**: Lists the ports associated to OPC UA servers.
- **servers**: Defines the endpoint for OPC UA connections of each server.
- **serviceAccount**: Sets up a service account just for the deployment.
- **podAnnotations** and **podLabels**: Custom pod annotations & labels placeholders.
- **podSecurityContext** and **securityContext**: Security settings for the pod.
- **ingress**: Configures ingress disabled by default.
- **resources**: Placeholder for resource limits and requests.
- **autoscaling**: Set the scaling parameters if enabled, disabled by default.
- **volumes** and **volumeMounts**: The definition of volumes and mounts.
- **nodeSelector**, **tolerations**, and **affinity**: Within the cluster, define rules of pod scheduling.

Templates Directory Templated .YAML for deploying resources in K8s is inside the templates directory.

Deployment Configuration (deployment.yaml) Deployment of OPC UA server pods is specified.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: opcua-servers
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: opcua-servers
  strategy: #update to support rolling update
    type: {{ .Values.updateStrategy.type }}
    rollingUpdate:
      maxUnavailable: {{ .Values.updateStrategy.rollingUpdate.maxUnavailable }}
      maxSurge: {{ .Values.updateStrategy.rollingUpdate.maxSurge }}
  template:
    metadata:
      labels:
        app: opcua-servers
    spec:
      containers:
        {{- range $server := .Values.servers }}
        - name: {{ $server.name }}
          image: "{{ $.Values.image.repository }}:{{ $.Values.image.tag }}"
          imagePullPolicy: {{ $.Values.image.pullPolicy }}
          ports:
            - containerPort: {{ $server.port }}
          env:
            - name: OPCUA_SERVER_NAME
              value: "{{ $server.name }}"
            - name: OPCUA_SERVER_PORT
              value: "{{ $server.port }}"
            - name: OPCUA_ENDPOINT
              value: "{{ $server.endpoint }}"
        {{- end }}
```

Code Listing A.7: deployment.yaml

Code Explanation

- **apiVersion** and **kind**: apps/v1 specifies the resource as a Deployment
- **metadata**:
  - **name**: Names the deployment opcua-servers.
- **spec**: Configures the Deployment.
  - **replicas**: Set the values.yaml of the number of pod replicas.
  - **selector**: Matches pods having labels app: opcua-servers.
  - **strategy**: Defines the rolling update strategy for zero downtime updates.
    - **type**: Configured to RollingUpdate as defined in values.yaml.
    - **rollingUpdate**:
      - **maxUnavailable**: Maximum number of unavailable pods during an update.
      - **maxSurge**: Describes the maximum number of pods to allow to be created during an update.
  - **template**: Defines the pod template for the Deployment.
    - **metadata - labels**: Labels the pod with ‘app: opcua-servers‘.
    - **spec - containers**:
      - Iterate on each server from values.yaml.

- **name**: Matches the name of server to each container name.
- **image**: Identifies the image, and tag, for each container.
- **imagePullPolicy**: Determines which images will be pulled based on controls as defined in values.yaml.
- **ports**: Maps each service's container port into a public port.
- **env**: For each container, defines environment variables.
- **OPCUA_SERVER_NAME**: Name of the OPC UA server.
- **OPCUA_SERVER_PORT**: Port assigned to the server.
- **OPCUA_ENDPOINT**: Server endpoint address.

Service Configuration (service.yaml) Exposes the OPC UA servers with NodePort service.

```
apiVersion: v1
kind: Service
metadata:
  name: opcua-servers-service
spec:
  selector:
    app: opcua-servers
  ports:
    {{- range $server := .Values.service.ports }}
    - name: {{ $server.name }}
      protocol: TCP
      port: {{ $server.port }}
      targetPort: {{ $server.targetPort }}
      nodePort: {{ $server.nodePort }}
    {{- end }}
  type: {{ .Values.service.type }}  # This will be NodePort
```

Code Listing A.8: service.yaml

Code Explanation

- **apiVersion** and **kind**: Marks this resource as Service using v1.
- **metadata**:
  - **name**: Names the service opcua-servers-service.
- **spec**: Configures the Service.
  - **selector**: Labeled on the pods with app: opcua-servers to direct traffic to one deployment or another.
  - **ports**: Goes through each port specified by the service in values.yaml.
    - **name**: Each service port sets a name.
    - **protocol**: TCP is used for communication.
    - **port**: Exposes the service port.
    - **targetPort**: Forwards traffic to target port of the container.
    - **nodePort**: External NodePort will be mapped for a outside cluster access.
  - **type**: Sets the Service type to valuesyaml (e.g., NodePort).

Horizontal Pod Autoscaling (HPA) Configuration (hpa.yaml) This yaml file is used to setting up horizontal pod autoscaling.

```
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "opcua-server.fullname" . }}
  labels:
    {{- include "opcua-server.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "opcua-server.fullname" . }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
    {{- if .Values.autoscaling.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetCPUUtilizationPercentage }}
    {{- end }}
    {{- if .Values.autoscaling.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetMemoryUtilizationPercentage }}
    {{- end }}
{{- end }}
```

Code Listing A.9: hpa.yaml

Code Explanation

- **Conditionally Enabled**: The autoscaler will be used only when the autoscalingenabled has set true in the system at values.yaml.
- **apiVersion** and **kind**: Defining it as autoscaling/v2, HorizontalPodAutoscaler relates to the resource.
- **metadata**:
  - **name**: Changes the HPA name to the full name of opcua-server.
  - **labels**: Uses standard labels in order to be consistent.
- **spec**: Main HPA configuration.
  - **scaleTargetRef**: Associates the autoscaler to the opcua-server Deployment.
  - **minReplicas** and **maxReplicas**: Validates the minimum and the maximum value of pod replicas.
  - **metrics**: Scales as per resource usage defined while Identifying triggers for scaling.
    - **CPU Utilization**: Scales based on average of CPU usage if defined.
    - **Memory Utilization**: Average use of memory if scales are specified.

Test Connection Configuration (test-connection.yaml) This yaml file is used to check the status of the deployed OPC UA server, and whether it is reachable.

```
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "opcua-server.fullname" . }}-test-connection"
  labels:
    {{- include "opcua-server.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['{{ include "opcua-server.fullname" . }}:{{ .Values.service.port }}']
  restartPolicy: Never
```

Code Listing A.10: test-connection.yaml

Code Explanation

- **apiVersion** and **kind**: Defined this resource Pod using v1.
- **metadata**:
  - **name**: Sets the Pod name as ‘opcua-server‘ followed by ‘-test-connection‘.
  - **labels**: Applicable to assign standard labels for identification purposes.
  - **annotations**: To set this Pod as a test hook, using helm.sh/hook: test which makes this Helm test resource.
- **spec**: Specifies the parameters of the test Pod.
  - **containers**:
    - **name**: Names the container ‘wget‘.
    - **image**: Runs a light-weight command using busybox image as the base.
    - **command** and **args**: Executes wget with the argument set to probe the connection to the service in the given port.
  - **restartPolicy**: Change the setting to Never to avoid the test Pod from automatically restarting when the test is done.

Ingress Configuration (ingress.yaml) This yaml file deals with access through ingress controllers.

```
{{- if .Values.ingress.enabled -}}
{{- $fullName := include "opcua-server.fullname" . -}}
{{- $svcPort := .Values.service.port -}}
{{- if and .Values.ingress.className (not (semverCompare ">=1.18-0" .Capabilities.
    KubeVersion.GitVersion)) }}
  {{- if not (hasKey .Values.ingress.annotations "kubernetes.io/ingress.class") }}
  {{- $_ := set .Values.ingress.annotations "kubernetes.io/ingress.class" .Values.ingress.
      className}}
  {{- end }}
{{- end }}
{{- if semverCompare ">=1.19-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1
{{- else if semverCompare ">=1.14-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1beta1
{{- else -}}
apiVersion: extensions/v1beta1
{{- end }}
kind: Ingress
metadata:
  name: {{ $fullName }}
  labels:
    {{- include "opcua-server.labels" . | nindent 4 }}
  {{- with .Values.ingress.annotations }}
  annotations:
    {{- toYaml . | nindent 4 }}
  {{- end }}
spec:
  {{- if and .Values.ingress.className (semverCompare ">=1.18-0" .Capabilities.KubeVersion
      .GitVersion) }}
  ingressClassName: {{ .Values.ingress.className }}
  {{- end }}
  {{- if .Values.ingress.tls }}
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ .path }}
            {{- if and .pathType (semverCompare ">=1.18-0" $.Capabilities.KubeVersion.
                GitVersion) }}
            pathType: {{ .pathType }}
            {{- end }}
            backend:
              {{- if semverCompare ">=1.19-0" $.Capabilities.KubeVersion.GitVersion }}
              service:
                name: {{ $fullName }}
                port:
                  number: {{ $svcPort }}
              {{- else }}
              serviceName: {{ $fullName }}
              servicePort: {{ $svcPort }}
              {{- end }}
          {{- end }}
    {{- end }}
{{- end }}
```

Code Listing A.11: ingress.yaml

Code Explanation

- **Conditionally Enabled**: The Ingress is only created if ingress.enabled is set to true in values.yaml.
- **apiVersion** and **kind**: Sets up the Ingress resource depending on the compatibility of Kubernetes versions.
- **metadata**:
  - **name**: Specifies the Ingress name to be the fullname of opcua-server.
  - **labels**: Implements standard identification attributes.
  - **annotations**: Defines annotations, including optional ingress.class.
- **spec**: Main Ingress configuration.
  - **ingressClassName**: Sets the Ingress class if defined in values.yaml.
  - **tls**: TLS is enabled for specific hosts if configured in values.yaml file.
  - **rules**: Specifies the routing of each host.
    - **host**: Describes each host for the purpose of routing the traffic.
    - **paths**: Describes the path and backend on which the request should be forwarded.
      - **pathType**: Decides the type of path matching, for instance Prefix.
      - **backend**: Implies directing the traffic to the service for a particular port.

Server Directory

The source code and dependencies of OPC UA server's core components and configurations belong to the Server directory.

Docker Configuration (Dockerfile) The OPC UA server container is built by following this Dockerfile.

```
# Use an official Python runtime as a parent image
FROM python:3.12-slim

# Set the working directory in the container
WORKDIR /app

# Install system dependencies with retry mechanism
RUN apt-get update || apt-get update && apt-get install -y \
    gcc \
    libffi-dev \
    libssl-dev \
    make \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Copy the current directory contents into the container at /app
COPY . /app/

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 4840 available to the world outside this container
EXPOSE 4840

# Run server.py when the container launches
CMD ["python", "server.py"]
```

Code Listing A.12: Dockerfile

Code Explanation

- **Base Image**: python:3.12-slim is the lightweight parent image to run Python applications.
- **Working Directory**: Changes the working directory inside the container to /app.
- **System Dependencies**: Installs essential packages (‘gcc‘, ‘libffi-dev‘, ‘libssl-dev‘, ‘make‘) for building Python packages with a retry mechanism.

  Installs packages to build Python packages with retry mechanism (gcc, libffi-dev, libssl-dev, make).
- **Copy Application Files**: Copies /app in the container from the contents in the current directory.
- **Python Dependencies**: Saves space by installing Python packages from requirements.txt without caching.
- **Expose Port**: Makes port 4840 accessible for external connections to this port.
- **Start Command**: Runs server.py on start of container.

Docker Registry Access (access_registry_docker.txt) This file contains credentials needed to access Docker registry.

```
# Username for Docker registry
# Use the GitHub Secrets variable for the username in CI/CD workflows, e.g., ${{ secrets.
    GHCR_USERNAME }}

# Personal Access Token (for GitHub Container Registry)
# Use the GitHub Secrets variable for the token in CI/CD workflows, e.g., ${{ secrets.
    GHCR_TOKEN }}

# Docker login command (use in CI/CD pipeline)
# Example command using GitHub Secrets in workflows:
# echo "${{ secrets.GHCR_TOKEN }}" | docker login ghcr.io -u "${{ secrets.GHCR_USERNAME
    }}" --password-stdin

# Command to run the Docker container
docker run --publish 4840:4840 ghcr.io/loukjeab/eulynx_sdi_opc_ua_bl4r3/server:v3
```

Code Listing A.13: access_registry_docker.txt

Code Explanation

- **Username for Docker Registry**: The registry username is stored as GHCR_USERNAME in CI/CD workflows and uses GitHub Secrets for this.
- **Personal Access Token**: Uses GHCR_TOKEN from GitHub Secrets to secure registry access.
- **Docker Login Command**: Authenticates with GitHub Container Registry (GHCR) in CI/CD workflows by using the stored secrets to log into that.
- **Run Docker Container**: Supports running the container with a sample command which will map external port 4840 to internal port 4840, allowing application access..

Python Requirements File (requirements.txt) All required Python dependencies for this OPC UA server are listed inside this txt file.

```
asyncua==1.1.0
pandas==2.2.2
openpyxl==3.1.5
pika==1.3.2
python-qpid-proton==0.39.0
pycryptodome
```

Code Listing A.14: requirements.txt

Diagnostic Scenario Excel File (BL4R3-rev01-Diagnostic_NodeID_List-scenario.xlsx) The OPC UA server uses this Excel file for scenario simulation, and its full explanation is explained in Appendix A.2.3 - Detailed Variable States for Scenario Simulation.

OPC UA Information Model XML Files The OPC UA Information Model for the server is described by the following XML files.

- eulynx.generic.bl4r3.rev01.xml
- eulynx.manufacturer.example.bl4r3.rev01.xml
- mdm.rev01.xml

Main Server Script (server.py) This is the main Python script implementing the OPC UA server, which has been already fully explain in Section 5.4 - OPC UA Server Implementation in Chapter 5 - Implementation.

Git Ignore File (.gitignore) The ignore file describes files and directories to be ignored from version control.

```
# Editors
.vscode/
.idea/

# Vagrant
.vagrant/

# Mac/OSX
.DS_Store

# Windows
Thumbs.db

# Source for the following rules: https://raw.githubusercontent.com/github/gitignore/
    master/Python.gitignore
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt
```

Code Listing A.15: .gitignore - 1/3

```
# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
.hypothesis/
.pytest_cache/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
.python-version

# celery beat schedule file
celerybeat-schedule

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject
```

Code Listing A.16: .gitignore - 2/3

```
# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# files
resources/backup_file.json
```

Code Listing A.17: .gitignore - 3/3

Code Explanation

- **Editor Files**: Exclude project files for common editors settings (e.g., .vscode/, .idea/).
- **Vagrant and OS Files**: Ignores OS-specific files (.DS_Store, Thumbs.db) and Vagrant files (.vagrant/).
- **Python Bytecode and DLLs**: Bytes and compiled files are not versioned to prevent bytecode. (__pycache__/, *.py[cod], *.so).
- **Distribution and Packaging**: Excludes files and folders containing Python packaging concerns. (build/, dist/, *.egg-info/).
- **PyInstaller Files**: Does not exclude files generated by PyInstaller (*.manifest, *.spec).
- **Logs and Test Coverage**: Ignores log files, test reports and coverage reports. (e.g., .tox/, .coverage, *.log).
- **Project-Specific Files**: Django, Flask, Sphinx and Scrapy files excluded. (db.sqlite3, .scrapy, docs/_build/).
- **Environment Files**: Skips virtual environment folders and files specific to an environment. (.env, env/, .venv/).
- **Notebook and Python Settings**: Excludes Jupyter, IPython, pyenv files. (.ipynb_checkpoints/, .python-version).
- **IDE Project Settings**: Excludes project settings for Spyder, Rope. (.spyderproject, .ropeproject).
- **Miscellaneous Caches and Temp Files**: Bygones, mypy cache, backups and more config files are ignored. (.mypy_cache/, resources/backup_file.json).

A.3. Test and Validation

A.3.1. OPC UA Information Model Mapping Verification

List of ObjectTypes

This appendix gives a comprehensive overview of the 111 ObjectTypes created for the OPC UA Information Model. In addition, each ObjectType has been demonstrated to follow the EU-Rail and EULYNX Interface Specification SDI. The Table A.7 - A.11 below lists ObjectType, the attributes of the ObjectType and the relevant identifiers for the ObjectType.

No.	Name	Abstr.	Reference Type	Target	NodeId
1	SubsystemType	Yes	IsImplementedBy	EquipmentType	ns=1;i=1002
2	FieldElementType	Yes	-	-	ns=1;i=1008
3	InterfaceType	Yes	-	-	ns=1;i=1150
4	SDI_Type	No	-	-	ns=1;i=1153
5	SDI_IO_Type	No	GeneratesEvent	LogEventInterface	ns=1;i=1158
6	SDI_LS_Type	No	GeneratesEvent	LogEventInterface	ns=1;i=1157
7	SDI_IC_Type	No	GeneratesEvent	LogEventInterface	ns=1;i=1159
8	SDI_P_Type	No	GeneratesEvent	LogEventInterface	ns=1;i=1155
9	SDI_TDS_Type	No	GeneratesEvent	LogEventInterface	ns=1;i=1156
10	SCI_PDI_Type	No	-	-	ns=1;i=1152
11	SCP_Type	No	-	-	ns=1;i=1250
12	SCI_IO_PDI_Type	No	GeneratesEvent	LogEventInterfacePdiEventType LogEventInterfaceType	ns=1;i=1164
13	SCI_LS_PDI_Type	No	GeneratesEvent	LogEventInterfacePdiEventType LogEventInterfaceType	ns=1;i=1163
14	SCI_IC_PDI_Type	No	GeneratesEvent	LogEventInterfacePdiEventType LogEventInterfaceType	ns=1;i=1165
15	SCI_P_PDI_Type	No	GeneratesEvent	LogEventInterfacePdiEventType LogEventInterfaceType	ns=1;i=1161
16	SCI_TDS_PDI_Type	No	GeneratesEvent	LogEventInterfacePdiEventType LogEventInterfaceType	ns=1;i=1167
17	SSI_Type	No	-	-	ns=1;i=1241
18	SMI_Type	No	-	-	ns=1;i=1050
19	ConfigurationItemType	No	-	-	ns=1;i=1052
20	TransportChannelType	Yes	-	-	ns=1;i=1151
21	TransportChannelRastaType	No	-	-	ns=1;i=1247
22	TransportChannelOpcuaType	No	-	-	ns=1;i=1244
23	OpcUaServerType	No	ServesInterface	SDI_Type & SMI_Type	ns=1;i=1228

Table A.7.: ObjectTypes Summary in EULYNX SDI OPC UA Model - Generic Namespace - 1/5

No.	Name	Abstr.	Reference Type	Target	NodeId
24	TlsCertificateType	No	IsTlsCertificateOf	TransportChannelType	ns=1;i=1235
25	EquipmentType	No	IsPartOfRedundancyGroup	RedundancyStatus	ns=1;i=1003
26	ControllerType	No	IsPartOfRedundancyGroup	RedundancyStatus	ns=1;i=1006
27	PhysicalNetworkInterfaceType	No	IsPartOfRedundancyGroup	RedundancyStatus	ns=1;i=1004
28	NetworkConfigurationType	No	-	-	ns=1;i=1128
29	StorageMediumType	Yes	IsPartOfRedundancyGroup	RedundancyStatus	ns=1;i=1011
30	StorageMedium FlashMemoryType	No	-	-	ns=1;i=1013
31	PowerSupplyType	No	-	-	ns=1;i=1048
32	PhysicalOutputType	No	-	-	ns=1;i=1017
33	PhysicalDigitalOutputType	No	-	-	ns=1;i=1020
34	PhysicalAnalogOutputType	No	-	-	ns=1;i=1019
35	PhysicalSeparatedOutputType	No	-	-	ns=1;i=1054
36	PhysicalInputType	No	-	-	ns=1;i=1014
37	PhysicalDigitalInputType	No	-	-	ns=1;i=1016
38	PhysicalAnalogInputType	No	-	-	ns=1;i=1015
39	PhysicalSeparatedInputType	No	-	-	ns=1;i=1063
40	InputDeviceType	No	-	-	ns=1;i=1079
41	InputButtonType	No	-	-	ns=1;i=1064
42	InputSwitchType	No	-	-	ns=1;i=1093
43	AuxillaryInputType	No	-	-	ns=1;i=1076
44	RedundancyGroupType	No	-	-	ns=1;i=1201
45	RedundancyStatusType	No	IsPartOfRedundancyGroup	RedundancyGroupType	ns=1;i=1021
46	LogEventType	Yes	-	-	ns=1;i=1216
47	LogEventInterfaceType	No	-	-	ns=1;i=1219
48	LogEventInterface PdiEventType	No	-	-	ns=1;i=1253
49	LogEventSubsystemType	No	-	-	ns=1;i=1217
50	MotorTurnDataType	Yes	-	-	ns=1;i=2002
51	MotorTurnData_1AC_Type	No	-	-	ns=1;i=2006
52	MotorTurnData_1AC_ ActiveCurrent Inductive Compensation_Type	No	-	-	ns=1;i=2003
53	MotorTurnData_1AC_ ActiveCurrent PhaseAngle Compensation_Type	No	-	-	ns=1;i=2008

Table A.8.: ObjectTypes Summary in EULYNX SDI OPC UA Model - Generic Namespace - 2/5

No.	Name	Abstr.	ReferenceType	Target	NodeId
54	MotorTurnData_AC_ Power_Type	No	-	-	ns=1;i=2004
55	MotorTurnData_3AC_Type	No	-	-	ns=1;i=2005
56	MotorTurnData_3AC_ Power_Type	No	-	-	ns=1;i=2011
57	MotorTurnData_ Hydraulic_ Type	No	-	-	ns=1;i=2014
58	PointType	No	GeneratesEvent ConnectsTo	LogEventSubsystemType PointTurnEventType SCI_P_PDI_Type SDI_P_Type	ns=1;i=1095
59	PointMachineType	No	-	-	ns=1;i=1091
60	PointTurnEventType	No	-	-	ns=1;i=1123
61	DetectionCircuitLeftType	No	HasPhysicalReferenceChannel	PhysicalAnalogInputType	ns=1;i=1090
62	DetectionCircuitRightType	No	HasPhysicalReferenceChannel	PhysicalAnalogInputType	ns=1;i=1092
63	GenericIOType	No	GeneratesEvent ConnectsTo	LogEventSubsystemType SCI_IO_PDI_Type SDI_IO_Type	ns=1;i=1073
64	PhysicalChannel ConnectionType	No	-	-	ns=1;i=1094
65	LogicalChannelType	Yes	-	-	ns=1;i=1070
66	LogicalInputChannelType	Yes	-	-	ns=1;i=1071
67	LogicalInputChannel SingleChannelType	No	HasPhysicalReferenceChannel	PhysicalInputType	ns=1;i=1190
68	LogicalInputChannel TwoChannelType	No	-	-	ns=1;i=1189
69	LogicalOutputChannelType	Yes	-	-	ns=1;i=1072
70	LogicalOutputChannel SingleChannelType	No	HasPhysicalReferenceChannel	PhysicalOutputType	ns=1;i=1191
71	LogicalOutputChannel TwoChannelType	No	-	-	ns=1;i=1192
72	LightSignalType	No	GeneratesEvent ConnectsTo	LogEventSubsystemType SCI_LS_PDI_Type SDI_LS_Type	ns=1;i=1080
73	LightSignalAdjacent OutputChannelType	No	HasPhysicalReferenceChannel	PhysicalOutputType	ns=1;i=1018
74	LightSignalAdjacent TrainControl ElementsType	No	-	-	ns=1;i=1009
75	LightPointType	Yes	-	-	ns=1;i=1069
76	LightPointMatrixType	No	-	-	ns=1;i=1121

Table A.9.: ObjectTypes Summary in EULYNX SDI OPC UA Model - Generic Namespace - 3/5

No.	Name	Abstr.	ReferenceType	Target	NodeId
77	LightPointSingle FilamentType	No	-	-	ns=1;i=1102
78	LightPointSingleLampType	No	-	-	ns=1;i=1125
79	LightPointSingleLEDType	No	-	-	ns=1;i=1126
80	IndicatorType	No	HasPhysicalReferenceChannel	PhysicalOutputType	ns=1;i=1124
81	DetectionDeviceType	Yes	-	-	ns=1;i=1041
82	DetectionDevice PunctiformType	Yes	-	-	ns=1;i=1043
83	DetectionDevice CountingHeadType	Yes	-	-	ns=1;i=1074
84	DetectionDevicePunctiform CountingHeadMagneticType	No	-	-	ns=1;i=1193
85	DetectionDevicePunctiform PresenceSensorType	No	-	-	ns=1;i=1127
86	DetectionDevicePunctiform PresenceDetectorMagneticType	No	-	-	ns=1;i=1045
87	DetectionDevicePunctiform PresenceDetectorLight BarrierType	No	-	-	ns=1;i=1010
88	DetectionDeviceLinearType	Yes	-	-	ns=1;i=1042
89	DetectionDevice LinearLoopType	No	-	-	ns=1;i=1046
90	DetectionDevice LinearTrack CircuitType	No	-	-	ns=1;i=1012
91	TrainDetectionSystemType	No	GeneratesEvent ConnectsTo	LogEventSubsystemType SCI_TDS_PDI_Type SDI_TDS_Type	ns=1;i=1077
92	TvpSectionType	Yes	-	-	ns=1;i=1075
93	TvpSectionAxleCounterType	No	GeneratesEvent	LogEventTvpSectionAxle CounterCommandType LogEventTvpSectionAxle CounterSweepingFailsType	ns=1;i=1022
94	TvpSection TrackCircuitType	No	GeneratesEvent	LogEventDetection DeviceLinearTrack CircuitOccupancyType	ns=1;i=1023
95	LogEventTvpSectionAxle CounterCommandType	No	-	-	ns=1;i=1049
96	LogEventTvpSectionAxle CounterSweepingFailsType	No	-	-	ns=1;i=1053

Table A.10.: ObjectTypes Summary in EULYNX SDI OPC UA Model - Generic Namespace - 4/5

No.	Name	Abstr.	Reference Type	Target	NodeId
97	LogEventDetectionDeviceLinear TrackCircuitOccupancyType	No	-	-	ns=1;i=1129
98	TrainDetectionPointType	No	-	-	ns=1;i=1062
99	LevelCrossingType	No	GeneratesEvent ConnectsTo	LogEventSubsystemType LogEventLcaRequestType LogEventLocalHandoverType SCI_IC_PDI_Type SDI_IC_Type	ns=1;i=1130
100	LevelCrossingProtection FacilityType	No	-	-	ns=1;i=1131
101	BarrierType	No	GeneratesEvent	BarrierTurnEventType	ns=1;i=1160
102	ObstacleDetectorType	No	-	-	ns=1;i=1168
103	ObstacleDetectorRadarType	No	-	-	ns=1;i=1174
104	ObstacleDetectorOtherType	No	-	-	ns=1;i=1177
105	DetectionElementType	No	-	-	ns=1;i=1184
106	RoadSignalsType	No	-	-	ns=1;i=1181
107	WarningBellType	No	-	-	ns=1;i=1187
108	WarningLampType	No	-	-	ns=1;i=1149
109	BarrierTurnEventType	No	-	-	ns=1;i=1167
110	LogEventLocalRequestType	No	-	-	ns=1;i=1179
111	LogEventLocalHandoverType	No	-	-	ns=1;i=1175

Table A.11.: ObjectTypes Summary in EULYNX SDI OPC UA Model - Generic Namespace - 5/5

List of Enumeration DataTypes

86 Enumeration DataType were summarized comprehensively in this appendix that each of them has been verified for compliance with EU-Rail and EULYNX Interface Specification SDI. Each Enumemation DataType and its inventory is listed in the following Table A.12 to A.14

No.	SDI-XX Cetagory	Name	NodeId
1	SDI-Generic	ActivationState	ns=1;i=3099
2	SDI-Generic	BasicDataReadable	ns=1;i=3097
3	SDI-Generic	ConnectionStatus	ns=1;i=3098
4	SDI-Generic	ControllerOperationStatus	ns=1;i=3082
5	SDI-Generic	ControllerResetReason	ns=1;i=3083
6	SDI-Generic	CoolingFanStatus	ns=1;i=3085
7	SDI-Generic	CpuHealthStatus	ns=1;i=3086
8	SDI-Generic	CpuLoadStatus	ns=1;i=3094
9	SDI-Generic	EquipmentReplaceabilityStatus	ns=1;i=3073
10	SDI-Generic	FieldElementOperationStatus	ns=1;i=3072
11	SDI-Generic	FluidLevelStatus	ns=1;i=3102
12	SDI-Generic	HighLow	ns=1;i=3081
13	SDI-Generic	InputSwitchPosition	ns=1;i=3112
14	SDI-Generic	InputValue	ns=1;i=3110
15	SDI-Generic	LogSeverityLevel	ns=1;i=3061
16	SDI-Generic	MultiStateDiscreteTypeSupplier	ns=1;i=3070
17	SDI-Generic	OutputValue	ns=1;i=3016
18	SDI-Generic	PdiError	ns=1;i=3096
19	SDI-Generic	PdiConnectionStatus	ns=1;i=3080
20	SDI-Generic	PdiEventNotification	ns=1;i=3084
21	SDI-Generic	PhysicalNetworkInterfaceOperationalStatus	ns=1;i=3074
22	SDI-Generic	PreloadState	ns=1;i=3087
23	SDI-Generic	RamHealthStatus	ns=1;i=3093
24	SDI-Generic	ScpConnectionStatus	ns=1;i=3067
25	SDI-Generic	StatusTechnical	ns=1;i=3088
26	SDI-Generic	TemperatureStatus	ns=1;i=3062
27	SDI-Generic	TlsStatus	ns=1;i=3068
28	SDI-Generic	TransportChannelRastaStatus	ns=1;i=3077
29	SDI-Generic	TransportLayer	ns=1;i=3076
30	SDI-Generic	VoltageStatus	ns=1;i=3075
31	SDI-Generic	WearStatus	ns=1;i=3103

Table A.12.: Enumeration DataType Summary in EULYNX SDI OPC UA Model SDI-Generic - 1/3

No.	SDI-XX Cetagory	Name	NodeId
32	SDI-P	PointAbleToMoveStatus	ns=1;i=3098
33	SDI-P	PointCommandedPosition	ns=1;i=3105
34	SDI-P	PointDriveCutOffPrinciple	ns=1;i=3108
35	SDI-P	PointMachine_Type	ns=1;i=3111
36	SDI-P	PointMovementStatus	ns=1;i=3078
37	SDI-P	PointPosition	ns=1;i=3084
38	SDI-P	PointPositionDegraded	ns=1;i=3090
39	SDI-P	PointTurnFailureReason	ns=1;i=3098
40	SDI-IO	DutyRatioFixedConfiguration	ns=1;i=3080
41	SDI-IO	FlashingPeriodFixedConfiguration	ns=1;i=3079
42	SDI-IO	LogicalInputValue	ns=1;i=3065
43	SDI-IO	LogicalOutputValue	ns=1;i=3068
44	SDI-IO	OutputDisturbanceStatus	ns=1;i=3071
45	SDI-IO	PhysicalChannelTwoChannelsType	ns=1;i=3114
46	SDI-IO	ValenceType	ns=1;i=3092
47	SDI-LS	FilamentStatus	ns=1;i=3091
48	SDI-LS	IndicatorStatus	ns=1;i=3089
49	SDI-LS	InterfaceConnectionStatus	ns=1;i=3104
50	SDI-LS	LampWireRole	ns=1;i=3048
51	SDI-LS	LightSignalAdjacentTrainControlSystem	ns=1;i=3101
52	SDI-LS	LogicalLightPointStatus	ns=1;i=3109
53	SDI-LS	NightDay	ns=1;i=3038

Table A.13.: Enumeration DataType Summary in EULYNX SDI OPC UA Model SDI-P, IO, and LS - 2/3

No.	SDI-XX Cetagory	Name	NodeId
55	SDI-TDS	ChangeTrigger	ns=1;i=3123
56	SDI-TDS	CircuitConnectionStatus	ns=1;i=3128
57	SDI-TDS	CommandTriggeredByRole	ns=1;i=3122
58	SDI-TDS	DirectionOfPassing	ns=1;i=3135
59	SDI-TDS	DriftStatus	ns=1;i=3095
60	SDI-TDS	OccupancyStatus	ns=1;i=3126
61	SDI-TDS	PassingStatus	ns=1;i=3138
62	SDI-TDS	QualityOfTransmission	ns=1;i=3141
63	SDI-TDS	StatusReceiver	ns=1;i=3144
64	SDI-TDS	StatusTransmitter	ns=1;i=3143
65	SDI-TDS	SweepingFailureReason	ns=1;i=3119
66	SDI-TDS	SweepingStatus	ns=1;i=3150
67	SDI-TDS	TvpSectionCommandType	ns=1;i=3103
68	SDI-TDS	TvpSectionPomStatus	ns=1;i=3104
69	SDI-TDS	TvpSectionPowerSupplyStatus	ns=1;i=3129
70	SDI-TDS	TvpSectionStatusTrackCircuit	ns=1;i=3113
71	SDI-TDS	UnitReceiverSignal	ns=1;i=3153
72	SDI-TDS	UnitReceiverThreshold	ns=1;i=3154
73	SDI-TDS	UnitTransmitterSignal	ns=1;i=3159
74	SDI-LC	BarrierBoomLightsStatus	ns=1;i=3118
75	SDI-LC	BarrierExpectedPosition	ns=1;i=3121
76	SDI-LC	BarrierStatus	ns=1;i=3130
77	SDI-LC	BarrierTurnFailureReason	ns=1;i=3136
78	SDI-LC	FunctionalStatus	ns=1;i=3142
79	SDI-LC	HandoverCommandStatus	ns=1;i=3145
80	SDI-LC	HandoverReactionStatus	ns=1;i=3149
81	SDI-LC	ObstacleDetectorStatus	ns=1;i=3154
82	SDI-LC	ProtectionFacilityStatus	ns=1;i=3152
83	SDI-LC	StatusDetectionElement	ns=1;i=3160
84	SDI-LC	TimeoutStatus	ns=1;i=3165
85	SDI-LC	WarningBellStatus	ns=1;i=3158
86	SDI-LC	WarningLampStatus	ns=1;i=3171

Table A.14.: Enumeration DataType Summary in EULYNX SDI OPC UA Model SDI-TDS, and LC - 3/3

List of EventType

This appendix provides a summary of all 11 EventType, each of which has been verified to meet SDI, EU-Rail and EULYNX Interface Specification. The Table A.15 below shows the EventType and its Parent.

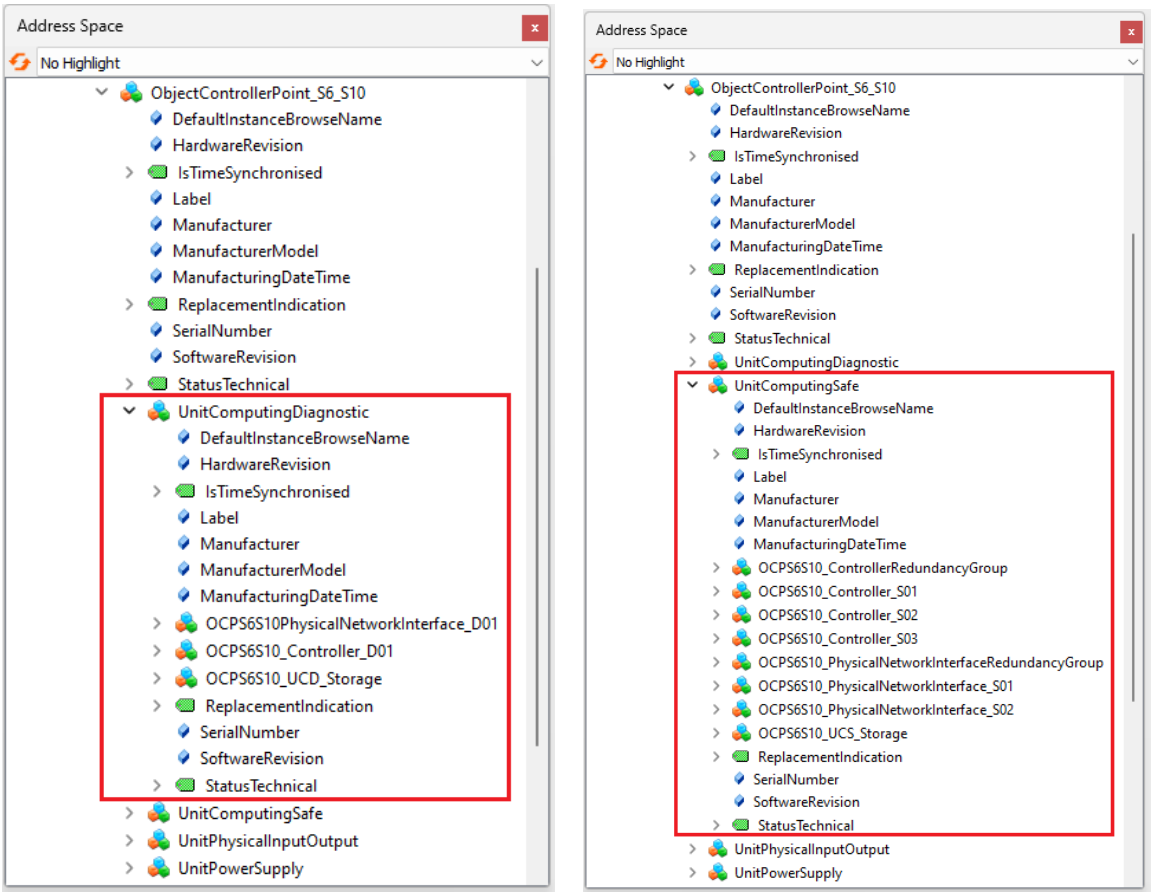
No.	SDI-XX Cetagory	Name	Parent	NodeId
1	SDI-Generic	LogEventType	BaseEventType	ns=1;i=1216
2	SDI-Generic	LogEventInterfaceType	LogEventType	ns=1;i=1219
3	SDI-Generic	LogEventInterfacePdiEventType	LogEventInterfaceType	ns=1;i=1253
4	SDI-Generic	LogEventSubsystemType	LogEventType	ns=1;i=1217
5	SDI-P	PointTurnEventType	LogEventSubsystemType	ns=1;i=1123
6	SDI-TDS	LogEventDetectionDevice LinearTrackCircuitOccupancyType	LogEventSubsystemType	ns=1;i=1129
7	SDI-TDS	LogEventTvpSection AxleCounterCommandType	LogEventSubsystemType	ns=1;i=1049
8	SDI-TDS	LogEventTvpSection AxleCounterSweepingFailsType	LogEventSubsystemType	ns=1;i=1053
9	SDI-LC	BarrierTurnEventType	LogEventSubsystemType	ns=1;i=1167
10	SDI-LC	LogEventLocalHandoverType	LogEventSubsystemType	ns=1;i=1175
11	SDI-LC	LogEventLocalRequestType	LogEventSubsystemType	ns=1;i=1179

Table A.15.: EventType Summary in EULYNX SDI OPC UA Model - Generic Namespace

A.3.2. Object Instantiation and Scenario Verification

PointEquipmentSet Unit Details

Detailed Figure of each unit in the PointEquipmentSet (UnitComputingDiagnostic, UnitComputingSafe, UnitPhysicalInputOutput, and UnitPowerSupply) are located here to confirm the attributes and structure per planned instantiation.

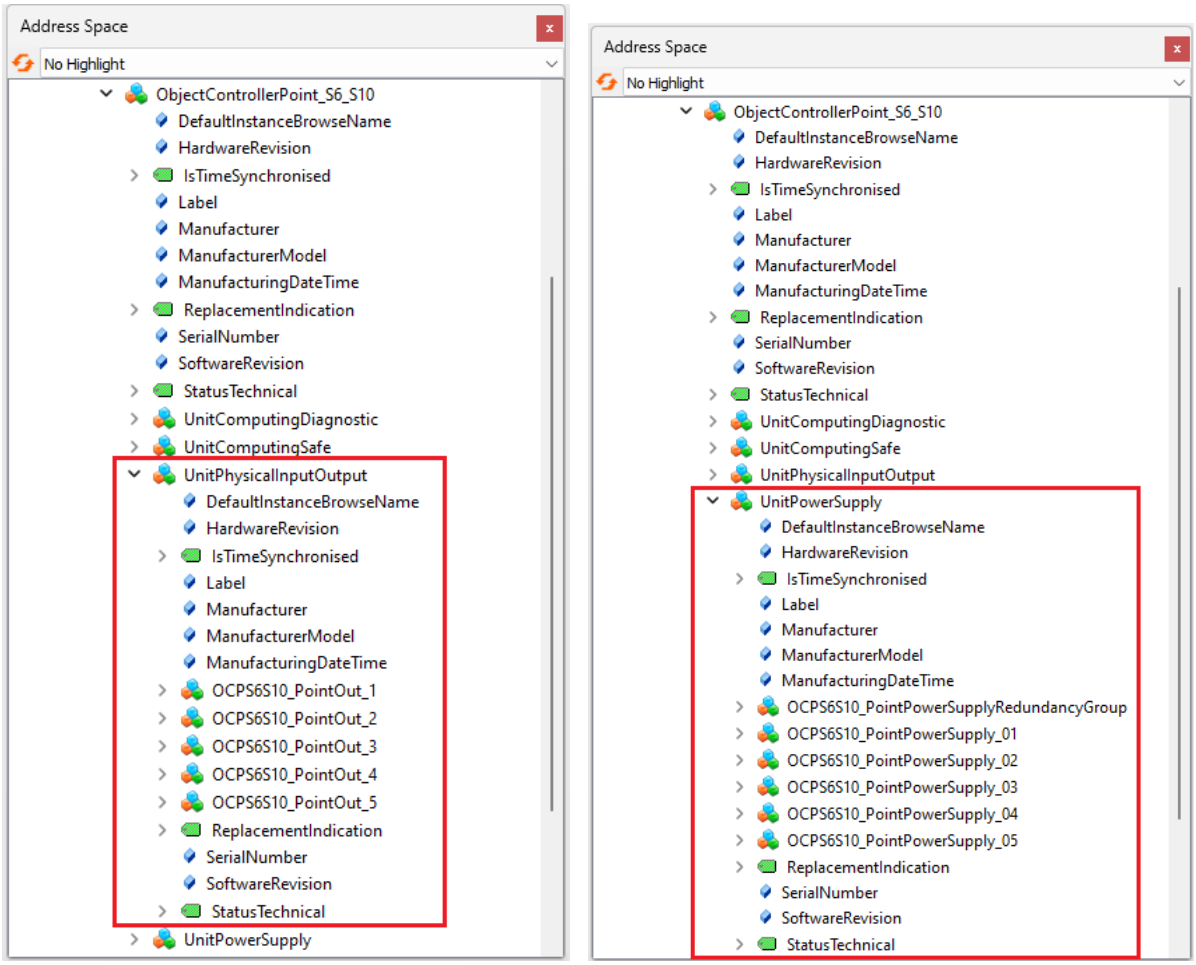


(a) UnitComputingDiagnostic in ObjectController S6-S10 (b) UnitComputingSafe in ObjectController S6-S10

Figure A.14.: UnitComputingDiagnostic and UnitComputingSafe in ObjectController S6-S10, own illustration

The UnitComputingDiagnostic and UnitComputingSafe Figure are shown above. Each of these units will be configured according to their implementation planned on PointEquipmentSet and will include essential diagnostic and safety functions.

Below shows the Figure of UnitPhysicalInputOutput and UnitPowerSupply components which are important for interfacing with physical I/O devices and keeping unit power stable throughout the ObjectController system respectively.



(a) UnitPhysicalInputOutput in ObjectController S6-S10 (b) UnitPowerSupply in ObjectController S6-S10

Figure A.15.: UnitPhysicalInputOutput and UnitPowerSupply in ObjectController S6-S10, own illustration

Finally, these detailed visualizations of the PointEquipmentSet units validates that each of this component has been instanced with the appropriate structure and attributes as designed for implementation. This is verification to ensure that the system setup is capable to the required diagnostic and operational capabilities.

Scenario 1: Critical Failure at Point S10

Data Access View UaExpert's Data Access View supports multple server monitoring (pointgroup-ffm, pointgroup-muc, pointgroup-ber, pointgroup-dus, pointgroup-emd, and pointgroup-ham) but this appendix will cover only pointgroup-emd in the critical failure scenario at Point S10. Figure A.16 - A.24 display real time server response for aggregate functions in AggregateAbleToMoveStatus, LastCommandedPosition, Position, etc., as configured in the Excel, with timestamps indicating that the data was processed immediately.

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:40:38.723 AM	9:40:38.723 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	0 (Unknown)	Int32	9:41:19.061 AM	9:41:19.061 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	0 (Unknown)	Int32	9:41:19.062 AM	9:41:19.062 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	0 (Unknown)	Int32	9:41:19.062 AM	9:41:19.062 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	0 (Unknown)	Int32	9:41:19.062 AM	9:41:19.062 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	0 (Unknown)	Int32	9:41:19.062 AM	9:41:19.062 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	0 (Unknown)	Int32	9:41:19.063 AM	9:41:19.063 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	0 (Unknown)	Int32	9:41:19.063 AM	9:41:19.063 AM	Good

Figure A.16.: Data Access View - Step 0: Default, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:40:38.723 AM	9:40:38.723 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good

Figure A.17.: Data Access View - Step 1: Initial Position and Command, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	2 (NotAble)	Int32	9:43:05.116 AM	9:43:05.116 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	2 (NotAble)	Int32	9:43:05.117 AM	9:43:05.117 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	4 (UnintendedPosition)	Int32	9:43:05.117 AM	9:43:05.117 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	4 (FailureCritical)	Int32	9:43:05.118 AM	9:43:05.118 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	2 (NotAble)	Int32	9:43:05.118 AM	9:43:05.118 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	4 (UnintendedPosition)	Int32	9:43:05.118 AM	9:43:05.118 AM	Good

Figure A.18.: Data Access View - Step 2: Point Fails to Reach End Position, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	2 (NotAble)	Int32	9:43:05.116 AM	9:43:05.116 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	2 (NotAble)	Int32	9:43:05.117 AM	9:43:05.117 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	4 (UnintendedPosition)	Int32	9:43:05.117 AM	9:43:05.117 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	4 (FailureCritical)	Int32	9:43:05.118 AM	9:43:05.118 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	4 (UnintendedPosition)	Int32	9:43:05.118 AM	9:43:05.118 AM	Good

Figure A.19.: Data Access View - Step 3: Repair Initiated, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:43:15.130 AM	9:43:15.130 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	1 (Left)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:00.112 AM	9:43:00.112 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	1 (Left)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good

Figure A.20.: Data Access View - Step 4: Repair Completed, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:43:15.130 AM	9:43:15.130 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	2 (Right)	Int32	9:43:20.136 AM	9:43:20.136 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	3 (NoEndpostion)	Int32	9:43:20.136 AM	9:43:20.136 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	2 (MovingToRight)	Int32	9:43:20.137 AM	9:43:20.137 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	3 (NoEndpostion)	Int32	9:43:20.137 AM	9:43:20.137 AM	Good

Figure A.21.: Data Access View - Step 5: Move Left to Right, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:43:15.130 AM	9:43:15.130 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	2 (Right)	Int32	9:43:20.136 AM	9:43:20.136 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	2 (Right)	Int32	9:43:25.143 AM	9:43:25.143 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:25.144 AM	9:43:25.144 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	2 (Right)	Int32	9:43:25.144 AM	9:43:25.144 AM	Good

Figure A.22.: Data Access View - Step 6: Reached Right, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:43:15.130 AM	9:43:15.130 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:30.153 AM	9:43:30.153 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	3 (NoEndpostion)	Int32	9:43:30.154 AM	9:43:30.154 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	1 (MovingToLeft)	Int32	9:43:30.154 AM	9:43:30.154 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	3 (NoEndpostion)	Int32	9:43:30.155 AM	9:43:30.155 AM	Good

Figure A.23.: Data Access View - Step 7: Move Right to Left, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-emd@127.0.0.1	NS4|Numeric|6023	AggregateAbleToMoveStatus	1 (Able)	Int32	9:43:15.130 AM	9:43:15.130 AM	Good
2	pointgroup-emd@127.0.0.1	NS4|Numeric|6304	LastCommandedPosition	1 (Left)	Int32	9:43:30.153 AM	9:43:30.153 AM	Good
3	pointgroup-emd@127.0.0.1	NS4|Numeric|6026	PointAbleToMoveStatus	1 (Able)	Int32	9:43:15.131 AM	9:43:15.131 AM	Good
4	pointgroup-emd@127.0.0.1	NS4|Numeric|6307	Position	1 (Left)	Int32	9:43:35.162 AM	9:43:35.162 AM	Good
5	pointgroup-emd@127.0.0.1	NS4|Numeric|6308	PositionDegraded	4 (NotApplicable)	Int32	9:17:27.129 AM	9:17:27.129 AM	Good
6	pointgroup-emd@127.0.0.1	NS4|Numeric|6305	MovementStatus	3 (NotMoving)	Int32	9:43:35.163 AM	9:43:35.163 AM	Good
7	pointgroup-emd@127.0.0.1	NS4|Numeric|6313	StatusTechnical	1 (Ok)	Int32	9:43:15.132 AM	9:43:15.132 AM	Good
8	pointgroup-emd@127.0.0.1	NS4|Numeric|6030	AbleToMoveStatus	1 (Able)	Int32	9:43:10.125 AM	9:43:10.125 AM	Good
9	pointgroup-emd@127.0.0.1	NS4|Numeric|6300	Position	1 (Left)	Int32	9:43:35.163 AM	9:43:35.163 AM	Good

Figure A.24.: Data Access View - Step 8: Reached Left, own illustration

Event View In Scenario 1, all server instances (pointgroup-ffm, pointgroup-muc, pointgroup-ber, pointgroup-dus, pointgroup-emb and pointgroup-ham) are connected and monitored concurrently, however, this verification will concentrate on pointgroup-ham for clarity. The overview captures the active event monitoring for all servers in the following Figure A.25.

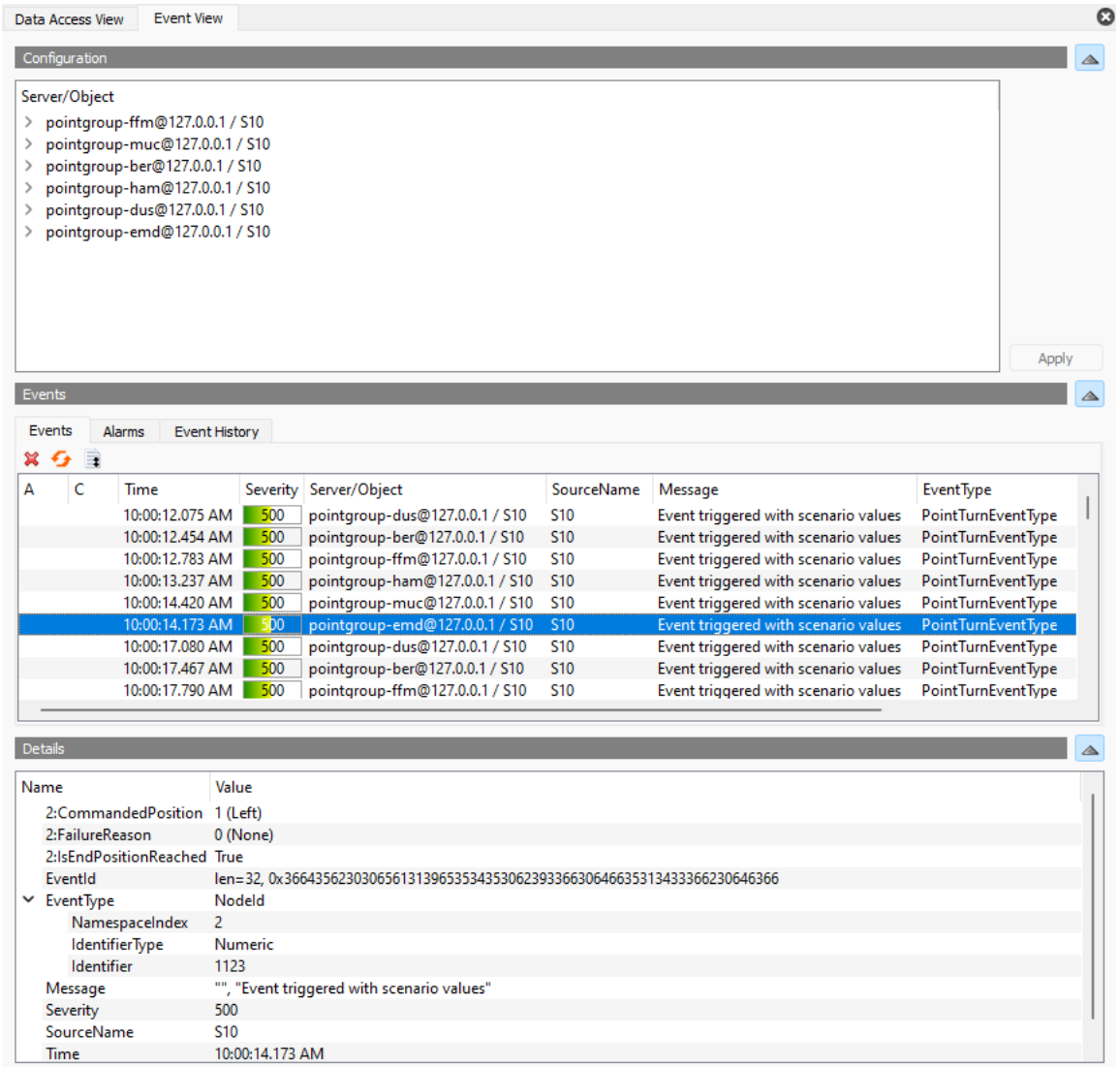


Figure A.25.: Overview of Event Monitoring for All Server Instances, own illustration

The following Figure A.26 - A.33, will only display the pointgroup-ham instance, capturing the UaExpert Event View for Scenario 1, really showing events and diagnostc messages triggered as the scenario progressed. FailureReason and IsEndPositionReached indicate the key indicators that the server identified and stated precisely the failure conditions.

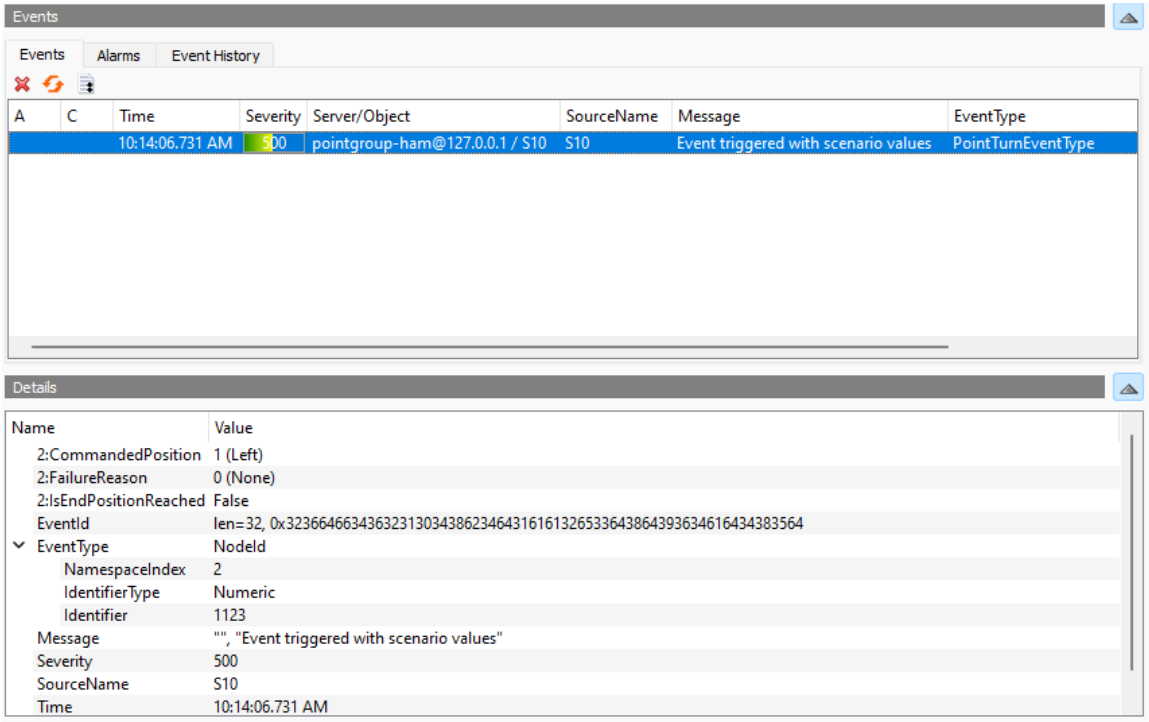


Figure A.26.: Event View - Step 1: Initial Position and Command, own illustration

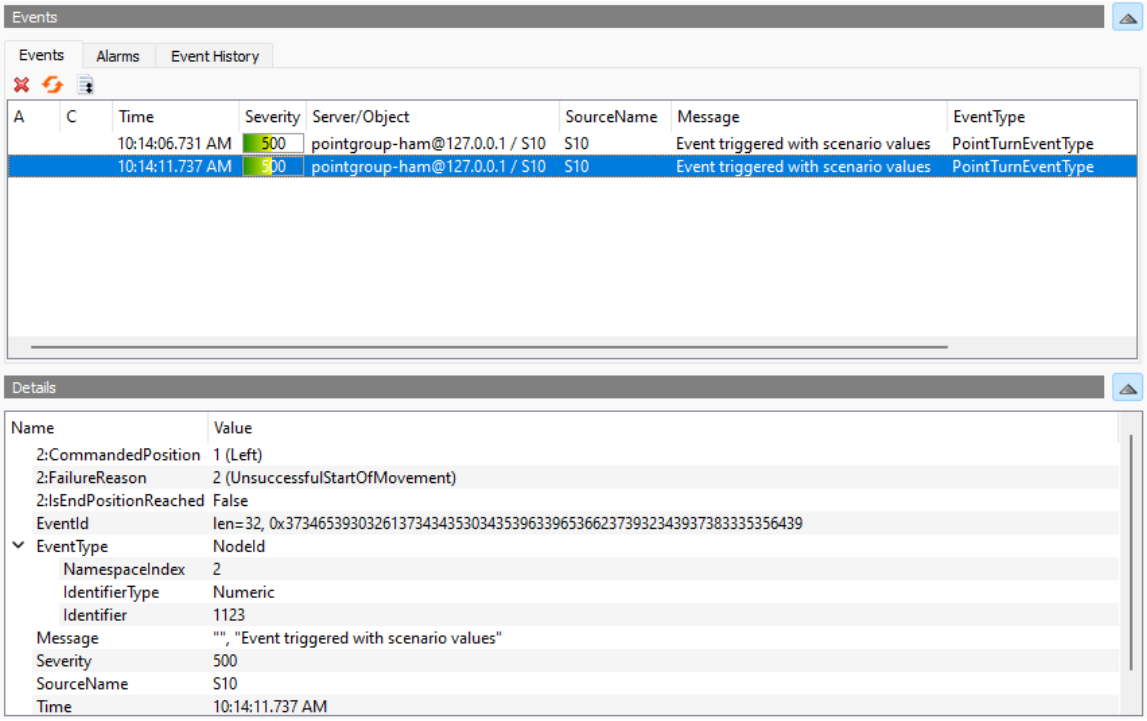


Figure A.27.: Event View - Step 2: Point Fails to Reach End Position, own illustration

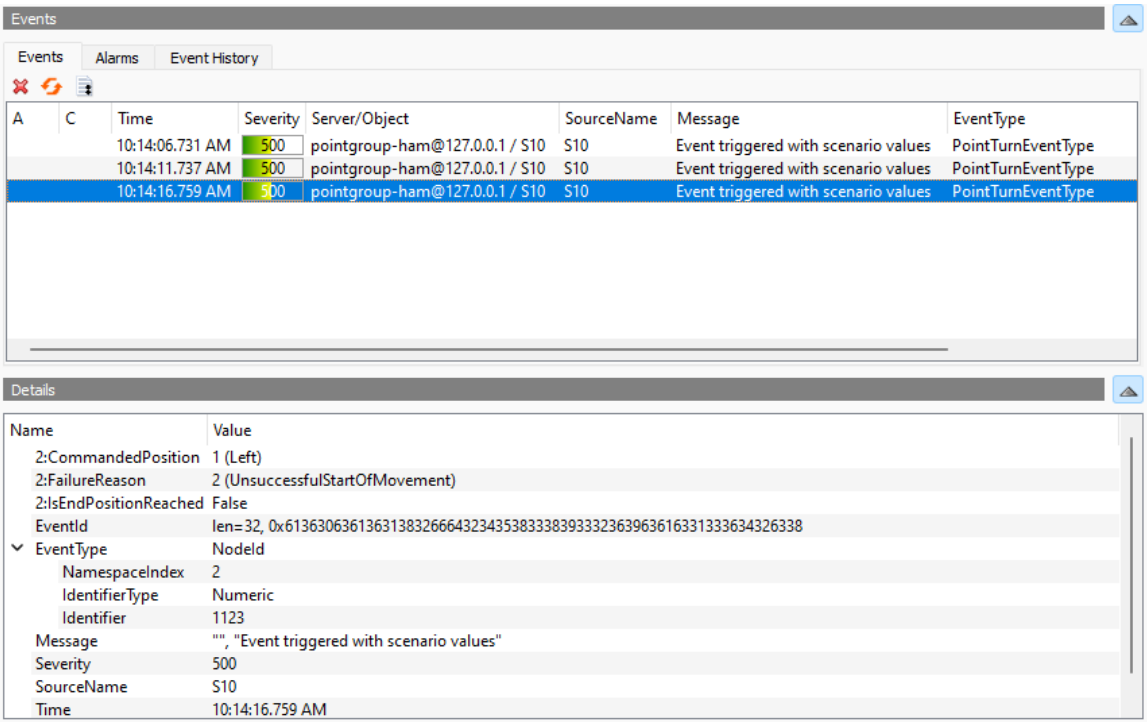


Figure A.28.: Event View - Step 3: Repair Initiated, own illustration

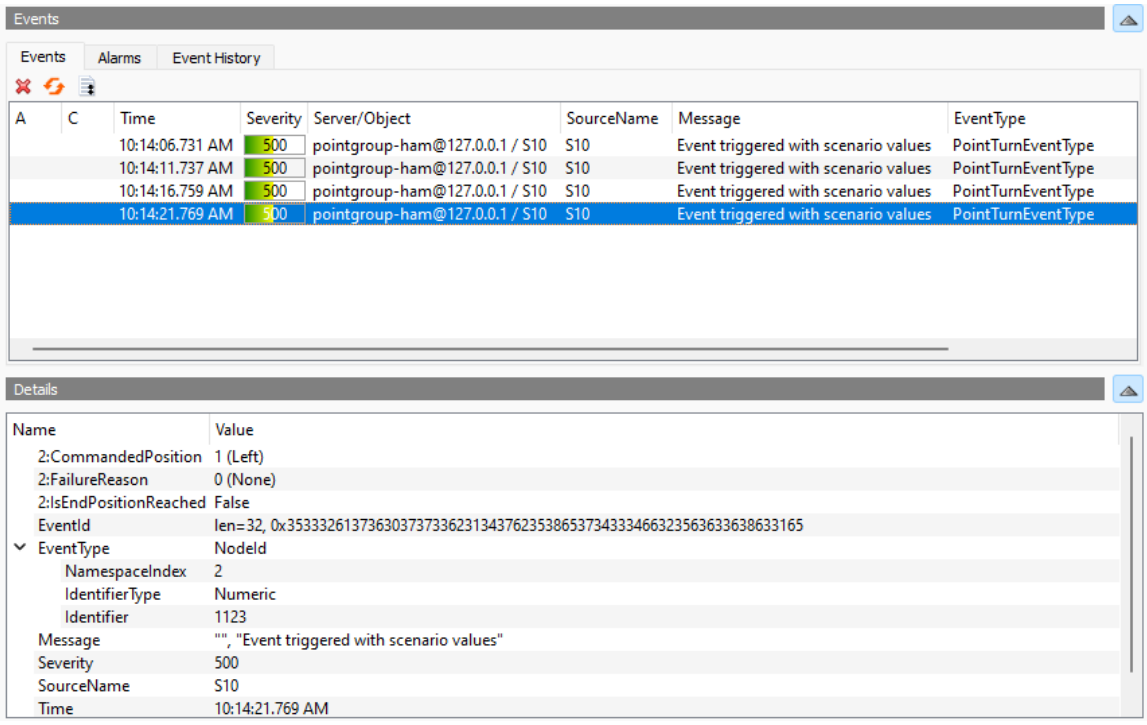


Figure A.29.: Event View - Step 4: Repair Completed, own illustration

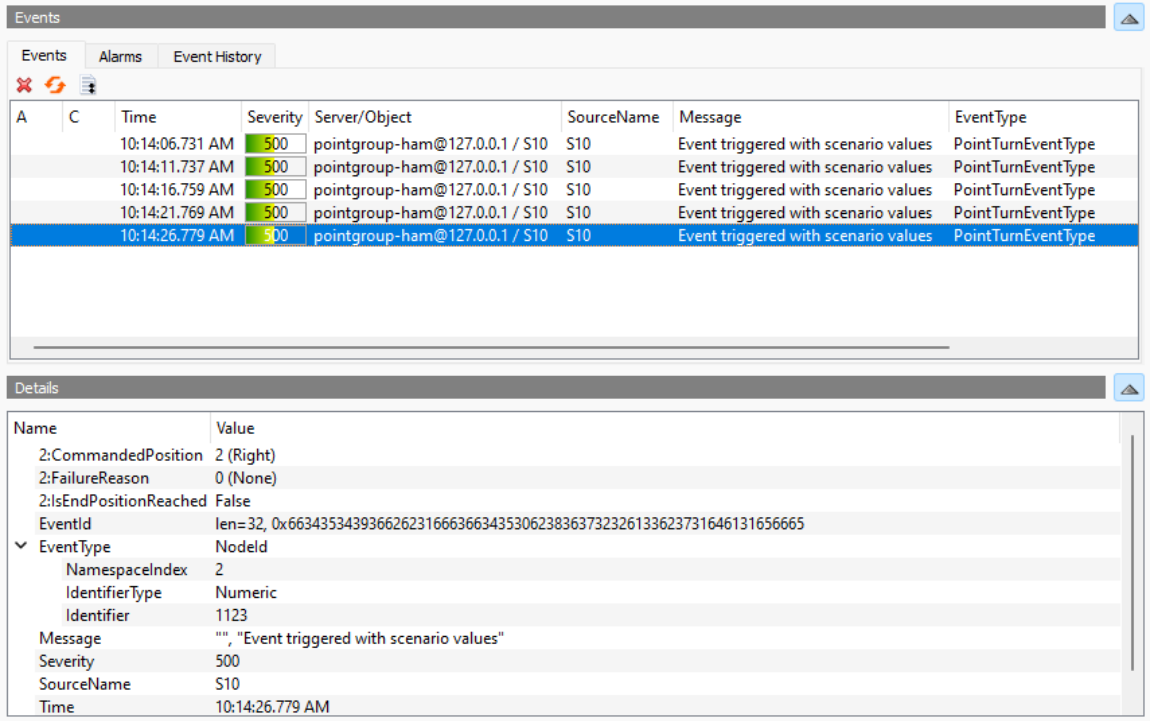


Figure A.30.: Event View - Step 5: Move Left to Right, own illustration

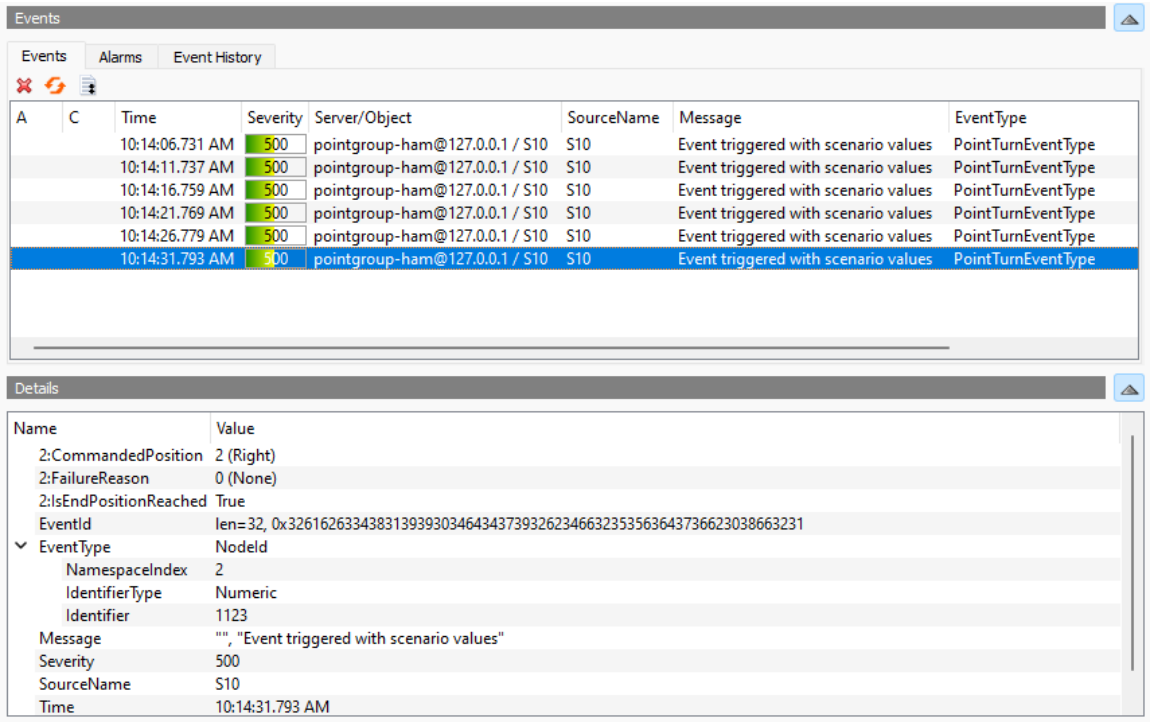


Figure A.31.: Event View - Step 6: Reached Right, own illustration

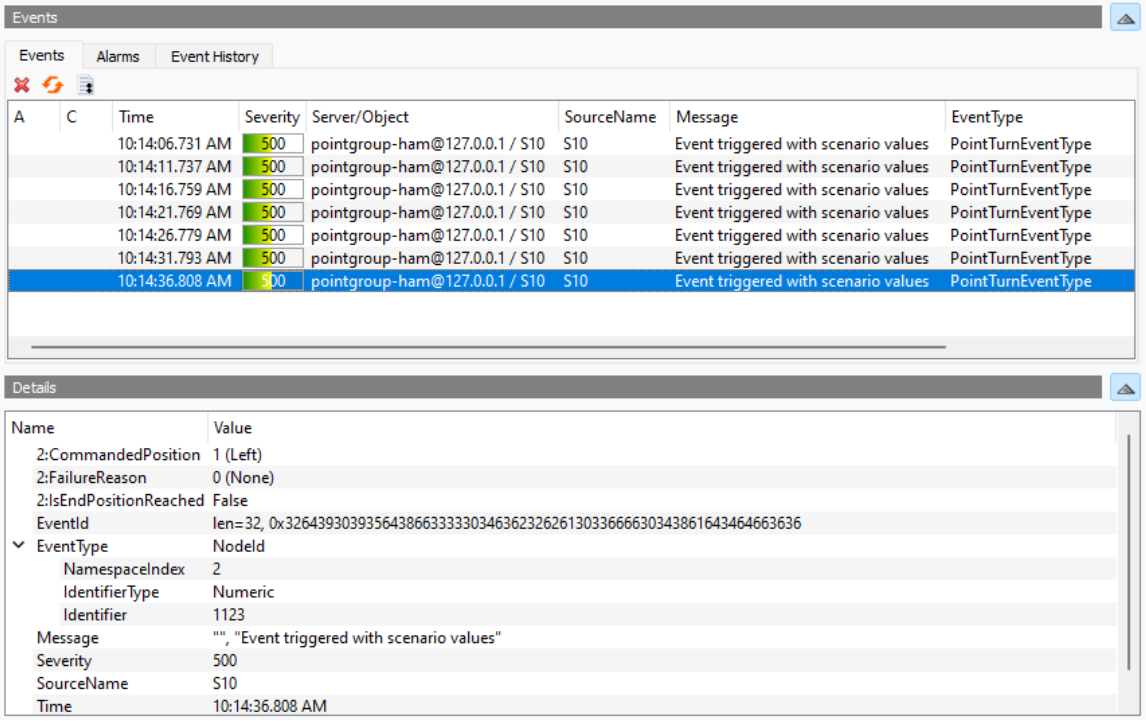


Figure A.32.: Event View - Step 7: Move Right to Left, own illustration

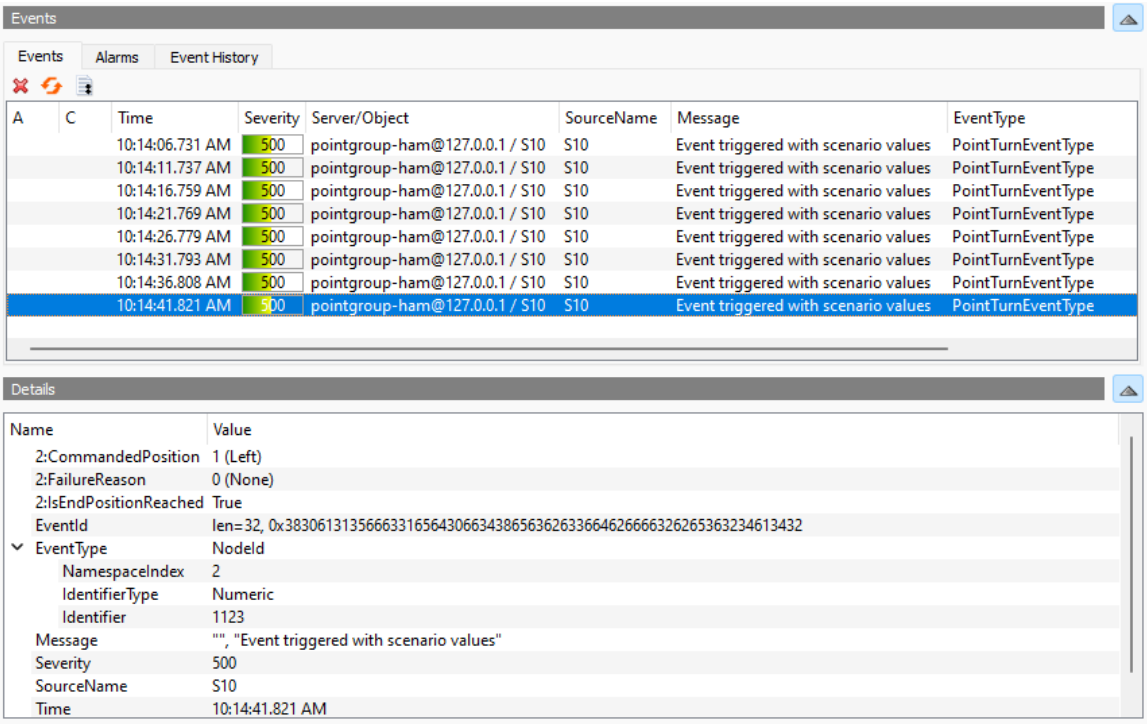


Figure A.33.: Event View - Step 8: Reached Left, own illustration

Scenario 2: Non-Critical Failure in Object Controller

Data Access View This appendix concentrates on pointgroup-ber of the to Object Controller, under the non-critical failure scenario at S6-S10. States of key variables such as StatusTechnical, OperationStatus, CoolingFanStatus, TemperatureStatus, CpuHealthStatus and RamHealthStatus are captured in the following Figure A.34 - A.45. The server's response to the simulated non critical failure was observed through UaExpert's Data Access View to monitor these variables. Data timestamps also sync to confirm the server handled the request immediately.

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	1 (Ok)	Int32	2:38:16.523 PM	2:38:16.524 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	1 (Ok)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	1 (Ok)	Int32	2:38:16.527 PM	2:38:16.527 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	1 (Booting)	Int32	2:38:16.527 PM	2:38:16.527 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.529 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	1 (Normal)	Int32	2:38:16.529 PM	2:38:16.529 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	1 (OnSite)	Int32	2:38:36.664 PM	2:38:36.664 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.34.: Data Access View - Step 0: Default, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	0 (Unknown)	Int32	2:40:02.004 PM	2:40:02.004 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	0 (Unknown)	Int32	2:40:02.004 PM	2:40:02.004 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.529 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	1 (Normal)	Int32	2:38:16.529 PM	2:38:16.529 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.35.: Data Access View - Step 1: Cooling Fan Failure, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	1 (Ok)	Int32	2:40:07.010 PM	2:40:07.010 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.529 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	1 (Normal)	Int32	2:38:16.529 PM	2:38:16.529 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.36.: Data Access View - Step 2: Temperature High Warning, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	1 (Ok)	Int32	2:40:07.010 PM	2:40:07.010 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.528 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	1 (Normal)	Int32	2:38:16.528 PM	2:38:16.529 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	2 (Degraded)	Int32	2:40:12.019 PM	2:40:12.019 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.37.: Data Access View - Step 3: RAM Degradation Detected, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	1 (Ok)	Int32	2:40:07.009 PM	2:40:07.009 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	1 (Ok)	Int32	2:40:07.010 PM	2:40:07.010 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	2 (Degraded)	Int32	2:40:17.033 PM	2:40:17.033 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	2 (High)	Int32	2:40:17.034 PM	2:40:17.034 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	2 (Degraded)	Int32	2:40:12.019 PM	2:40:12.019 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.38.: Data Access View - Step 4: CPU Health Degraded, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	2 (Warning)	Int32	2:40:22.056 PM	2:40:22.056 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	2 (Warning)	Int32	2:40:22.057 PM	2:40:22.057 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	2 (Warning)	Int32	2:40:22.058 PM	2:40:22.058 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	2 (Degraded)	Int32	2:40:17.033 PM	2:40:17.033 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	0 (Unknown)	Int32	2:40:22.060 PM	2:40:22.060 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	2 (Degraded)	Int32	2:40:12.019 PM	2:40:12.019 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.39.: Data Access View - Step 5: Status Warning Triggered, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.071 PM	2:40:27.071 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.071 PM	2:40:27.071 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.072 PM	2:40:27.072 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	2 (InOperation)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	2 (Degraded)	Int32	2:40:17.033 PM	2:40:17.033 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	0 (Unknown)	Int32	2:40:22.060 PM	2:40:22.060 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	3 (Failure)	Int32	2:40:27.073 PM	2:40:27.073 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.40.: Data Access View - Step 6: RAM Failure, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.522 PM	2:38:16.523 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.071 PM	2:40:27.071 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:38:16.525 PM	2:38:16.525 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.071 PM	2:40:27.071 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	3 (FailureNonCritical)	Int32	2:40:27.072 PM	2:40:27.072 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	3 (Fallback)	Int32	2:40:32.084 PM	2:40:32.084 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	3 (Failure)	Int32	2:40:32.085 PM	2:40:32.085 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	3 (Critical)	Int32	2:40:32.085 PM	2:40:32.085 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	3 (Failure)	Int32	2:40:27.073 PM	2:40:27.073 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.41.: Data Access View - Step 7: CPU Failure, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:38:16.521 PM	2:38:16.521 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	2 (ReplaceableAtOperation)	Int32	2:40:37.093 PM	2:40:37.093 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.093 PM	2:40:37.094 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:38:16.524 PM	2:38:16.524 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	2 (ReplaceableAtOperation)	Int32	2:40:37.094 PM	2:40:37.094 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.094 PM	2:40:37.094 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.096 PM	2:40:37.096 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	3 (Fallback)	Int32	2:40:32.084 PM	2:40:32.084 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	3 (Failure)	Int32	2:40:32.085 PM	2:40:32.085 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	0 (Unknown)	Int32	2:40:37.097 PM	2:40:37.097 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	3 (Failure)	Int32	2:40:27.073 PM	2:40:27.073 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	5 (InternalFailure)	Int32	2:40:37.097 PM	2:40:37.097 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.42.: Data Access View - Step 8: Controller Critical Failure, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	false	Boolean	2:40:42.103 PM	2:40:42.103 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	2 (ReplaceableAtOperation)	Int32	2:40:37.093 PM	2:40:37.093 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.093 PM	2:40:37.094 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	false	Boolean	2:40:42.104 PM	2:40:42.104 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	2 (ReplaceableAtOperation)	Int32	2:40:37.094 PM	2:40:37.094 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.094 PM	2:40:37.094 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.096 PM	2:40:37.096 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	3 (Fallback)	Int32	2:40:32.084 PM	2:40:32.084 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	2 (Failure)	Int32	2:40:02.005 PM	2:40:02.005 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	2 (TooHigh)	Int32	2:40:07.010 PM	2:40:07.011 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	3 (Failure)	Int32	2:40:32.085 PM	2:40:32.085 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	0 (Unknown)	Int32	2:40:37.097 PM	2:40:37.097 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	3 (Failure)	Int32	2:40:27.073 PM	2:40:27.073 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	4 (InternalMaintenanceFailure)	Int32	2:40:42.107 PM	2:40:42.107 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.43.: Data Access View - Step 9: Unit Computing Safe Failure, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	false	Boolean	2:40:42.103 PM	2:40:42.103 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	0 (Unknown)	Int32	2:40:47.113 PM	2:40:47.113 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.093 PM	2:40:37.094 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	false	Boolean	2:40:42.104 PM	2:40:42.104 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	0 (Unknown)	Int32	2:40:47.113 PM	2:40:47.113 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.094 PM	2:40:37.094 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.0	String	2:38:36.662 PM	2:38:36.662 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.662 PM	2:38:36.662 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_SC_P_2_V1	String	2:38:36.663 PM	2:38:36.663 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.0	String	2:38:36.663 PM	2:38:36.663 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	4 (FailureCritical)	Int32	2:40:37.096 PM	2:40:37.096 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	0 (Unknown)	Int32	2:40:47.113 PM	2:40:47.113 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	0 (Unknown)	Int32	2:40:47.114 PM	2:40:47.114 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	0 (Unknown)	Int32	2:40:47.114 PM	2:40:47.114 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	0 (Unknown)	Int32	2:40:47.114 PM	2:40:47.114 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	0 (Unknown)	Int32	2:40:37.097 PM	2:40:37.097 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	0 (Unknown)	Int32	2:40:47.114 PM	2:40:47.114 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	0 (Unknown)	Int32	2:40:47.114 PM	2:40:47.114 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bookworm	String	2:38:36.664 PM	2:38:36.664 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-07-15T10:32:17.000Z	DateTime	2:38:36.665 PM	2:38:36.665 PM	Good

Figure A.44.: Data Access View - Step 10: Unknown due to replacment, own illustration

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	pointgroup-ber@127.0.0.1	NS3|Numeric|6887	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.538 PM	2:28:05.538 PM	Good
2	pointgroup-ber@127.0.0.1	NS3|Numeric|7295	IsTimeSynchronised	true	Boolean	2:40:52.119 PM	2:40:52.119 PM	Good
3	pointgroup-ber@127.0.0.1	NS3|Numeric|6893	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:40:52.120 PM	2:40:52.120 PM	Good
4	pointgroup-ber@127.0.0.1	NS3|Numeric|6891	StatusTechnical	1 (Ok)	Int32	2:40:52.120 PM	2:40:52.120 PM	Good
5	pointgroup-ber@127.0.0.1	NS3|Numeric|6255	DefaultInstanceBrowseName	0, "Equipment"	QualifiedName	2:28:05.571 PM	2:28:05.571 PM	Good
6	pointgroup-ber@127.0.0.1	NS3|Numeric|7299	IsTimeSynchronised	true	Boolean	2:40:52.120 PM	2:40:52.120 PM	Good
7	pointgroup-ber@127.0.0.1	NS3|Numeric|6781	ReplacementIndication	1 (ReplacementNotNeeded)	Int32	2:40:52.120 PM	2:40:52.120 PM	Good
8	pointgroup-ber@127.0.0.1	NS3|Numeric|6859	StatusTechnical	1 (Ok)	Int32	2:40:52.120 PM	2:40:52.120 PM	Good
9	pointgroup-ber@127.0.0.1	NS3|Numeric|6271	HardwareRevision	v1.1	String	2:40:52.120 PM	2:40:52.120 PM	Good
10	pointgroup-ber@127.0.0.1	NS3|Numeric|6765	ManufacturingDateTime	2024-08-12T15:32:17.000Z	DateTime	2:40:52.121 PM	2:40:52.121 PM	Good
11	pointgroup-ber@127.0.0.1	NS3|Numeric|6857	SerialNumber	SN_OC_P_2_V2	String	2:40:52.121 PM	2:40:52.121 PM	Good
12	pointgroup-ber@127.0.0.1	NS3|Numeric|6858	SoftwareRevision	v1.1	String	2:40:52.121 PM	2:40:52.121 PM	Good
13	pointgroup-ber@127.0.0.1	NS3|Numeric|6919	DefaultInstanceBrowseName	0, "Controller"	QualifiedName	2:28:05.581 PM	2:28:05.581 PM	Good
14	pointgroup-ber@127.0.0.1	NS3|Numeric|6923	StatusTechnical	1 (Ok)	Int32	2:40:52.121 PM	2:40:52.121 PM	Good
15	pointgroup-ber@127.0.0.1	NS3|Numeric|6920	OperationStatus	1 (Booting)	Int32	2:40:52.121 PM	2:40:52.121 PM	Good
16	pointgroup-ber@127.0.0.1	NS3|Numeric|7337	CoolingFanStatus	1 (Normal)	Int32	2:40:52.122 PM	2:40:52.122 PM	Good
17	pointgroup-ber@127.0.0.1	NS3|Numeric|6924	TemperatureStatus	1 (Normal)	Int32	2:40:52.122 PM	2:40:52.122 PM	Good
18	pointgroup-ber@127.0.0.1	NS3|Numeric|6917	CpuHealthStatus	1 (Normal)	Int32	2:40:52.122 PM	2:40:52.122 PM	Good
19	pointgroup-ber@127.0.0.1	NS3|Numeric|6918	CpuLoadStatus	1 (Normal)	Int32	2:40:52.122 PM	2:40:52.122 PM	Good
20	pointgroup-ber@127.0.0.1	NS3|Numeric|6921	RamHealthStatus	1 (Normal)	Int32	2:40:52.122 PM	2:40:52.122 PM	Good
21	pointgroup-ber@127.0.0.1	NS3|Numeric|7336	BootingLastReason	3 (InternalMaintenanceOk)	Int32	2:40:52.123 PM	2:40:52.123 PM	Good
22	pointgroup-ber@127.0.0.1	NS3|Numeric|7339	OperatingSystem	Linux Debian Bullseye	String	2:40:52.123 PM	2:40:52.123 PM	Good
23	pointgroup-ber@127.0.0.1	NS3|Numeric|7335	BootingLastDateTime	2024-08-12T15:32:17.000Z	DateTime	2:40:52.123 PM	2:40:52.123 PM	Good

Figure A.45.: Data Access View - Step 11: After Replacement Status, own illustration