NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

HENRIQUE EDUARDO MAGALHÃES PAIS DE SÁ

BSc in Electrical and Computer Engineering

DEVELOPMENT OF AUTOMATED NOVAAS CONNECTORS

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DEVELOPMENT OF AUTOMATED NOVAAS CONNECTORS

HENRIQUE EDUARDO MAGALHÃES PAIS DE SÁ

BSc in Electrical and Computer Engineering

Adviser: Pedro Maló
NOVA School of Science and Technology

Co-adviser: Giovanni di Orio
UNINOVA

Examination Committee

Chair: Rodolfo Oliveira
NOVA School of Science and Technology

Rapporteurs: Filipe Moutinho
NOVA School of Science and Technology

Pedro Maló
NOVA School of Science and Technology

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2023

Development of automated NOVAAS Connectors

Copyright © Henrique Eduardo Magalhães Pais de Sá, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) (Lourenço, 2021).

Acknowledgements

I want to start by thanking my adviser, Professor Pedro Maló and my co-adviser Giovanni di Orio. Their help and guidance was essential in the making of this dissertation. I am indebted to the professor for allowing me to compose a dissertation on a topic that I find most interesting. Their insights and the discussions had, helped me improve my work ethic and provided the motivation to do my best.

I also want to thank NOVA School of Science and Technology for providing a conducive academic environment and access to resources essential to the completion of this dissertation.

Finally, I want to express my deepest gratitude to my family and friends. Their encouragement and belief in me during this process has been highly motivating. Also, without the support and camaraderie of the 99% institution I would not be the person I am today, and for that I am eternally grateful. I also want to thank my girlfriend for motivating me in the final stretch, and for being there for me when a break was needed.

"There is a way out of every box, a solution to every puzzle; it's just a matter of finding it."

— Captain Jean-Luc Picard

Abstract

The rising popularity of Internet of Things (IoT) reveals humanity's trend towards a more interconnected world. As more and more people interact with this environment, the problems of interoperability and scalability rise. With the large quantity of data moving through the IoT space, several issues surface on many levels. Therefore, it is vital that the data-transfer layer of IoTs function as efficiently and expeditiously as possible.

This dissertation aims to solve some of the issues that might exist when connecting devices to an IoT, by creating an "IoT Connector" and applying it to the NOVA Asset Administration Shell (NOVAAS).

This connector is built using the Python programming language, and acts as a driver for assets connected to NOVAAS. It takes the JavaScript Object Notation (JSON) file that represents an asset as the input and populates a Node-Red flow with the gathered asset properties. This tool significantly reduces the amount of effort required by the developer, effectively automating this task.

Additionally, the NOVAAS Connector created is implemented in an industrial real-world application.

Keywords: Internet of Things, Asset Administration Shell, Connector, Adapter, Asset, JSON, Python

Resumo

O aumento da popularidade da ”Internet das Coisas” revela a tendecência da humanidade para um mundo cada vez mais interligado. À medida que a civilização interage com este conceito, os problemas de interoperabilidade e escalabilidade vão aumentando. Tendo em conta a elevada quantidade de dados recebida e transmitida nestas redes, várias dificuldades surgem a diversos níveis de operação. Consequentemente, é vital que a camada de transferência de dados destas redes funcione o mais eficiente e expedito possível.

Esta dissertação visa a resolver alguns dos problemas que possam surgir quando se ligam dispositivos a uma rede IoT, ao criar e implementar um ”conector IoT” aplicado à plataforma NOVAAS.

O conector é construido em Python e funciona como um adaptador para ativos industriais ligados à plataforma NOVAAS. Usa como entrada o ficheiro JSON de descrição dos ativos e preenche um fluxo de Node-Red com as propriedades adquiridas. Com a ajuda desta ferramenta, o desenvolvedor da NOVAAS pode, efetivamente, automatizar a tarefa manual de construção de adaptadores.

Adicionalmente, o conector NOVAAS é implementado em contexto industrial.

Palavras-chave: Internet das Coisas, Asset Administration Shell, Conector, Adaptador, Ativos industriais, JSON, Python

Contents

List of Figures ix

Acronyms xii

1 Introduction 1

1.1 Motivation 1

1.2 Problem Description 1

1.3 Demonstration of the Problem 2

1.4 Contributions 3

1.4.1 NOVAAS Connector Tool Developed 3

1.4.2 Industrial Application of the Tool in the BD4NRG Project 3

1.5 Dissertation Structure 3

2 Background 5

2.1 Internet of Things 5

2.1.1 Industrial Internet of Things 5

2.1.2 Industry 4.0 6

2.2 Asset Administration Shell 6

2.3 NOVAAS: NOVA Asset Administration Shell 8

2.4 Node-Red 8

3 State of the art 9

3.1 Software Connectors 9

3.2 Software Design Patterns 10

3.2.1 The Adapter Pattern 10

3.2.2 Adapter types comparison 14

3.2.3 The Facade Pattern 14

3.2.4 The Decorator Pattern 15

3.3 Choosing the most fitting pattern 15

3.4 Applications of adapters in various real world scenarios 16

3.4.1 Wrappers applied to legacy code 16
3.4.2 Developing a Cloud Environment with Software Design Patterns 17
3.4.3 Building an Interface for health monitoring devices, using the wrapper pattern 18
3.4.4 IoT Framework using wrappers 19
3.5 Semi-automatic code generation 19
3.5.1 Automatic generation of OPC UA connectors 20
3.5.2 Generating Node-Red flows in an IoT context 21
3.6 Final considerations 21

4 NOVAAS Connector 22
4.1 Connector Structure 22
4.2 Solution Implementation 23
4.2.1 First version 24
4.2.2 Second version 30
4.2.3 Final version 31

5 Tests and Results 35
5.1 Main NOVAAS Implementation Test 35
5.2 PROSYS OPC-UA Simulation Test 38
5.3 NOVAAS Hydraulic Simulation Test 39
5.4 User Experience: Results 40

6 Industrial Application 42
6.1 Big Data for Next Generation Energy 42
6.2 Large-scale Pilot Nine Introduction 43
6.3 NOVAAS Connector's implementation in LSP9 44

7 Conclusions and future work 46

Bibliography 48

Appendices

A Appendix 52
A.1 Main test results 52
A.2 Prosys test results 56
A.3 Hydraulic Plant Simulator test results 59

List of Figures

1.1 Snippet of NOVAAS application in Node-Red, representing a test asset . . . 2

2.1 Proposed structure of the Asset Administration Shell (AAS), adapted from (di Orio et al., 2019) . . . 7

3.1 Object adapter structure, taken from (Shvets, 2023) . . . 11
3.2 Class adapter structure, taken from (Freeman and Robson, 2020a) . . . 13
3.3 Diagram of a compiler facade, taken from (Gamma et al., 1994b) . . . 14
3.4 Example diagram of a decorator, taken from ("Design patterns - decorator pattern", 2023) . . . 15
3.5 Class diagram of the cloud environment model, taken from (Markoska et al., 2016) . . . 17
3.6 Classloader diagram, taken from (Kobylarz and Danda, 2013) . . . 18
3.7 Wrappers Repository, taken from (Martino et al., 2016) . . . 19
3.8 NOVAAS Connector structure . . . 21

4.1 Structure of NOVAAS including the Connector, adapted from (Di Orio, 2021) 22
4.2 Flow diagram of tool's tasks . . . 23
4.3 Node from Listing 8 in Node-Red flow . . . 27
4.4 Node group connected to the first of Property, sample asset . . . 28
4.5 Property 1 configuration, sample asset . . . 28
4.6 Example of a *trigger_out* node with seven properties linked . . . 32
4.7 Property handler subflow template . . . 32

5.1 Picture of the physical asset . . . 35
5.2 Information section of the NOVAAS compact cylinder application . . . 36
5.3 Environment tab of this application . . . 37
5.4 List of all the Properties/Events contained in the asset file . . . 37
5.5 *OperationalData* tab of PROSYS OPC-UA Simulation Server . . . 38
5.6 List of properties in PROSYS OPC-UA Simulation Server . . . 38
5.7 *OperationalData* tab of an Hydraulic Plant Simulation . . . 39

6.1 Solar Panel farm where large-scale pilot number nine takes place, located in Alcoutim. Taken from "ENERCOUTIM webpage", 2023 43
6.2 Overview of NOVAAS implementation in LSP9 44

A.1 Test result of 5.1, Property groups 1, 3, 5 and 7 52
A.2 Test result of 5.1, Property groups 2, 4 and 6 53
A.3 Test result of 5.1, Property 1 configuration 53
A.4 Test result of 5.1, Property 2 configuration 54
A.5 Test result of 5.1, Property 3 configuration 54
A.6 Test result of 5.1, Property 4 configuration 54
A.7 Test result of 5.1, Property 5 configuration 55
A.8 Test result of 5.1, Property 6 configuration 55
A.9 Test result of 5.1, Property 7 configuration 55
A.10 Test result of 5.2, Property groups 1, 3 and 5 56
A.11 Test result of 5.2, Property groups 2, 4 and 6 57
A.12 Test result of 5.2, Property 1 configuration 57
A.13 Test result of 5.2, Property 2 configuration 58
A.14 Test result of 5.2, Property 3 configuration 58
A.15 Test result of 5.2, Property 4 configuration 58
A.16 Test result of 5.2, Property 5 configuration 59
A.17 Test result of 5.2, Property 6 configuration 59
A.18 Test result of 5.3, Property groups 1, 2 and 3 60
A.19 Test result of 5.3, Property 1 configuration 60
A.20 Test result of 5.3, Property 2 configuration 61
A.21 Test result of 5.3, Property 3 configuration 61

List of Listings

1 Example of an object adapter in Java, from ("Adapter", 2023) 12
2 Adapter class code for the previous Listing 1, from ("Adapter", 2023) . . 13
3 Algorithm used by (Lyu et al., 2022) 20
4 Initial snippet of first version script 24
5 Top-most segment of JSON asset file 25
6 Inside *"submodels"* JSON object 25
7 Third loop, which parses all nested Collections 26
8 Snippet of Node-Red flow node in JSON 27
9 Section of code depicting the creation of JSON objects in final Node-Red file 29
10 Creation and connection of the change node to the Property node 30
11 Abbreviated section of the code tasked with creating the property handler subflow template 33
12 Snippet of the code that generates the *trigger_in* node 33
13 Logic in creating the position of the nodes 34

Acronyms

AAS Asset Administration Shell *(pp. ix, 2, 3, 6–8, 44, 46, 47)*
API Application Programming Interface *(pp. 8, 17)*

BD4NRG Big Data for Next Generation Energy *(pp. 3, 42, 43)*

CPS Cyber-Physical Systems *(p. 6)*

I4.0 Industry 4.0 *(p. 6)*
IIoT Industrial Internet of Things *(pp. 5, 6)*
IoT Internet of Things *(pp. v, vi, 1, 2, 5, 6, 8, 19, 21, 42)*
IT Information Technology *(pp. 6, 44)*

JNI Java Native Interface *(p. 16)*
JSON JavaScript Object Notation *(pp. v, vi, xi, 16, 20, 21, 23–32, 36, 46)*

NOVAAS NOVA Asset Administration Shell *(pp. v, vi, ix, x, 2, 3, 8, 15, 16, 18, 19, 21–24, 35, 36, 44–46)*

OT Operational Technology *(p. 6)*

SGBDAA Smart Grid Big Data Analytics Alliance *(p. 42)*

UML Unified Modeling Language *(p. 21)*

XML Extensible Markup Language *(pp. 20, 21)*

1

Introduction

1.1 Motivation

Throughout our daily lives, the IoT, is omnipresent. It is in our homes, offices, and cars. It is even on our bodies through wearable devices like smart watches or in our wallets in the form of credit or debit cards. Nowadays it is common to find even more technological advances in this field such as contactless cards, which let you complete transactions without the need to insert the card in the payment terminal. Yet another demonstration of IoT in the real world is our own smartphone that can act as an electronic key to unlock an electric scooter found on city streets, for example.

All these examples, and more, are shaping the present and future of mankind not only from a personal perspective, but also from the perspective of a community. We can also apply these concepts to a larger-scale framework, by implementing IoT in entire cities. By deploying various electronic technologies such as pollution sensors, traffic heat maps and more, it is possible to create an interconnected network of devices – an IoT - that keeps us informed of what is happening in real time.

1.2 Problem Description

There is no set framework for an IoT, however, every IoT is composed of multiple layers. Every layer that connects either vertically or horizontally, requires some form of shared protocol in how the data is handled between segments. This is especially true when it comes to enterprise applications, where massive amounts of data are exchanged from machine to machine. This intermediary that bridges the gap between the devices and the data collecting processes in an IoT is called middleware. A large number of studies have been made on this subject such as the one authored by (Razzaque et al., 2016), where many different middleware frameworks are reviewed.

If we view an IoT network as a conglomerate of interconnected devices and the data they share between themselves, it is imperative that the middleware – the layer with the purpose of transferring data from layer to layer – functions as efficiently and expeditiously

as possible. An IoT service relies heavily on handling data throughout all its components. Making the simplest and most efficient approach to building a middleware solution is a key aspect in an IoT network. This is true for all IoTs, whether it is a Healthcare application or an Industry Enterprise execution.

One emerging middleware solution is the AAS, fruit of the new Industry 4.0 revolution (Tantik and Anderl, 2017a). This framework functions as a middleware attached to an asset, which in turn allows it to communicate in a digital-twin environment.

The focus of this dissertation is on simplifying the creation and implementation of software connectors, specially IoT based connectors. The task at hand will involve building a solution based on a structural design pattern which will interface with an asset and provide its data through to the NOVAAS middleware (di Orio et al., 2019).

1.3 Demonstration of the Problem

In order to help the reader better understand the issue at hand, this section will describe the problem succinctly, specifically in regards to the AAS - NOVAAS. This AAS is built in Node-Red - a visual, low-code programming tool ("Node-Red", 2013) - and is composed of multiple flows.

In Fig.1.1, a snippet of the NOVAAS back-end is shown. Each flow represents a structural component of this AAS, such as the "User Interface" or the "Interaction Manager". For this dissertation, the relevant component is the "Operational Data" flow. Every AAS represents an asset or a group of assets. In this case, we will use a compact cylinder as an example asset. This AAS is available in GitLab, along with other examples of asset implementations (Di Orio, 2023).

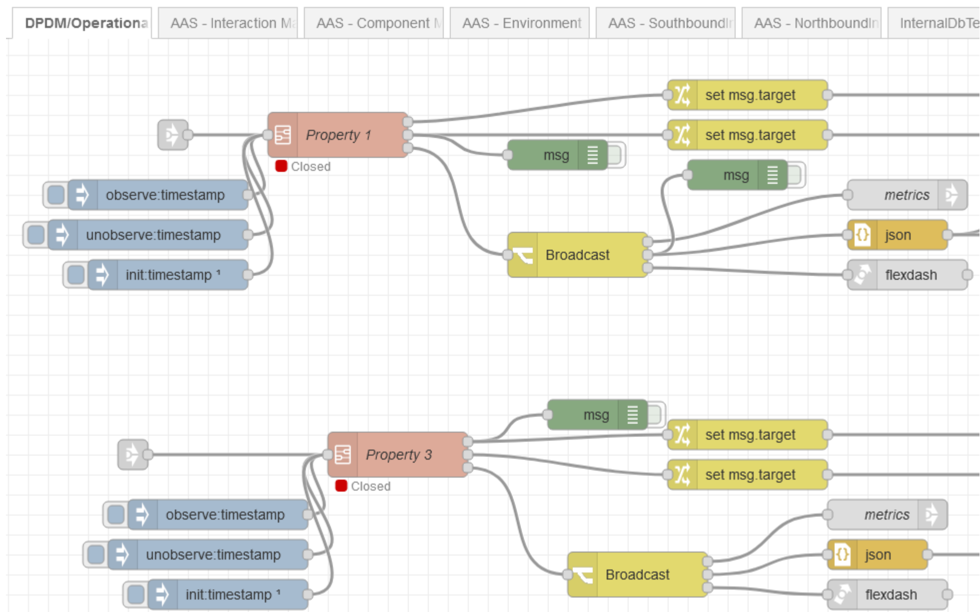

Figure 1.1: Snippet of NOVAAS application in Node-Red, representing a test asset

Every flow is similar to other instances of the NOVAAS with different assets, except the ”Operational Data” flow. Every asset is different when it comes to data types, property names and events. This means that the flow has to be hand-made according to the specifications of the asset. For example, one asset might have three properties, while another can have ten properties aggregated into two ”Submodel Element Collections”. Because of this aspect, creating a NOVAAS model for a new asset, presents a significant amount of effort to develop. Due to the time investment and complexity of differing asset representations, production of new models is less efficient than desired.

The logical step to take is to make the process of constructing this flow more efficient and less time consuming. The ideal solution would involve removing the manual development of each Operational Data flow, by automating it.

1.4 Contributions

This dissertation has provided two significant contributions in the field of Software Connectors, applied to AAS. They are as follows:

1.4.1 NOVAAS Connector Tool Developed

The development of the NOVAAS Connector generation tool constitutes the first contribution. This tool was successfully created, improving the workflow of the NOVAAS developer, by automating a laborious manual task. This also constitutes an advancement in the development of Adapters in an Industrial setting, when dealing with the variety and abundance of assets with different data representations.

A demonstration and detailed explanation of the solution implemented is present in Chapter 4. The tests performed on this tool and their respective results, are presented in Chapter 5.

1.4.2 Industrial Application of the Tool in the BD4NRG Project

This tool was also integrated in a real-world scenario, more specifically, the Big Data for Next Generation Energy (BD4NRG) European project. Chapter 6 provides an overview of this European project, along with an explanation of the tool's integration in it. This industrial application demonstrates the real-world usefulness of the tool created, providing another helpful contribution to the development of automated adapter tools.

1.5 Dissertation Structure

This dissertation consists of seven Chapters.

The first Chapter introduces the problem and delineates the contributions of the dissertation.

Chapter 2 goes over a few topics that are not essential to this document, but might shed some light on some of the concepts mentioned throughout.

Chapter 3 touches on the state of the art of the main issue this dissertation aims to solve.

Chapter 4 demonstrates the software tool created to solve the problem as well as its detailed implementation.

Chapter 5 shows the testing made on the tool and its corresponding results.

Chapter 6 displays the real-world industrial application of the tool developed.

Lastly, Chapter 7 presents the conclusions and future work to be done.

2 Background

This Chapter will provide further information on the research done, helping the reader to better understand some concepts which might not be directly related to the problem this dissertation aims to solve. Although not essential to understanding the problem, the background covers a few topics that should enable a more complete view of the environment surrounding the topic of this dissertation.

2.1 Internet of Things

According to (Lynn et al., 2020), the Internet of Things (IoT) is a concept that envisions the intercommunication of everyday objects through the internet. These objects, often referred to as ”things”, comprise a wide variety of devices, including consumer electronics like smartphones and wearables to industrial sensors and actuators.

Over the past decade, the IoT has gained considerable momentum and has the potential to bring about a paradigm shift in how we consume information and interact with our surroundings (Sunyaev, 2020).

It can be understood from both technical and socio-technical perspectives. Technically, it represents an ecosystem of interconnected devices with the ability to be configured based on structured and synergistic standards and guidelines. From a socio-technical standpoint, it emphasizes the integration of the physical world, digital virtual world, and human perception.

This topic encompasses a large amount of concepts that are important to understanding this dissertation, but not relevant enough to be explained in detail. For more information on this topic, see (Atlam et al., 2018).

2.1.1 Industrial Internet of Things

The Industrial Internet of Things (IIoT) is a subgroup of the Internet of things (IoT), and it refers to industrial connectivity technologies applied in the automation and in the machine-to-machine sector. IIoT systems help companies gather data about their manufacturing processes so that they can be more efficient and sustainable.

Much like how IoT is changing the way society interacts with technology, IIoT is bringing forth an industrial change towards the digital and smart devices era. This constitutes the integration of Operational Technology (OT) with Information Technology (IT) (Mandler et al., 2016).

2.1.2 Industry 4.0

The Industry 4.0 (I4.0) concept, first defined in Germany, has been gaining international recognition and is now widely adopted as a means of applying new IIoT solutions to boost production efficiency through smart, IoT enabled services within the framework of smart factories (Sisinni et al., 2018).

Cyber-Physical Systems (CPS) make it possible to represent physical objects in the digital world, using now standard technologies to integrate these objects together. This virtual representation of a physical object in real time, represents a new concept - the ”digital twin”. These digital twins are creating new opportunities to innovate across the entire product life span, from initial design all the way through recycling of manufactured products.

2.2 Asset Administration Shell

Asset Administration Shells (AAS) are a fundamental component of the Industrial Internet of Things (IIoT) and Industry 4.0, acting as gateways that interface with legacy software systems to communicate information about assets. They were created to increase the efficiency of industry by managing assets, promoting cooperation between companies and making data available. AAS can be thought of as a digital twin, or mirror image, of physical assets located within factories.

An AAS serves as the central coordinator for an entity’s communication with other systems while housing its essential functionality. It comprises two key interfaces: an internal, manufacturer-specific one for interacting with the entity itself, and a uniform external interface for communicating with other systems. This design allows for effective standardization without compromising the entity’s core functionality.

The entity under administration can encompass a collection of devices or nested I4.0 components. This ability to administer the collection of nested devices, significantly augments the intricacy of the top-tier AAS, but enables the creation of a flexible network within the entire production framework (Tantik and Anderl, 2017b).

From analysing Fig.2.1, an AAS is composed of two sections: the header; and the body. The manifest, located in the header, is the most important component in the header. It contains crucial information like the identification and functionality of the sub-components and part-models. This is essential for enabling communication between other CPS. It serves as the cornerstone for other I4.0 units to establish connectivity with the AAS, thereby facilitating interoperability.

The body of the AAS, is further divided into various part-models, which can encompass data models, groups of functions, or components. Each of these contains its own header attached to the encompassing body manifest. These headers are standardized and reference specific attributes. The body of each part-model contains the main payload, which holds the information and method execution (Tantik and Anderl, 2017b).

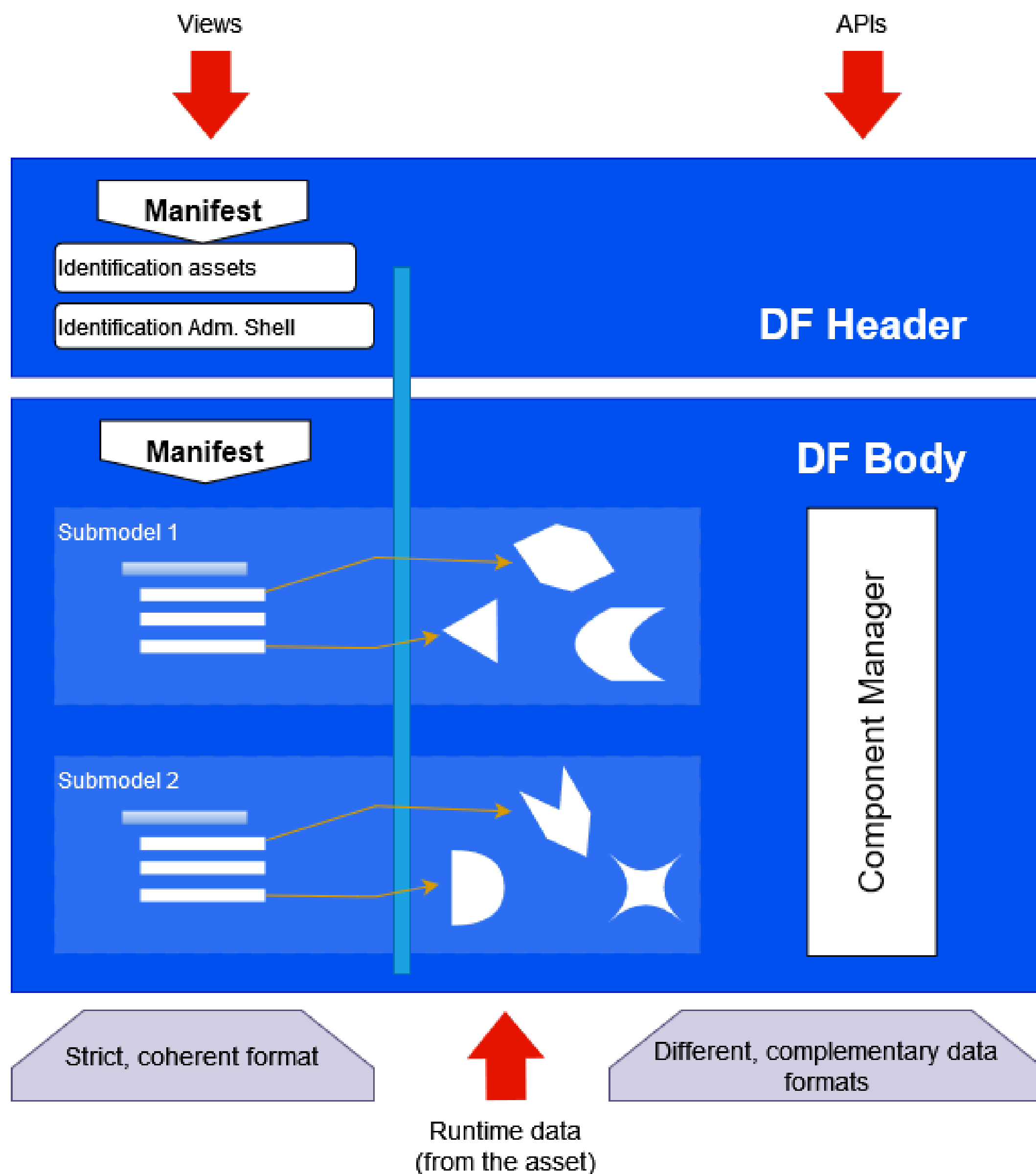

Figure 2.1: Proposed structure of the AAS, adapted from (di Orio et al., 2019)

2.3 NOVAAS: NOVA Asset Administration Shell

NOVAAS is one of solution of AAS made at NOVA School of Science and Technology, authored by (di Orio et al., 2019). This solution provides a reference implementation of the AAS, with the use of internet technologies being the main focus. It is implemented in Node-Red, chosen by the author due to its simplicity in creating and debugging multiple containerized versions of an AAS. All versions of this solution are available in the open-source platform, GitLab at (Di Orio, 2023) and a detailed explanation of this platform can be found at (Di Orio, 2021).

2.4 Node-Red

Node-Red is a visual programming tool tailored for IoT development. Its browser-based editor enables users to connect devices, Application Programming Interface (API)s, and online services through a flow-based interface. With nodes representing various functions, users create workflows by linking them together.

This approach facilitates the creation of automation and integration without extensive coding, making it user-friendly for both developers and non-developers. Node-Red is commonly employed for simplifying the connection and automation of IoT devices, sensors, and data services. A detailed explanation can be found at ("Node-Red", 2013).

3

State of the art

This Chapter will touch on the current status of the problem and the study made in search of solutions that fit this dissertation. It is split into two sections: Software Connectors and Software Design Patterns.

The first section contains a general overview of software connectors and how the solution to be built is related to this area.

The second section gives insight on the theoretical structure approach to building the solution. At the end, the adapter design pattern is chosen and some applications of adapters in real world scenarios are reviewed.

3.1 Software Connectors

Software systems have grown increasingly complex due to the integration of diverse components and the need for interoperability. Software connectors play a pivotal role in facilitating communication and interaction among these components. A definition for a software connector can be found in (Taylor et al., 2009). In the referenced book, the author states that a software connector, in the context of software architecture, is an element tasked with facilitating communication and interaction between different software components.

These connectors act as intermediaries to the components responsible for the main functions of the software, like data processing. In doing so, Connectors help achieve modularity and reusability by allowing software components to collaborate without tightly coupling them together. Software connectors can take various forms, each designed to address specific communication patterns and requirements.

In the case of this dissertation, as mentioned in Chapter 1, the connector to be built will handle the flow of data and its manipulation. It can be considered a connector, because it is facilitating the transfer of data between different software components - the asset designation and the Node-Red flow - which is one of the key roles of a software connector. However, it is important to note that the term ”connector” can have different meanings in different contexts, and not everyone might use the term in this way. The term ”software connector” has evolved and shifted depending on the specific context it is applied in.

After some research on this topic, some generic solutions to creating software connectors were found. However, most were applied in more complex systems related to different environments. Some of these include big enterprise-level applications that handle massive amounts of data on a regular basis. These are often proprietary technologies, owned by Amazon, Google, Microsoft, among others. Because of this factor, it is difficult to obtain peer-reviewed, open-source solutions related to this dissertation. This is specially true, given the niche and very recent topic of Asset Administration Shells in the Industry 4.0 environment, which this connector is inserted in.

The next section will go over the theoretical structure approaches to building a software connector.

3.2 Software Design Patterns

In order to build this connector it is important to get acquainted with the various software design patterns that can be employed. In this section, the relevant structural patterns and their characteristics will be described in detail, to help determine the most appropriate to use as a framework. Afterwards, semi-automatic generation of connectors will be touched on.

Software design patterns are standardized and reusable solutions for frequently encountered issues in software design. These patterns are language-agnostic and not tied to any particular programming language or platform, but rather provide a common vocabulary and a high-level set of principles for designing software.

These design patterns were formally introduced to software engineering by (Gamma et al., 1994d) and have, since then, been considered best practice in software development. It is possible to divide the software design patterns into three main categories: creational, structural, and behavioural.

1. Creational patterns focus on methods for creating objects that meet the specific requirements.

2. Structural patterns address object composition, establishing connections between objects to build more extensive structures.

3. Behavioral patterns concentrate on the interaction between objects, detailing their interplay and collaborative functioning.

For this dissertation, the most applicable structural patterns will be reviewed, more specifically the Adapter, Facade and Decorator patterns.

3.2.1 The Adapter Pattern

According to (Gamma et al., 1994a) the Adapter design pattern, also known as wrapper, is a design pattern of the Structural category, that enables the collaboration of two disparate

interfaces. It does this by wrapping an instance of one class with an interface that is expected by the client.

The Adapter pattern is frequently employed to enable preexisting classes to function with others without the need to alter their source code. For example, imagine that you have a legacy class that needs to be integrated into a new application. The legacy class has a different interface than the classes in the new application, so you cannot use it directly. Instead, you can create an adapter that wraps the legacy class and presents the new interface to the rest of the application.

This concept can be divided into two categories: object adapter and class adapter.

3.2.1.1 Object Adapter

In this implementation the adapter acts as an interface for one object and wraps another object. It can be written in any commonly used programming language. To better understand this concept, we can analyse the mock diagram in Fig.3.1, as reported by (Shvets, 2023).

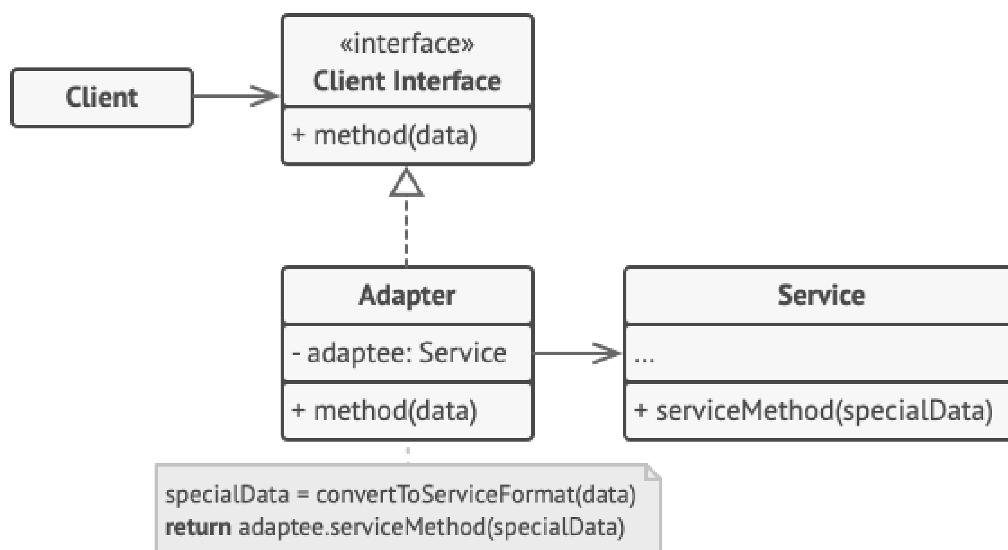

Figure 3.1: Object adapter structure, taken from (Shvets, 2023)

The Client class encapsulates the real-life domain logic of the software. The Client Interface defines a set of rules that other classes must adhere to in order to work with the Client class.

The Service class is a useful, but incompatible, class that the Client is unable to use directly. It may be a third-party or legacy class.

The Adapter class acts as a bridge between the Client and Service classes. It implements the Client Interface and wraps the Service object. When the Client makes a request via the Adapter, it translates the request into a format that the Service object can understand and passes it along to the Service object.

The Client code is decoupled from the Adapter class as long as it communicates with the Adapter through the Client Interface. This means that new types of Adapters can be introduced to the program without affecting the existing Client code. This can be beneficial if the interface of the Service class is modified or replaced. In that case, a new Adapter class can be created without modifying the Client code.

Let's look at a practical example, taken from ("Adapter", 2023), implemented in Java. In this example (as seen in Listing 1) the interface *RowingBoat* and the class *FishingBoat* are declared. Both act similarly, with *RowingBoat* implementing an interface method, *row()*, and *FishingBoat* implementing a *sail()* function. An aditional *Captain* class is declared, implementing the *RowingBoat* interface. Let us assume that the *Captain* class needs to implement the *sail()* method, present in the *FishingBoat* class. Without changing the source code, the *Captain* class cannot use this method, seeing as if it only implements the *RowingBoat* interface.

```
1  public interface RowingBoat {
2    void row();
3  }
4
5  @Slf4j
6  public class FishingBoat {
7    public void sail() {
8      LOGGER.info("The fishing boat is sailing");
9    }
10 }
11
12 public class Captain {
13
14   private final RowingBoat rowingBoat;
15   // default constructor and setter for rowingBoat
16   public Captain(RowingBoat rowingBoat) {
17     this.rowingBoat = rowingBoat;
18   }
19
20   public void row() {
21     rowingBoat.row();
22   }
23 }
```

Listing 1: Example of an object adapter in Java, from ("Adapter", 2023)

By using the *FishingBoatAdapter* (in Listing 2) as the input for the *Captain* class, the *sail()* method is now available through the *row()* method. This way, without changing the source code (in Listing 1), the *Captain* class can now implement the *FishingBoat* method.

```java
@Slf4j
public class FishingBoatAdapter implements RowingBoat {

  private final FishingBoat boat;

  public FishingBoatAdapter() {
    boat = new FishingBoat();
  }

  @Override
  public void row() {
    boat.sail();
  }
}
```

Listing 2: Adapter class code for the previous Listing 1, from ("Adapter", 2023)

3.2.1.2 Class Adapter

In accordance with ("Adapter", 2023), the class adapter differs from the object adapter, as the former requires multiple inheritance to be implemented. This means that the class in question inherits properties from more than one parent class. Because of this, Java cannot be used to implement this type of adapter.

As seen in Fig.3.2, the class adapter diagram is similar to the object adapter diagram. The main difference is that the relation between the adapter and the adaptee is now the same between the adapter and the target class. This means the adapter inherits two classes, as opposed to one in the object adapter diagram, where the adapter's relation to the adaptee is that of composition.

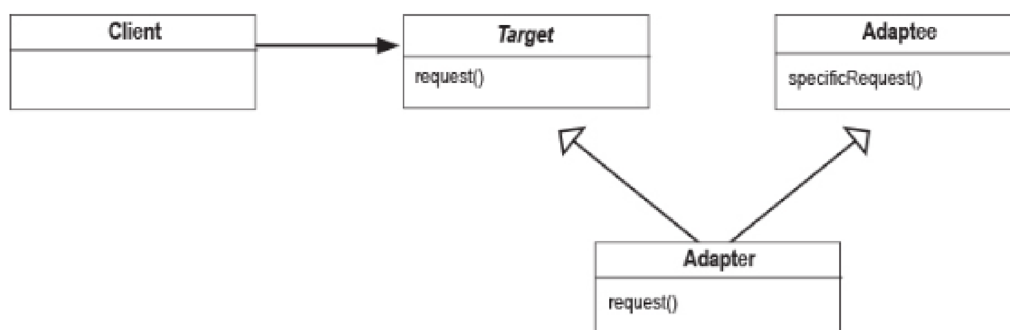

Figure 3.2: Class adapter structure, taken from (Freeman and Robson, 2020a)

3.2.2 Adapter types comparison

Because there are two accepted methods of implementing adapters in software engineering, it is pertinent to compare both the positives and negatives of each.

A class adapter will not adapt sub-classes of the target class, because it commits to a concrete adaptee class. It may override the adaptee's actions, seeing as it is a subclass of the adaptee. It implements only one object, making the path to the adaptee a straightforward one (Gamma et al., 1994a).

On the other hand, an object adapter, opposed to the class adapter, lets one adapter class work with many adaptees and their subclasses. It also cannot override the adaptee's actions as easily, having to refer to the sub-classes of the adaptee, should these exist.

3.2.3 The Facade Pattern

This pattern encapsulates a number of subsystem classes into a single interface, removing the complexity of using the subsystem, as (Freeman and Robson, 2020b) mention in their book. It makes sense to use this pattern in a complex system, where higher-level abstraction is needed. It simplifies the use of the system and it provides a straightforward interface good enough for most clients.

As mentioned in ("Facade", 2023), it is also useful for when there are many dependencies between clients and abstracted classes. By implementing a facade, the subsystem can be separated from both clients and other subsystems, resulting in increased subsystem autonomy and flexibility. Let us look at the example of the compiler, demonstrated in Fig.3.3.

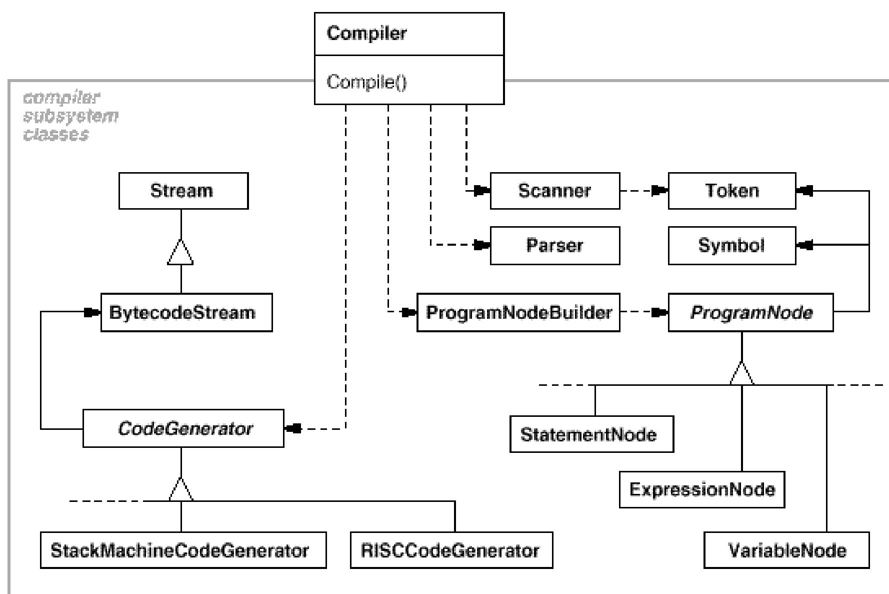

Figure 3.3: Diagram of a compiler facade, taken from (Gamma et al., 1994b)

In this diagram the compiler acts as an higher-level interface - a facade - that acts a shield that stands in between a client and all the subsystem classes. This way, if the client needs to use any of these classes like the *Scanner* or the *Parser*, it can call on the *Compiler* facade without having to go through the complexity of the subsystem.

3.2.4 The Decorator Pattern

The last structural pattern apt for the NOVAAS Connector is the decorator pattern. This pattern's purpose, as stated by (Gamma et al., 1994c) is to enable additional actions in an object, by attaching itself dynamically. A decorator can be used as an alternative to sub-classing as a method of broadening functionality.

By using this pattern, the user can add new features without altering the core functionality of the object it is attached to.

Fig.3.4 represents a diagram of a decorator. In line with ("Design patterns - decorator pattern", 2023), the *Shape* class in turn has *Circle* and *Rectangle* sub-classes, both with method *draw()*. In this case the Decorator adds the functionality of red shapes. Instead of altering the original sub-classes, this decorator wraps the main class and adds extra utility. In addition to this, other decorators can be added on top to add even more functionality.

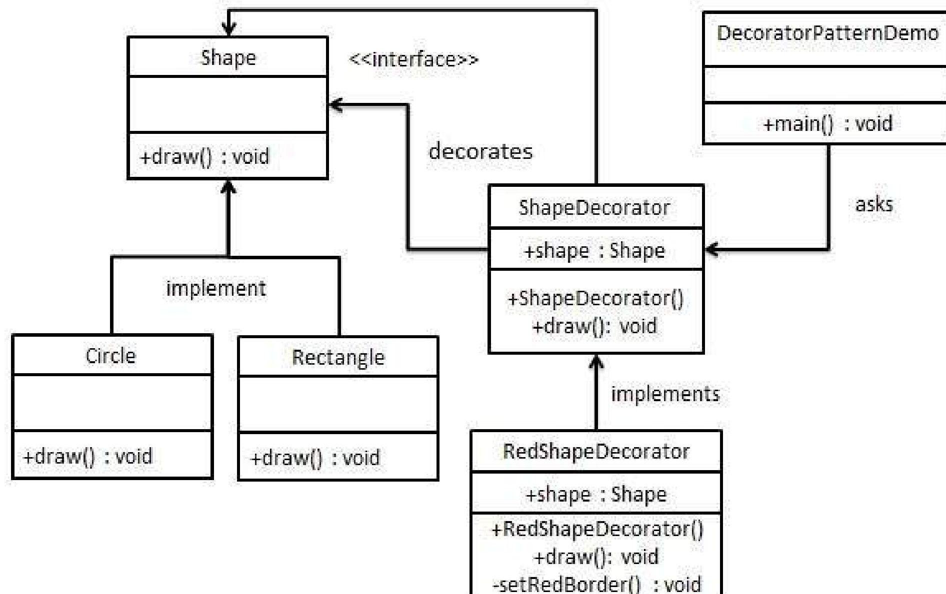

Figure 3.4: Example diagram of a decorator, taken from ("Design patterns - decorator pattern", 2023)

3.3 Choosing the most fitting pattern

After the brief introductions to the described design patterns in Subsections 3.2.1, 3.2.3 and 3.2.4, a choice has to be made regarding which framework to implement upon the

creation of the NOVAAS Connector.

The purpose of this connector is to transform an asset's properties, represented in a JSON file, into a formulaic Node-Red flow. This Connector has to work with multiple assets, that might have their working properties represented in various different ways.

For example, one asset might incorporate a temperature sensor, while another might have an operational pressure sensor. Because of this, the two JSON files where their properties are described will deviate when it comes to these sensors. In order for the Connector to work with both of these and multiple other different assets, its execution has to be as generic as possible. The end goal is to create a system that translates data from any asset into nodes in the NOVAAS, represented in a Node-Red flow.

If we consider the JSON data model as a client and its file, the interface, the Connector would bridge the gap between this interface and the Node-Red flow. The Connector would be an adapter in this scenario, by introducing the necessary "data translation" functionality without altering the JSON file. Because this system is not a complex set of various classes that need to be aggregated into one interface, the facade pattern would not apply well here. As for the decorator pattern, due to the JSON file being solely a data representation without any computing logic, this pattern would not work, as there is no core functionality to attach itself to. With this information, the most fitting pattern would be the object adapter pattern.

3.4 Applications of adapters in various real world scenarios

For this section, several applications of software wrappers were studied to find the best approach to building the NOVAAS connector.

3.4.1 Wrappers applied to legacy code

The first paper analysed, (Malinova, 2009), attempts to re-purpose native legacy code related to physics simulations. The author applies the adapter, proxy and facade design patterns to connect legacy code to Java Native Interface (JNI).

At the end of the article, the author mentions a real use case scenario of applying the adapter pattern to the simulation software platform "Plasimo". Because this software is written in C++, an adapter is necessary to translate all the calls to Java. This wrapper was implemented with JNI, in order to make a class library that bridges the required functions implemented in "Plasimo".

With this study it is clear to see that one of the advantages of the adapter design pattern is the easy implementation of legacy code into newer platforms, without having to alter any of the original library code.

3.4.2 Developing a Cloud Environment with Software Design Patterns

The second paper in analysis was authored by (Markoska et al., 2016). This paper describes the creation of an Interoperable Cloud Environment with the help of design patterns, namely the adapter pattern. The authors present the workings of the *Eucalyptus* and *OpenStack* open-source cloud API. They then attempt to create a wrapper that connects both of these APIs. This wrapper's intention is to implement a web service that allows communication to occur between both cloud services.

In Fig.3.5, the class diagram of the entire cloud API environment is demonstrated, including both classes represented by a design pattern - the factory pattern and the adapter pattern. Both *OpenStack* and *Eucalyptus* APIs inherit the *CloudAPI* generic interface class. These API classes are also connected to the *CloudAPI* factory, which implements the method *getAPI()*.

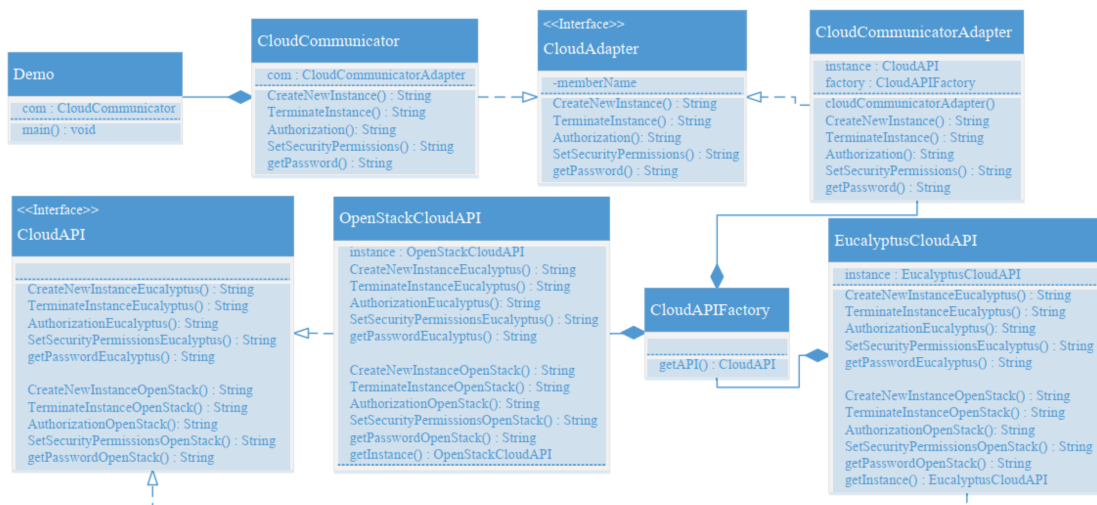

Figure 3.5: Class diagram of the cloud environment model, taken from (Markoska et al., 2016)

The *CloudAdapter* interface is implemented by the concrete *CloudCommunicatorAdapter* class, which adds all the functionality of communication with both cloud APIs. This communication is made through an input string, instead of singular functions for each cloud. The interface also implements a *CloudCommunicator* class responsible for answering method calls from the *Demo* class.

With this execution style, it is possible to integrate APIs of multiple different cloud services, with minimal changes to the preexisting logic. Therefore, with the help of the adapter pattern, the authors of this paper have made an interoperable cloud environment, where a single interface interacts with various cloud APIs. In addition to the *Eucalyptus* and *OpenStack* APIs, the adapter class may extend to other cloud services.

3.4.3 Building an Interface for health monitoring devices, using the wrapper pattern

Another use case for the adapter pattern is exemplified by (Kobylarz and Danda, 2013). In this paper the authors justify the use of a device wrapper, arguing for the need for an interoperable environment in the field of bluetooth-based health monitoring devices. According to the author, in this field, multiple vendors offer a variety of devices that often do not carry the necessary tools to allow straightforward communication between themselves. Because of this, the authors manifest the importance of a common interface for these devices.

The execution of this system involves the use of an adapter service, implemented in Java, as to allow support for different data protocols. The author used multiple different protocols for communication, including Bluetooth, Ant+ and Wi-Fi. The way this service handles all the different data protocols is very relevant in the context of this dissertation, given the points discussed in Section 3.3. According to the author, the application changes how it responds to these different protocols based on the connected equipment.

Furthermore, as seen in Fig.3.6, the class *DeviceWrapper* allows for concurrent transmission of data from different devices. This service is multi-threaded, meaning each adapter is executed in a single thread, demonstrated in the *ConnectionThread* class.

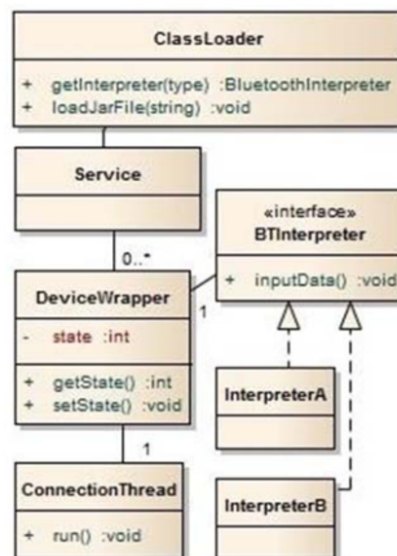

Figure 3.6: Classloader diagram, taken from (Kobylarz and Danda, 2013)

In the end, the author successfully tested this application in multiple different Android devices and several different Bluetooth devices. This might reveal a positive look into how the NOVAAS Connector can be implemented.

3.4.4 IoT Framework using wrappers

This paper analyses a multitude of frameworks in the IoT space and provides a solution which implements the wrapper pattern. The authors (Martino et al., 2016), emphasize the rising popularity of IoT and the lack of sensor interoperability. A problem also noted by the author of the paper analysed in Subsection 3.4.3, where in this case, the topic extends to sensors of all types. Although the paper is relevant to this dissertation in broad terms, I will focus on studying the method (Martino et al., 2016) used to create and integrate wrappers in this specific framework.

This paper's main application of the adapter pattern comes from the module named *Wrappers Repository*, represented in the schema of this framework. Its function is to store all the preexisting solutions to the sensors previously mentioned in the article. Therefore, this system is built in such a way that if a new sensor is connected to the framework, this repository is probed for any solution specific to this sensor.

In Fig.3.7, a model of the Wrappers Repository is presented. The principal property of this repository is to link the sensor wrappers and code snippets to the contextual descriptions in the Sensors Ontology. The code snippets are programmed in C/C++ or Java libraries.

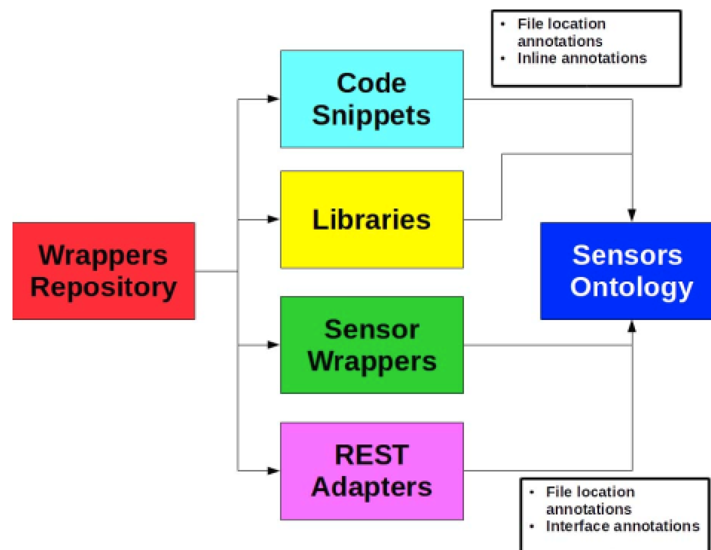

Figure 3.7: Wrappers Repository, taken from (Martino et al., 2016)

This method of implementing wrappers is fitting for the creation of the NOVAAS Connector, considering the multitude of assets that can be connected to NOVAAS.

3.5 Semi-automatic code generation

With the groundwork of building an adapter model, the next step is to study how this adapter can be automatically or semi-automatically generated. The main goal of the

solution is to provide an automated connector that can interface with any asset, achieving interoperability. This is done to eliminate the manual labor involved in creating individual connectors for each different asset.

3.5.1 Automatic generation of OPC UA connectors

The first article related to this topic is authored by (Lyu et al., 2022), and it describes the implementation of automatically generated Function Blocks for an OPC UA server. The main contributions of this paper are centered around simplifying the connection between an industry standard control application and a digital-twin factory by automating this process.

The relevancy of this paper towards this dissertation comes from the method used to implement the connections between the system files using Extensible Markup Language (XML) and JSON specifications. The authors use an *XML2JSON* parser to turn the XML system file into a JSON object. The algorithm used to generate the new JSON object is demonstrated in Listing 3.

```
1  Result: Generated sys file object
2  /*Convert XML system files to JSON system files*/
3  sysJson = JSON.parse(original sys file);
4  FB[] = sysJson.FB //Get the list of all Function Blocks;
5  while FB[] has next element do
6      /*If this FB is I/O-accessing*/
7      if element.type == IX OR element.type == QX then
8          IOFB.push(element); /*Store it in I/O list*/
9      end
10 end
11 while IOFB[] has next element do
12     /*Copy all metadata from I/O FBs and paste to generated OPC UA FBs*/
13     opc_param = genParam(element);
14     opc_name = genName(element);
15     opc_type = genType(element);
16     while element.EC[] has next eveCon do
17         opc_EC.push(genEveCon(eveCon)); /*Copy and paste event connections*/
18     end
19     while element.DC[] has next dataCon do
20         opc_DC.push(genDataCon(dataCon)); /*Copy and paste data connections*/
21     end
22 end
23 /*Generate JSON FB from parsed metadata*/
24 newSysJson = genSysFileObject(opc_name, opc_type,
25 opc_param, opc_EC[], opc_DC[]);
26 writeToXML(newSysJson);
```

Listing 3: Algorithm used by (Lyu et al., 2022)

According to this algorithm, the application takes the parsed JSON file code and builds two Function Blocks. While the first one is populated with the entire system file, the second one only takes elements that access the I/O. The second part of this algorithm manages all the I/O connections present in the latter Function Block. Finally, a new JSON file is generated from the parsed metadata and converted back to XML.

3.5.2 Generating Node-Red flows in an IoT context

The second paper studied, authored by (Clerissi et al., 2018), proposes an approach for building and testing a Node-Red implementation of an IoT. It starts by modelling the behaviour of the system using an Unified Modeling Language (UML) diagram for each Node-Red node. This task has to be done manually, requiring coordination between the developers in charge of modelling and developing the system.

After obtaining the UML model, it is possible to generate the Node-Red flows that represent these models. According to the authors, this is done by transforming the XML file (representing the UML model) into a JSON representation. This transformation is then automated, based on the algorithm described in section 3.4 of the paper.

3.6 Final considerations

After the study done on the theoretical approaches to building the solution, the next step is to build the solution with the help of these approaches. In Fig.3.8, the theoretical adapter structure that represents the NOVAAS Connector is demonstrated. The Connector acts as an adapter to the asset representation, providing an interface to the NOVAAS platform.

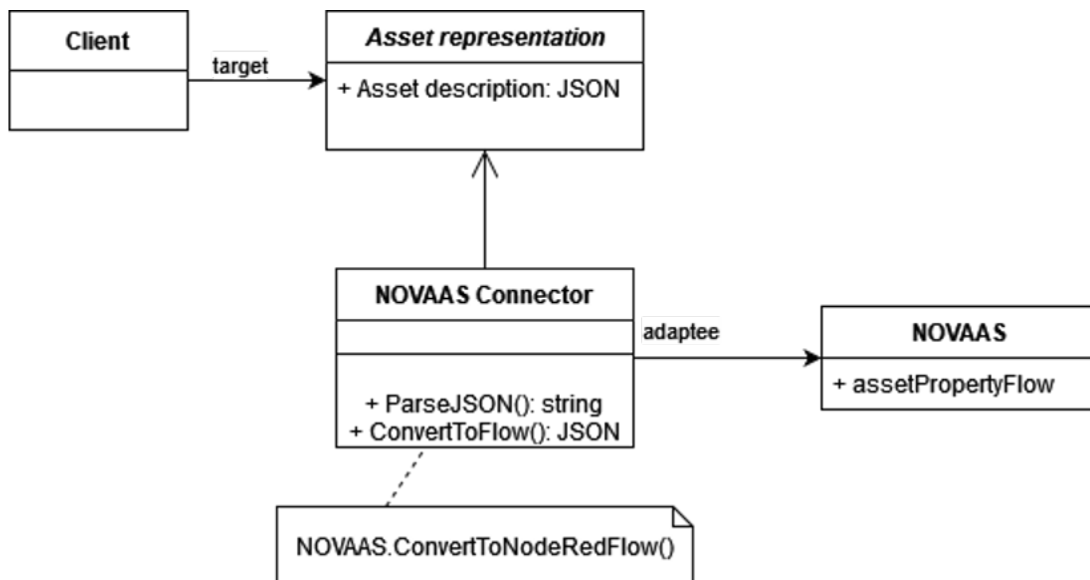

Figure 3.8: NOVAAS Connector structure

The upcoming Chapter will present an analysis on the construction of the Connector.

4

NOVAAS Connector

This Chapter is divided into two sections: The architectural structure of the proposed solution and its implementation.

4.1 Connector Structure

In order to understand the purpose of this solution, it is necessary to look at the structure of the environment where it is intended to be implemented. From Fig.4.1, we gather that the connector will act as an adapter for the assets.

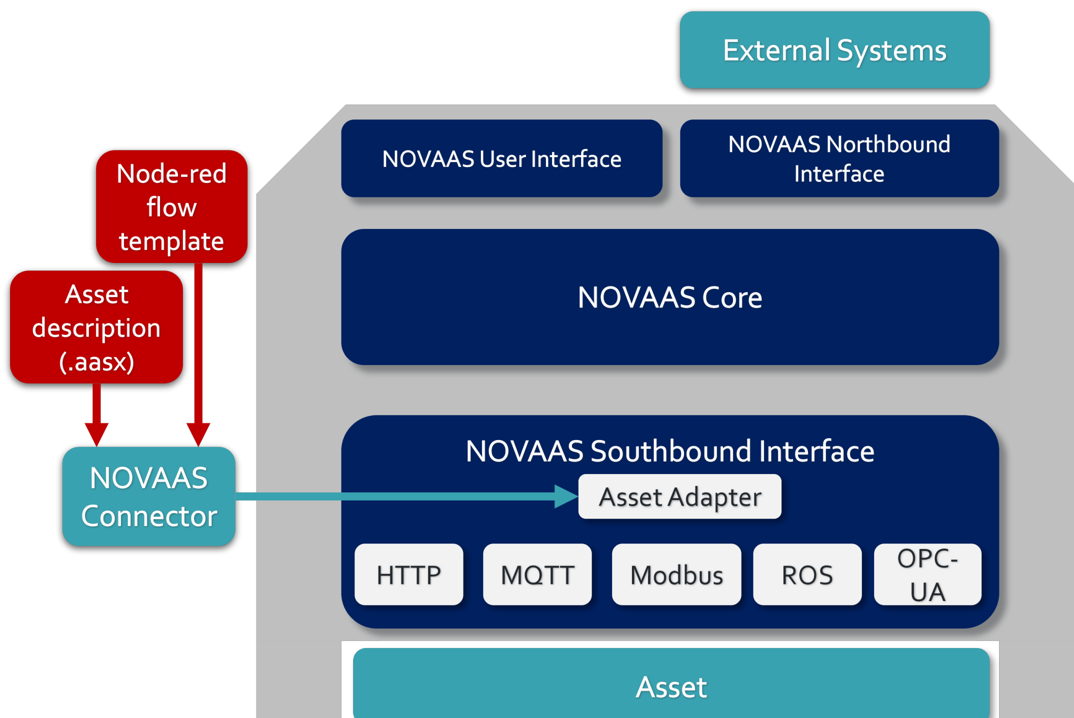

Figure 4.1: Structure of NOVAAS including the Connector, adapted from (Di Orio, 2021)

Its intent is to connect the asset to the NOVAAS platform, allowing it to communicate the necessary data and translate it into a compatible representation. The Connector outputs

the data into the Operational Data section of the NOVAAS platform, acting as an adapter for the asset's properties. Creating a connector that automates the linkage between an asset and the NOVAAS is essential in enhancing interoperability while drastically reducing the need for manual intervention. This solution acts as a bridge, seamlessly integrating different assets with various and diverse properties.

The following section will go over the practical implementation of the proposed solution.

4.2 Solution Implementation

As mentioned in Chapter 1, the proposed solution aims to facilitate the creation of a specific Node-Red flow, located on every iteration of NOVAAS. The Node-Red flow in question is the Operational Data flow, which must include the properties of the asset submodel.

In order to generate this flow, it is necessary to first gather all the relevant properties of the asset, which are located in the JSON within the .aasx file. After retrieving all the necessary properties, the data is processed and stored locally.

The Node-Red flow template - a JSON file - is then read and altered with JSON objects representing the properties gathered in the previous step. The Node-Red flow is now complete and can be imported into the NOVAAS. A diagram representing the flow of the described steps is represented in Fig.4.2.

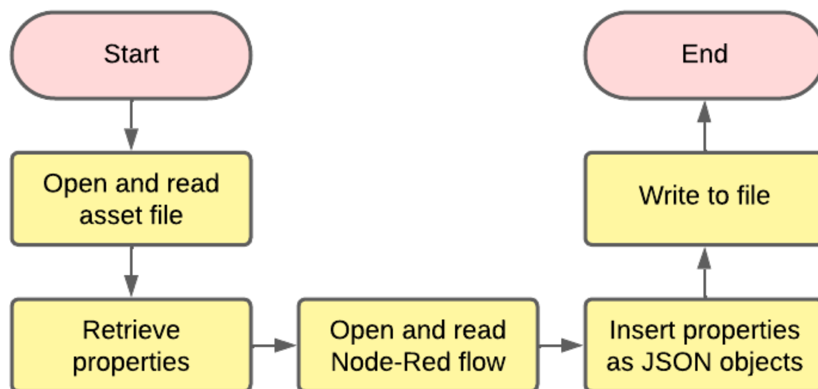

Figure 4.2: Flow diagram of tool's tasks

In an effort to create the most compatible adapter, three versions of the tool where created. Other than improving the modularity of the program, this technique also made it simpler to create a more complete tool. Each version adds more complexity and completeness than the last.

The program language chosen to build this software was Python, due to its simplicity in handling data engineering issues and the extensive availability of third-party libraries.

There were three libraries used: the "json" library; the "tkinter" library; and the "secrets" library.

The first library's usefulness is apparent, given both the input and output products constitute a JSON file. The second library allows for interfacing with the user's Operating System, specifically used, in this instance, for opening the file explorer and selecting the desired JSON files. The last library adds the ability to generate secure random numbers for the purpose of unique identification of nodes. The NOVAAS Connector is available at (Pais de Sá, 2023a).

4.2.1 First version

All versions of the connector follow the full task diagram represented in Fig.4.2, the difference in each version resides in the final Node-Red flow. In Listing 4, a fraction of the initial part of the code is shown.

```
1  root = tk.Tk()
2  root.withdraw()
3  filepath_assetjson = tk.filedialog.askopenfilename(initialdir="./v1",
4  title="Asset JSON file:")
5
6  f = open(filepath_assetjson)
7  data = json.load(f)
8
9  for i in range(0,len(data["submodels"])) :
10     if data["submodels"][i]["idShort"] == "OperationalData":
11         aas_opdata = data["submodels"][i]["submodelElements"]
12         break
13
14 for i in range(0,len(aas_opdata)) :
15     if aas_opdata[i]["modelType"]["name"] == "SubmodelElementCollection":
16         coll_list.append(aas_opdata[i]["value"])
17
18     if aas_opdata[i]["modelType"]["name"] == "BasicEvent":
19         for j in range(2,len(aas_opdata[i]["observed"]["keys"])):
20             if j == len(aas_opdata[i]["observed"]["keys"]) - 1:
21                 evt_name += aas_opdata[i]["observed"]["keys"][j]["value"]
22                 + "Evt"
23             else:
24                 evt_name += aas_opdata[i]["observed"]["keys"][j]["value"]
25                 + "."
26
27         n_properties_w_event +=1
28         evt_list.append(evt_name)
29         evt_name = ""
```

Listing 4: Initial snippet of first version script

Following the script step-by-step, we start by opening the JSON file using the "tkinter" library. This file is available at (Di Orio, 2023), in any of the projects, inside the "files" directory - designated as *"model.aasx"*. This file can, in some cases, have over two thousand lines of code. For this reason, most of it is abbreviated in the following Listings. For a better understanding of this file's structure, I recommend the reader to download the file at any of the projects located in the referenced link.

The file explorer will prompt the user to select the asset JSON file. This file is a representation of an asset situated inside a previously provided .aasx file. After this, we use the "json" library to load the JSON file into a variable named *data*.

We can then use this variable to parse the JSON file, looking for the desired properties of the asset. This is done in two "for" loops, demonstrated in Listing 4 - lines 13 and 18. To better understand these loops, a fragment of the JSON file of a test asset being parsed is shown in Listings 5 and 6. For the sake of brevity, only the relevant JSON objects are demonstrated. Each of these contain multiple elements, describing the asset in extensive detail.

```
1 {
2     "assetAdministrationShells": [...],
3     "assets": [...],
4     "submodels": [...],
5     "conceptDescriptions": [...]
6 }
```

Listing 5: Top-most segment of JSON asset file

```
1  {
2     "submodels": [
3       {...},
4       {...},
5       {...},
6       {
7       "idShort":"OperationalData",
8       "modelType": {
9           "name": "Submodel"
10        },
11      "submodelElements":[...]
12      }
13    ]
14 }
```

Listing 6: Inside *"submodels"* JSON object

The first loop searches for the *"OperationalData"* section inside the *"submodels"* object. This section contains all of the properties and events that need to be extracted. After saving

the *"submodelElements"* object to the variable, *aas_opdata*, we can then do a surface-level search for all of the properties in the file. This is done in the second loop demonstrated in Listing 4 - line 18.

All the necessary properties are located in the *"BasicEvent"* model type, however these can be found inside a *"SubmodelElementCollection"* which in turn, can also be inside another object of that type. Because there is no limit to how many nested Collections might exist in an asset file, it is necessary to build an algorithm that can find all of these objects, regardless of size. Therefore, a third loop was implemented for the purpose of parsing all the existing Collections, nested or otherwise. It is important to note that the main goal of this algorithm is to find the *"BasicEvent"* model type, which can be found at any point of the search, inside or outside of a Collection.

To reiterate, in the second loop, when a *"SubmodelElementCollection"* is found, it is appended to a list - the *coll_list* variable. This list will then be used in the third loop, where, one by one, these Collections will be parsed. If no *"SubmodelElementCollection"* is found in the asset file, the properties are stored in another list - the *evt_list* variable.

The third and final loop of the third step in Fig.4.2 is shown in Listing 7.

```
1  while len(coll_list) != 0:
2      initial_len = len(coll_list)
3
4      for i in range(0,len(coll_list)):
5          for j in range(0,len(coll_list[i])):
6              if coll_list[i][j]["modelType"]["name"] == "SubmodelElementCollection":
7                  coll_list.append(coll_list[i][j]["value"])
8
9              if coll_list[i][j]["modelType"]["name"] == "BasicEvent":
10                 for k in range(2,len(coll_list[i][j]["observed"]["keys"])):
11                     if k == len(coll_list[i][j]["observed"]["keys"])-1:
12                         evt_name += coll_list[i][j]["observed"]["keys"][k]["value"]
13                         + "Evt"
14                     else:
15                         evt_name += coll_list[i][j]["observed"]["keys"][k]["value"]
16                         + "."
17
18                 n_properties_w_event +=1
19                 evt_list.append(evt_name)
20                 evt_name = ""
21
22     del coll_list[:initial_len]
```

Listing 7: Third loop, which parses all nested Collections

Instead of searching the top-most JSON object *"submodelElements"* like the previous loop, it searches all the *"SubmodelElementCollection"* that were found and appended to the list (*coll_list*). At the same time, if this loop finds another Collection, it is added to the

same list. This final loop stops only when no more Collections are found.

Having parsed and stored all the events and properties of the asset, we can now move on to steps 3, 4 and 5 of Fig.4.2. The final Node-Red flow is also configured as a JSON file, however, it is structured differently from the asset JSON file. A snippet of the this file is shown in Listing 8, as an example. In Fig.4.3 the visual representation of this node is demonstrated.

```
{
    "id": "806b9c4f8b70c7cd",
    "type": "link in",
    "z": "128ab3134a54f6b4",
    "name": "trigger_in_1",
    "links": [
        "1103084ec6579d65"
    ],
    "x": 185,
    "y": 480,
    "wires": [
        [
            "60fc493c21196b3f"
        ]
    ]
},
```

Listing 8: Snippet of Node-Red flow node in JSON

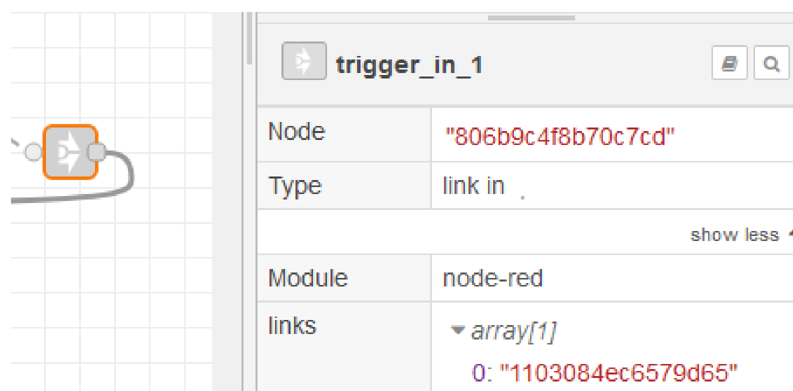

Figure 4.3: Node from Listing 8 in Node-Red flow

Listing 8 represents a node in the Node-Red flow. A flow is comprised of multiples of these dictionaries, each with different configurations, portraying different node types. Looking at this specific example, the first element in this dictionary is an *"id"*. This *"id"* is defined by a unique, 8-bit hexadecimal value, which is present on every node. This value is also used to indicate a connecting node. The *"wires"* key contains a different hexadecimal value, which means that, in this case, the node with the *"806b9c4f8b70c7cd"*

"id" is connected to the node with the *"60fc493c21196b3f" "id"*. Also present in most nodes is the *"x"* and *"y"* keys. These represent the location of the node within the visual flow.

This node is part of a larger group of nodes that are implemented for every property in the asset. In Fig.4.4, this group of nodes is displayed.

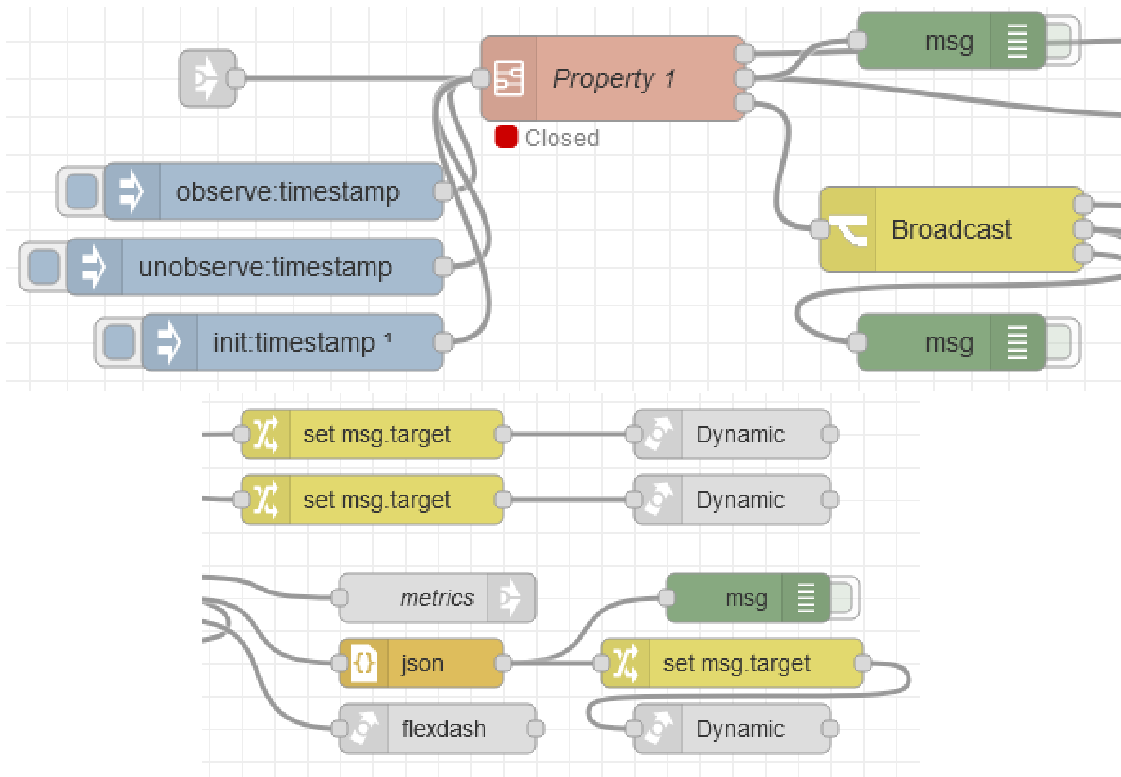

Figure 4.4: Node group connected to the first of Property, sample asset

The first version takes an incomplete Node-Red JSON file, and writes the configuration of the Property nodes with the events and property names. A picture of this configuration is shown in Fig.4.5.

Edit subflow instance: Property handler

Edit subflow template | Delete

Properties

Name	Property 1
PropertyName	CurrentOperatingPressure
PropertyLink	http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CurrentOperatingPressure
HistoryLength	14400
PropertyLinkEvt	http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CurrentOperatingPressureEvt

Figure 4.5: Property 1 configuration, sample asset

The *PropertyName*, *PropertyLink*, *HistoryLength* and *PropertyLinkEvt* elements were filled in with the data gathered from the asset file. This is done for all properties in the Node-Red flow. The logic for configuring these settings for each property is demonstrated in Listing 9.

```
for i in range(0,len(flow_data)):

    if "type" in flow_data[i].keys():

        if flow_data[i]["type"] == ("subflow:" + propertyhandler_id)
        and ("Property " in flow_data[i]["name"]):
            n_properties_in_flow += 1

            if len(flow_data[i]["env"]) > 0:
                continue

            property_dict = {}
            property_dict["name"] = "PropertyName"
            property_dict["type"] = "str"
            property_dict["value"] = evt_list[j].replace("Evt","")

            propertylink_dict = {}
            propertylink_dict["name"] = "PropertyLink"
            propertylink_dict["type"] = "str"
            propertylink_dict["value"] = aas_id + "/OperationalData/"
            + evt_list[j].replace("Evt","")

            historylength_dict = {}
            historylength_dict["name"] = "HistoryLength"
            historylength_dict["type"] = "num"
            historylength_dict["value"] = history_length

            propertylinkevt_dict = {}
            propertylinkevt_dict["name"] = "PropertyLinkEvt"
            propertylinkevt_dict["type"] = "str"
            propertylinkevt_dict["value"] = aas_id
            + "/OperationalData/" + evt_list[j]

            flow_data[i]["env"].insert(0,property_dict)
            flow_data[i]["env"].insert(1,propertylink_dict)
            flow_data[i]["env"].insert(2,historylength_dict)
            flow_data[i]["env"].insert(3,propertylinkevt_dict)

```

Listing 9: Section of code depicting the creation of JSON objects in final Node-Red file

After opening the Node-Red flow JSON file, a ”for” loop looks for the section of the file containing the ”Property handler” node. This is where each property node is configured,

as shown in Fig.4.5. After finding the correct JSON objects, a second loop - demonstrated in Listing 9 - creates a dictionary for each element to be configured in each property. These are then inserted into their respective nodes and written to the file after every property is configured.

4.2.2 Second version

This version performs the same logic as the previous version with added functionality. Where the first version only altered the configuration of the property nodes, this one aims to create all the nodes connected to the main property node - depicted in Fig.4.4, in red. After performing the initial algorithms to get the asset properties and inputting the data into the property nodes, the rest of the nodes are created for each property. This is assuming that the input Node-Red JSON is only populated by the property nodes.

The snippet of code shown in Listing 10 demonstrates the creation and insertion of one of the nodes.

```
change_node = {
    "id": flow_data[i]["wires"][0][0],
    "type": "change",
    "z": flow_id,
    "name": "",
    "rules": [
        {
            "t": "set",
            "p": "target",
            "pt": "msg",
            "to": "southbound.updateSubscriptions",
            "tot": "str"
        }
    ],
    "action": "",
    "property": "",
    "from": "",
    "to": "",
    "reg": False,
    "x": flow_data[i]["x"] + 400,
    "y": flow_data[i]["y"] - 40,
    "wires": [
        [
            secrets.token_hex(8)
        ]
    ]
}
```

Listing 10: Creation and connection of the change node to the Property node

This section of the code generates the twelve nodes that follow the property node (not shown). Among these are the *broadcast*, *link* and *debug* nodes. The connection between nodes is done by giving this node the same *id* found in the *"wires"* section of the previous node. In this case the previous node is the property 1 node, which contains three outputs with one wire in the first and third outputs and two wires in the second output - as seen in Fig.4.4. Because this change node is connected to the first output of the property node, its *id* is located in the initial position of the *"wires"* dictionary - line 3 of Listing 10.

Further reviewing the main components in this object, we find the *'z'* key has the value of the *id* of the flow, which in this and in all instances of these nodes, indicates the *OperationalData* tab of the Node-Red platform. Another important aspect of this object is its location within the flow, signified by the *'x'* and *'y'* keys. In this version, because the main property nodes are already implemented and in the right place, the position for all the following nodes is simply incremented or decremented by a fixed value. This makes all nodes fall into a fixed pattern, only being offset by the initial Property node.

Lastly, the *"wires"* key contains all the attached nodes to this one. Because we are creating these nodes one by one, from first to last, their *ids* are not set yet. To do this we must create a unique identifier for every node. This is done by invoking the *secrets* library and using the function demonstrated in line 25 of Listing 10 to create an 8-bit hexadecimal value. The node that we wish to connect to this one, will have the same id as the one found on this dictionary. The other "key, value" pairs are filled in with the data from a template. These values are constant and do not need to be altered per property.

4.2.3 Final version

The third and final version implements the previous versions' functionalities along with the added task of creating the Property nodes. The input JSON file of the Node-Red flow follows a basic template that is unchanged for any asset. This template is provided by the developer, and is also available at (Di Orio, 2023). This makes the final version of the tool the most generic one, providing the most usability and versatility. The extra functionality brings two new challenges.

First, the *trigger_out* node needs to be populated with links associated to each property *trigger_in* node. In Fig.4.6, a *trigger_out* node is shown, in this case, with seven properties attached. Secondly, the Property nodes must be created and placed in a clean and presentable format. In this case the nodes will be placed in two rows with sufficient space around to improve readability. For example, if there are eight properties, this script will create two rows with four property groups on each row.

The script follows the same logic as the two previous versions, however, we must first create the property node handler. This handler represents the structure of every property node, which is formatted as a subflow with eleven nodes - shown in Fig.4.7. Each node in this subflow must be created and populated before the main property configuration loop. This will act as a template for every Property node, therefore, it only needs to be

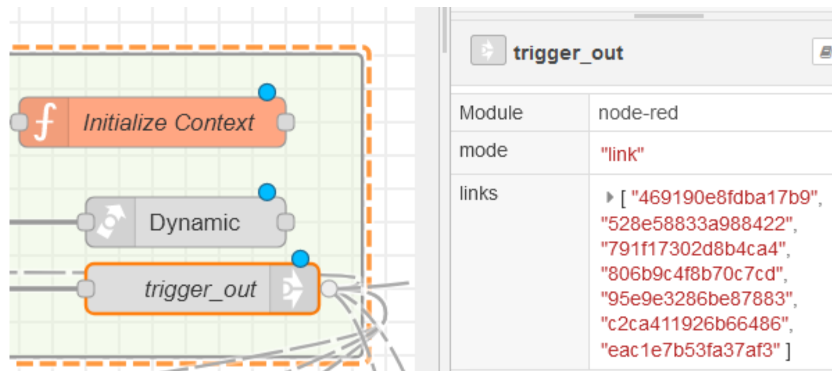

Figure 4.6: Example of a *trigger_out* node with seven properties linked

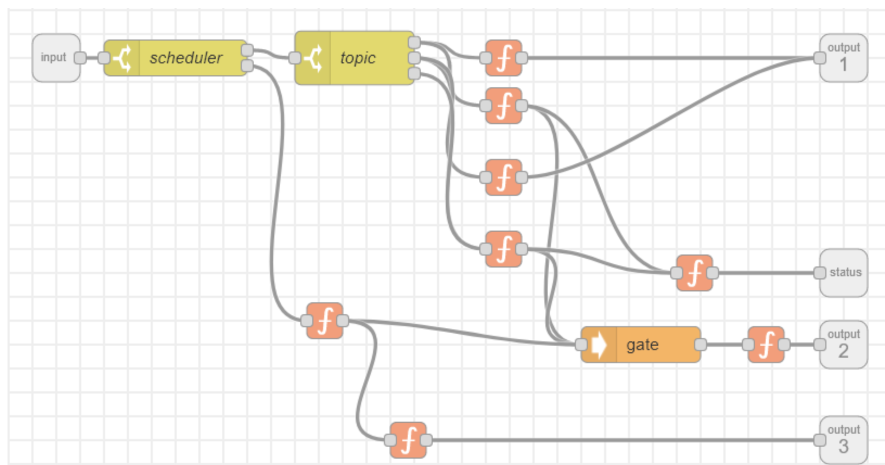

Figure 4.7: Property handler subflow template

created and inserted into the JSON file once. A section of the code that performs this task is presented in Listing 11.

After making this template, we can now move on to creating and placing the *trigger_in*, *timestamp* and *Property* nodes - shown in Fig.4.4 to the left of and including the Property node. It is important to note that each *trigger_in* node *id* needs to be present in the *trigger_out "links"* array. Because the flow template leaves this array empty, we now must populate it with new *ids* that will correspond with *trigger_in* nodes. This implementation is shown in Listing 12.

First, the *trigger_in* id is created and added to the *"links"* array of the *trigger_out* node. Then the *trigger_in* node is created and appended to the JSON. This is all done in a loop, for each property. Following this, we can now create the remaining nodes - the *timestamp* and *Property* nodes. It is at this step that the position of each node is also set. Listing 13 shows the simple logic that defines the x and y values for every *trigger_in* node. The *x_pos* variable has two possible values, while the *y_pos* variable iterates based on how many properties exist.

```
1       property_handler = {...}
2       ...
3       node4 = {
4           "id": "f894203f2fd3ac35",
5           "type": "gate",
6           "z": "41b3a1439ccec2c1",
7           "name": "",
8           "controlTopic": "control",
9           "defaultState": "closed",
10          "openCmd": "open",
11          "closeCmd": "close",
12          "toggleCmd": "toggle",
13          "defaultCmd": "default",
14          "persist": False,
15          "x": 1490,
16          "y": 800,
17          "wires": [ ["e62907e7bcbce663"] ]
18      }
19      node5 = {...}
20      ...
21      flow_data.insert(0,property_handler)
22      flow_data.insert(1,node1)
23      ...
```

Listing 11: Abbreviated section of the code tasked with creating the property handler subflow template

```
1  if "name" in flow_data[i].keys() and flow_data[i]["name"] == "trigger_out":
2              for p in range(n_properties_w_event):
3
4                  property_trigger_id = secrets.token_hex(8)
5                  flow_data[i]["links"].append(property_trigger_id)
6                  trigger_in_node = {
7                      "id": property_trigger_id,
8                      "type": "link in",
9                      "z": flow_id,
10                     "name": "trigger_in_" + str(p+1),
11                     "links": [ flow_data[i]["id"] ],
12                     "x": x_pos,
13                     "y": y_pos,
14                     "wires": [ [secrets.token_hex(8)] ]
15                 }
```

Listing 12: Snippet of the code that generates the *trigger_in* node

```
#if current property number is uneven, the next is even,
#meaning only x position changes
#else next is uneven,
#meaning x position goes back to initial and y position increments

#like so:
# (x,y)                  (x+i, y)
# (x, y+j)               (x+i, y+j)
# Property 1             Property 2
# Property 3             Property 4

if (p+1)%2 == 1:
    x_pos = 1725 #x_pos + 1540
else:
    x_pos = 185 #x_pos - 1540
    y_pos = y_pos + 320
```

Listing 13: Logic in creating the position of the nodes

After this section, the rest of the code is identical to the previous version.

In the next Chapter, the three main test cases will be shown, giving a better view of the tool in practice.

5

Tests and Results

For the creation of this tool, three distinct cases were tested. These are based on the three different implementations of the NOVAAS, available to the public in the GitLab repository (Di Orio, 2023). Each represent a functional model, making them the best test cases to use in the creation of the solution. In this chapter, these tests will be shown and analysed.

This tool is automating a manual task carried out by the developer. Due to the fact that it involves manually placing nodes depending on the number of properties and events present in the asset, the time and effort is somewhat variable and hard to quantify. Because of this, a qualitative results comparison method was chosen, in the form of a questionnaire. This questionnaire was given to a user selected by the developer, after using this tool. The results of this questionnaire act as a metric to classify the usefulness and performance of the tool. The results will be demonstrated at the end of this chapter. If the reader wishes to review in detail and compare along with the tests that will be presented, the tool is available at (Pais de Sá, 2023a) and the questionnaire at (Pais de Sá, 2023b).

5.1 Main NOVAAS Implementation Test

The first test is done on the main NOVAAS implementation located in GitLab. This implementation represents a stock NOVAAS, and the asset being administered is a compact cylinder DPDM, made by Festo ("Compact cylinder", 2023). In Figures 5.1 and 5.2, this NOVAAS integration along with the asset in question are demonstrated.

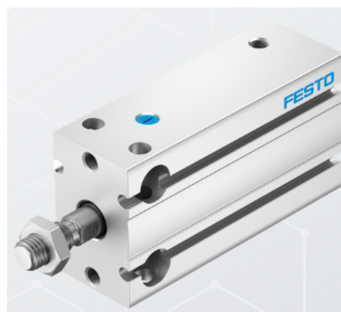

Figure 5.1: Picture of the physical asset

Figure 5.2: Information section of the NOVAAS compact cylinder application

Fig.5.3 shows the environment tab of the NOVAAS UI. These four Submodel *idShort* sections are extracted directly from the asset's JSON file, demonstrated in the previous chapter. Only the *OperationalData idShort* is relevant in the creation of this tool. All the necessary properties and their events are present under this tab. In this specific case the asset contains seven properties, each with an event associated - totaling out to fourteen elements.

By running the script, we can identify these properties, as demonstrated by Fig.5.4, which shows the print of the list of properties gathered. All the properties are in the form of events, as seen by the *Evt* keyword at the end of each element. This makes the property node configuration step easier, by just removing the three characters at the end for the property names and keeping the event names unchanged.

After running the script, the resulting Node-Red flow is integrated into the NOVAAS *OperationalData* tab. This test passed with no errors, by populating the Node-Red flow with the seven properties and events present in the asset. In Appendix A, Section A.1 the resulting Node-Red flow tab is shown, along with each property node configuration. Looking at the full implementation and the JSON file provided by this tool side by

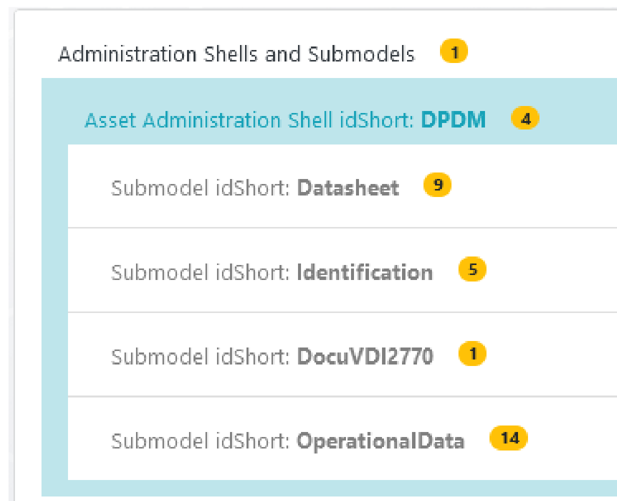

Figure 5.3: Environment tab of this application

```
PROBLEMS   OUTPUT   TERMINAL   ...        Python
PS D:\novaas\app_final\novaas_connector> & C:/Python310/python.exe d:/n
ovaas/app_final/novaas_connector/v3/connector_v3.py
Please choose the asset file (.aasx)
AAS ID: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340
['CurrentOperatingPressureEvt', 'ResponseTimeEvt', 'ValvePositionEvt',
'ResponseTimeMovingAvgEvt', 'CylinderStatusEvt', 'CycleEvt', 'ResponseT
imePredEvt']
Please choose the template flow (.json)
PS D:\novaas\app_final\novaas_connector>
```

Figure 5.4: List of all the Properties/Events contained in the asset file

side, there are no distinctions with the exception of the different randomly generated hexadecimal *ids*. It can be, therefore, considered a successful test.

5.2 PROSYS OPC-UA Simulation Test

This test is very similar to the previous one, with a change in the asset administered. A PROSYS OPC-UA Simulation Server is the asset in question, containing six properties as demonstrated in Fig.5.5.

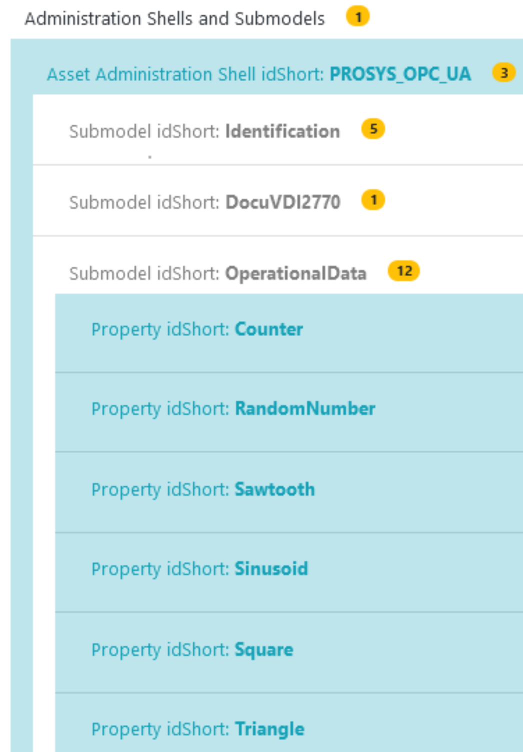

Figure 5.5: *OperationalData* tab of PROSYS OPC-UA Simulation Server

Running the script, we obtain these properties in the form of events, similarly to the previous test - shown in Fig.5.6.

```
PROBLEMS   TERMINAL   ...                 Python
_final/novaas_connector/v3/connector_v3.py
Please choose the asset file (.aasx)
AAS ID: https://example.prosys.com/ids/aas/9574_3103_7012_8644
['CounterEvt', 'RandomNumberEvt', 'SawtoothEvt', 'SinusoidEvt',
 'SquareEvt', 'TriangleEvt']
Please choose the template flow (.json)
PS D:\novaas\app_final\novaas_connector>
```

Figure 5.6: List of properties in PROSYS OPC-UA Simulation Server

In appendix A, Section A.2, the resulting Node-Red flow tab is shown, along with each property node configuration. Once again, the test is successful, showing the tool performs

properly with different quantities of properties. For both this and the previous test, the number of events matched the number of properties. This will not be verified in the next test.

5.3 NOVAAS Hydraulic Simulation Test

The final test asset is an Hydraulic Plant Simulation. This test differs from the previous two in the number of events per property. Looking at Fig.5.7 it is apparent that three *Submodel Element Collections* exist within another *Submodel Element Collection*.

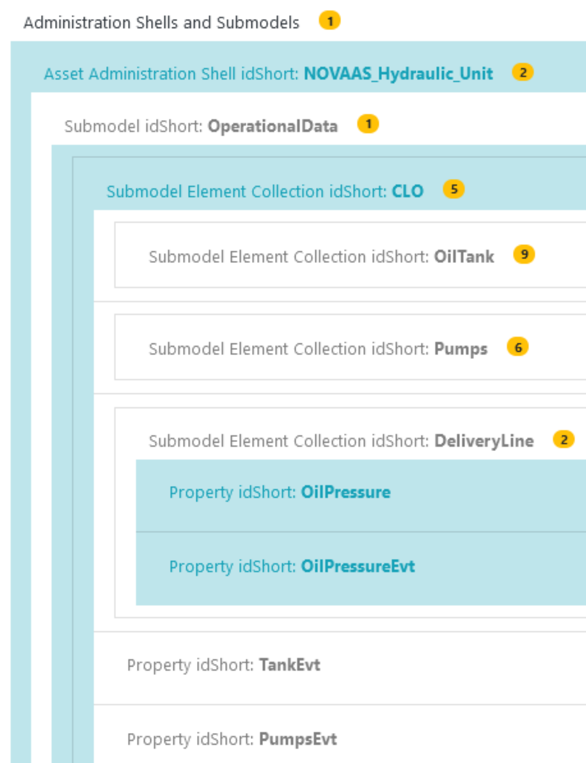

Figure 5.7: *OperationalData* tab of an Hydraulic Plant Simulation

Within each Collection under *CLO* multiple properties exist, however, for the configuration of each property node, only the events are necessary. Furthermore, these events must indicate the corresponding Collections they are included in. This is done by separating the hierarchy of Collections by periods. For example, to assign the proper link in the configuration of the *OilPressure* event, the last section of the *url* will include the designation, */CLO.DeliveryLine.OilPressure*. This indicates that the *OilPressure* Property is located within the *DeliveryLine* Collection, and this Collection is located within the *CLO* Collection.

Because only three events exist in this asset, only three property nodes will be created. The inclusion of *Submodel Element Collections* is therefore tested in this implementation. To

note that the number of nested Collections possibly present in an asset file is undefined, adding an extra layer of complexity to the parsing of events.

After running the script on this final test-case, the resulting Node-Red flow is identical to the one implemented - as one can see in Appendix A, Section A.3 - indicating the test was passed successfully.

5.4 User Experience: Results

After running the tests and implementing this tool, we can gather quantifiable metrics, such as, execution time, test coverage and the error rate. However, these are only relevant when attempting a comparison with the creation of a connector without using this tool.

Due to the fact that manually creating the connector involves placing a variable number of nodes and with different configurations, it is difficult to accurately quantify the metrics for comparison. Therefore, a qualitative analysis in the form of a questionnaire was made and given to an individual chosen by the developer and the developer himself. This inquiry was made after the use of the tool, in order to classify its performance, ease of use, run time, error rate and error prevention. The questionnaire is available at (Pais de Sá, 2023b).

When asked how easy to use the tool was, on a scale of one to five, with five being the easiest, two answers were received with the number three. When asked for comments, one user reported some difficulty understanding each step of using the tool. Another user suggested better user interaction.

The second question was related to the celerity of creating the connectors while using the tool. This question had two possible responses: ″yes″ and ″no″. When asked if using the tool was faster than creating connectors manually, both users answered yes.

The third question asked how easy it was to modify certain parameters. In this case both users answered ″yes″ - from a possible ″yes″ or ″no″ answer.

The fourth question was related to errors. When asked if any error occurred when using the tool, both users answered ″Some errors″, from a possible ″No″, ″Some errors″ or ″Many errors″ answer. When asked for the errors detected, one user indicates that when altering a parameter, no success message is returned, making it difficult for the user to see if the changes were made. Another user reported that a small part of the resulting flow was not altered when it should have been.

The fifth question was also related to errors. When asked if, in the users opinion, the use of this tool prevented human errors, both users answered ″yes″ - from a possible ″yes″ or ″no″ answer. When asked for comments on what type of errors it might prevent, both users mentioned errors in the manual creation and configuration of the property nodes.

The sixth and seventh questions were related to feedback. When asked if the user would recommend the use of this tool, both users answered ″yes″ - from a possible ″yes″, ″no″ or ″maybe″ answer. When asked for feedback for improving the tool, one

user recommended a better user interaction. Another user asked for clear log messages confirming the actions and a bug fix for the error mentioned above.

With these results, the tool created is demonstrated to work on most scenarios with the exception of a small, non-critical error that can be easily fixed. This will be done in the next release of the Connector tool. There are some points to improve, mainly when it comes to user interaction. A better interface can be implemented, along with user action feedback, in the form of message logs. These improvements are planned as future work.

6 Industrial Application

To demonstrate this tool in action, it was implemented into a real-world application. In this chapter, the real-world project will be presented along with how the tool was implemented in it.

6.1 Big Data for Next Generation Energy

The Big Data for Next Generation Energy (BD4NRG) european project aims to unlock and exploit the capability of large-scale data in the energy sector, enabling improved operation for all stakeholders in the energy value chain.

The project offers a Smart Energy reference architecture designed to harmonize with the objectives outlined in the Big Data Value Association (BDVA) Strategic Research and Innovation Agenda (SRIA), IDSA, and FIWARE architectures. This architecture ensures seamless compatibility between cutting-edge large-scale data technologies and the established standards and functional frameworks of smart grids. More on this can be found in ("Big Data Value Association Strategic Research and Innovation Agenda", 2017); ("IDSA: Driving data freedom for the whole world", 2017); and ("BDVA, FIWARE, GAIA-X and IDSA Launch an Alliance to Accelerate Business Transformation in the Data Economy", 2021).

The project advances and expands scalable solutions for preserving sovereignty through a hybrid Distributed Ledger Technology (DLT) and off-chain data governance system. It also enhances big data elastic pipeline orchestration, leverages IoT and edge AI for federated learning, and establishes a tokenized marketplace for multi-resource sharing. BD4NRG offers an open and partitioned large-scale data analytics toolbox that serves as a versatile front-end for developing all-encompassing data analysis services.

This framework's effectiveness is confirmed through the implementation of predictive and prescriptive edge AI-driven big data analytics across 12 extensive pilot projects. These pilots are executed by various energy stakeholders, encompassing the entire energy value chain.

Ultimately, BD4NRG establishes a dynamic data-centric ecosystem known as the Smart

Grid Big Data Analytics Alliance (SGBDAA). This alliance brings together emerging energy data providers, entices SMEs to offer innovative energy services with cascading funding, and verifies a hybrid energy/industry value chain that promotes collaborative B2B digital platforms ("Big Data for Next Generation Energy (BD4NRG)", 2021).

The tool developed was implemented, specifically, in large-scale pilot nine. The next section will explain this project in detail.

6.2 Large-scale Pilot Nine Introduction

Large-scale pilot nine of the Big Data for Next Generation Energy (BD4NRG) project is developed in partnership with ENERCOUTIM. It is a pilot project that aims to demonstrate the advantages of large-scale data analytics in the energy sector by developing a smart energy management system for a microgrid in Portugal, shown in Fig.6.1. This information can be found in ("Ongoing Projects", 2022).

The ENERCOUTIM pilot project is focused on developing an integrated energy management system that can optimize the use of renewable energy sources and storage systems. The system uses big data analytics to predict energy demand and supply, and to optimize the operation of the microgrid. This supports asset-level anomaly detection and prediction, facilitating real-time decision-making for coordinated flexibility provisioning. More on ("LSP9:Collaborative aggregated energy generation prediction - ENERCOUTIM", 2022).

The project aims to demonstrate that big data analytics can help reduce energy costs, improve energy efficiency, and increase the use of renewable energy sources. These encompass additional sources of renewable energy generation, water distribution, waste management, university campuses, and other municipal assets.

Figure 6.1: Solar Panel farm where large-scale pilot number nine takes place, located in Alcoutim. Taken from "ENERCOUTIM webpage", 2023

In the ENERCOUTIM Solar Demonstration Platform, various infrastructure components are interconnected, gathering data from a wide array of sources, including meteorological and radiation data, as well as data from multivariate indoor sensors and energy smart meters, among others. The primary goals of ENERCOUTIM include reducing costs associated with managing decentralized renewable energy generation assets and optimizing the management of Virtual Power Plants.

Additionally, the pilot project will set up comprehensive data repositories spanning the entire region. These repositories will serve as a demonstration of enhanced efficiency in the water-energy nexus, identify new value-generating opportunities through data analysis, and streamline the implementation of decentralized energy systems through the application of big data analytics.

6.3 NOVAAS Connector's implementation in LSP9

In accordance with the objectives of this project, the NOVAAS platform was implemented, giving access to a digital-twin for the assets present. This AAS was deployed in IT infrastructure and connected with data-loggers, providing useful analytics services.

Each solar panel in the plant represents an asset with large amounts of data available for analysis. The NOVAAS platform is then connected to each inverter, which acts as the interface with the asset. This inverter represents the solar panel and connects to a data-logger. Multiples of these data-loggers then connect to the NOVAAS platform, using MQTT and REST event-based communications. An overview of this architecture is shown in Fig.6.2.

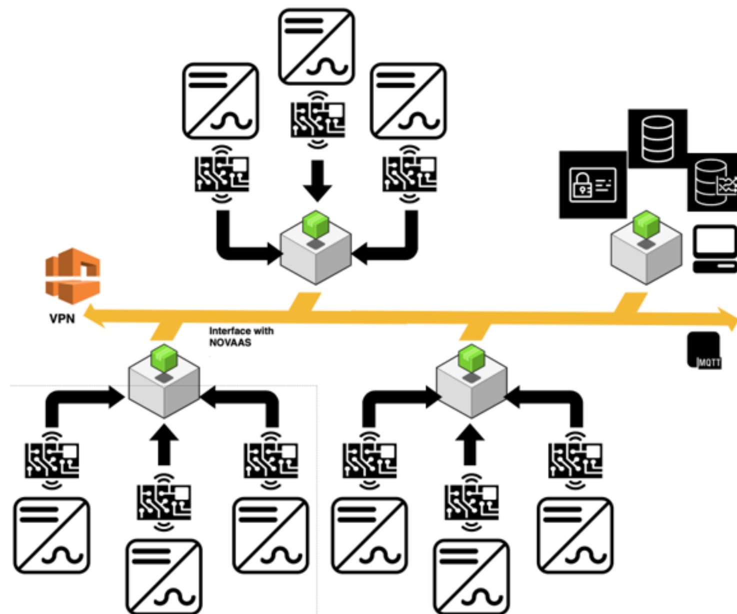

Figure 6.2: Overview of NOVAAS implementation in LSP9

The NOVAAS connector developed in this dissertation was used in the creation of the NOVAAS implementation deployed. Each inverter has an asset file that is fed into the NOVAAS Connector tool. This asset file contains all the properties of the solar panel, represented by the inverter.

7

Conclusions and future work

To conclude, this dissertation started by finding the effort of creating and implementing Software Connectors to be an issue. To try to solve this issue, the State of the Art on Software Connectors when applied to digital-twins was studied. Because of the niche topic of this dissertation, no specifically useful solutions were found to solve the problem. Consequently, only the theoretical structure approaches to building a solution were studied. In the end, after a careful study of some potentially apt software design patterns, the adapter, or "wrapper", software design pattern methodology was chosen for implementation.

As for the practical side of building a solution, the Python programming language was used, along with the necessary libraries. The script was built with the help of three tests provided by the developer of NOVAAS. The results were found to be positive, showing the usefulness and performance of the tool. The creation of this tool was successful, passing all tests. On top of this, the tool was implemented in a real-world scenario. This industrial application, used on the Big Data for Next Generation Energy, large-scale pilot number nine, proved that the solution was successful in an industrial real-world environment.

As for future work, there are two main points to be expanded upon, along with some smaller issues.

The first main point is related to the testing of the tool. Due to the lack of a significant amount of test-cases, the solution might be limited in scope. To rectify this, a test generation tool should be built. The JSON file containing the asset description follows a template with possibly nested variables that might change in size without previous indication. An asset might have one nested *SubmodelElementCollection*, zero, or sixteen. This raises the complexity and variability significantly. The tool to create these tests can be built based on the template given, however due to the time constraints, it could not be built additionally to the creation of the NOVAAS Connector.

The second point to improve is related to the specificity of the tool. The solution provided is working for the NOVAAS implementation of Asset Administration Shells. It does not guarantee function for other implementations of Asset Administration Shells like the BaSyx or FA3ST AAS. This can be expanded upon, by making the connector more

generic and available to other AAS.

The smaller issues, already mentioned in Chapter 5, are related to user interaction and a small bug fix. When it comes to user interaction, a better user interface should be developed, along with better action feedback. The second issue is in the form of a non-critical bug. This bug will be fixed in the next release of the tool.

Bibliography

Adapter. (2023). Java Design Patterns. https://java-design-patterns.com/patterns/adapter/#explanation. (Cit. on pp. 12, 13)

Atlam, H. F., Walters, R., & Wills, G. (2018). Internet of things: State-of-the-art, challenges, applications, and open issues. *International Journal of Intelligent Computing Research (IJICR)*, *9*(3), 928–938 (cit. on p. 5).

Bdva, fiware, gaia-x and idsa launch an alliance to accelerate business transformation in the data economy. (2021). FIWARE. https://www.fiware.org/news/bdva-fiware-gaia-x-and-idsa-launch-an-alliance-to-accelerate-business-transformation-in-the-data-economy/. (Cit. on p. 42)

Big data for next generation energy (bd4nrg). (2021). UNINOVA. https://www.uninova.pt/project/big-data-next-generation-energy-bd4nrg. (Cit. on p. 43)

Big data value association strategic research and innovation agenda. (2017). Big Data Value Association. https://www.bdva.eu/SRIA. (Cit. on p. 42)

Clerissi, D., Reggio, G., Leotta, M., & Ricca, F. (2018). Towards an approach for developing and testing node-red iot systems. *EnSEmble 2018 - Proceedings of the 1st ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering, Co-located with FSE 2018*, 1–8. https://doi.org/10.1145/3281022.3281023 (cit. on p. 21)

Compact cylinder. (2023). Festo. https://www.festo.com/net/pt_pt/SupportPortal/MobileDefault.aspx?q=LX7GYCYYJKR#2. (Cit. on p. 35)

Design patterns - decorator pattern. (2023). Tutorials Point. https://www.tutorialspoint.com/design_pattern/decorator_pattern. (Cit. on p. 15)

Di Orio, G. (2021). *Webinar: Asset administration shell (aas) - interoperable digital twins for industry4.0*. Youtube. https://youtu.be/jXQ8Nq4yjS4?t=1872. (Cit. on pp. 8, 22)

Di Orio, G. (2023). *Novaas collection*. https://gitlab.com/novaas. (Cit. on pp. 2, 8, 25, 31, 35)

di Orio, G., Maló, P., & Barata, J. (2019). Novaas: A reference implementation of industrie4.0 asset administration shell with best-of-breed practices from it engineering. *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, *1*, 5505–5512. https://doi.org/10.1109/IECON.2019.8927081 (cit. on pp. 2, 7, 8)

Enercoutim webpage. (2023). ENERCOUTIM. https://www.enercoutim.eu/. (Cit. on p. 43)

Facade. (2023). Java Design Patterns. https://java-design-patterns.com/patterns/facade/#applicability. (Cit. on p. 14)

Freeman, E., & Robson, E. (2020a). Chapter 7. being adaptive: The adapter and facade patterns. In *Head first design patterns, 2nd edition* (pp. 700–705). O'Reilly Media, Inc. (Cit. on p. 13).

Freeman, E., & Robson, E. (2020b). Chapter 7. being adaptive: The adapter and facade patterns. In *Head first design patterns, 2nd edition* (pp. 750–752). O'Reilly Media, Inc. (Cit. on p. 14).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994a). Chapter 4. structural patterns. In *Design patterns elements of reusable object-oriented software* (pp. 197–201). Addison Wesley. (Cit. on pp. 10, 14).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994b). Chapter 4. structural patterns. In *Design patterns elements of reusable object-oriented software* (pp. 249–251). Addison Wesley. (Cit. on p. 14).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994c). Chapter 4. structural patterns. In *Design patterns elements of reusable object-oriented software* (pp. 237–240). Addison Wesley. (Cit. on p. 15).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994d). *Design patterns elements of reusable object-oriented software*. Addison Wesley. (Cit. on p. 10).

Idsa: Driving data freedom for the whole world. (2017). International Data Spaces Association. https://internationaldataspaces.org/we/the-association/. (Cit. on p. 42)

Kobylarz, D., & Danda, J. (2013). A common interface for bluetooth-based health monitoring devices. *Proceedings - 29th Southern Biomedical Engineering Conference, SBEC 2013*, 153–154. https://doi.org/10.1109/SBEC.2013.85 (cit. on p. 18)

Lourenço, J. M. (2021). *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. https://github.com/joaomlourenco/novathesis/raw/master/template.pdf. (Cit. on p. ii)

Lsp9:collaborative aggregated energy generation prediction - enercoutim. (2022). BD4NRG. https://www.bd4nrg.eu/pilots-applications. (Cit. on p. 43)

Lynn, T., Endo, P. T., Ribeiro, A. M. N. C., Barbosa, G. B. N., & Rosati, P. (2020). The internet of things: Definitions, key concepts, and reference architectures. In T. Lynn, J. G. Mooney, B. Lee, & P. T. Endo (Eds.), *The cloud-to-thing continuum: Opportunities and challenges in cloud, fog and edge computing* (pp. 1–22). Springer International Publishing. https://doi.org/10.1007/978-3-030-41110-7_1. (Cit. on p. 5)

Lyu, T., Atmojo, U. D., & Vyatkin, V. (2022). On automatic generation of opc ua connections in iec 61499 automation systems. *Proceedings - 2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems, ICPS 2022*. https://doi.org/10.1109/ICPS51978.2022.9816861 (cit. on p. 20)

Malinova, A. (2009). Design approaches to wrapping native legacy codes. *Nauchni Trudove. Matematika, Plovdivsko Universitetsko Izdatelstvo, Plovdiv, 0204-5249*, *36*, 89–100 (cit. on p. 16).

Mandler, B., Marquez-Barja, J., Campista, M., Caganova, D., Chaouchi, H., Zeadally, S., Badra, M., Giordano, S., Fazio, M., Somov, A., & Vieriu, R. (2016). *Internet of things: Iot infrastructures: Second international summit, iot 360°*. (Cit. on p. 6).

Markoska, E., Ackovska, N., Ristov, S., Gusev, M., & Kostoska, M. (2016). Software design patterns to develop an interoperable cloud environment. *2015 23rd Telecommunications Forum, TELFOR 2015*, 986–989. https://doi.org/10.1109/TELFOR.2015.7377630 (cit. on p. 17)

Martino, B. D., Esposito, A., & Cretella, G. (2016). Towards a iot framework for the matchmaking of sensors' interfaces. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.111 (cit. on p. 19)

Node-red. (2013). Retrieved 2023, from https://nodered.org/. (Cit. on pp. 2, 8)

Ongoing projects. (2022). ENERCOUTIM. https://www.enercoutim.eu/projects/. (Cit. on p. 43)

Pais de Sá, H. (2023a). *Novaas connector github repository*. https://github.com/HenriqueSa13/novaas_connector. (Cit. on pp. 24, 35)

Pais de Sá, H. (2023b). *Novaas connector questionnaire*. https://forms.gle/TC9vBD3qhYym7Qgq9. (Cit. on pp. 35, 40)

Razzaque, M. A., Milojevic-Jevric, M., Palade, A., & Cla, S. (2016). Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, *3*, 70–95. https://doi.org/10.1109/JIOT.2015.2498900 (cit. on p. 1)

Shvets, A. (2023). *Software adapter*. Refactoring.Guru. https://refactoring.guru/design-patterns/adapter. (Cit. on p. 11)

Sisinni, E., Saifullah, A., Han, S., Jennehag, U., & Gidlund, M. (2018). Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, *14*(11), 4724–4734. https://doi.org/10.1109/TII.2018.2852491 (cit. on p. 6)

Sunyaev, A. (2020). The internet of things. In *Internet computing: Principles of distributed systems and emerging internet-based technologies* (pp. 301–337). Springer International Publishing. https://doi.org/10.1007/978-3-030-34957-8_10. (Cit. on p. 5)

Tantik, E., & Anderl, R. (2017a). Integrated data model and structure for the asset administration shell in industrie 4.0. *Procedia CIRP*, *60*, 86–91. https://doi.org/10.1016/j.procir.2017.01.048 (cit. on p. 2)

Tantik, E., & Anderl, R. (2017b). Potentials of the asset administration shell of industrie 4.0 for service-oriented business models. *Procedia CIRP*, *64*, 363–368. https://doi.org/10.1016/j.procir.2017.03.009 (cit. on pp. 6, 7)

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). Software architecture: Foundations, theory and practice. Wiley. (Cit. on p. 9).

A

Appendix

A.1 Main test results

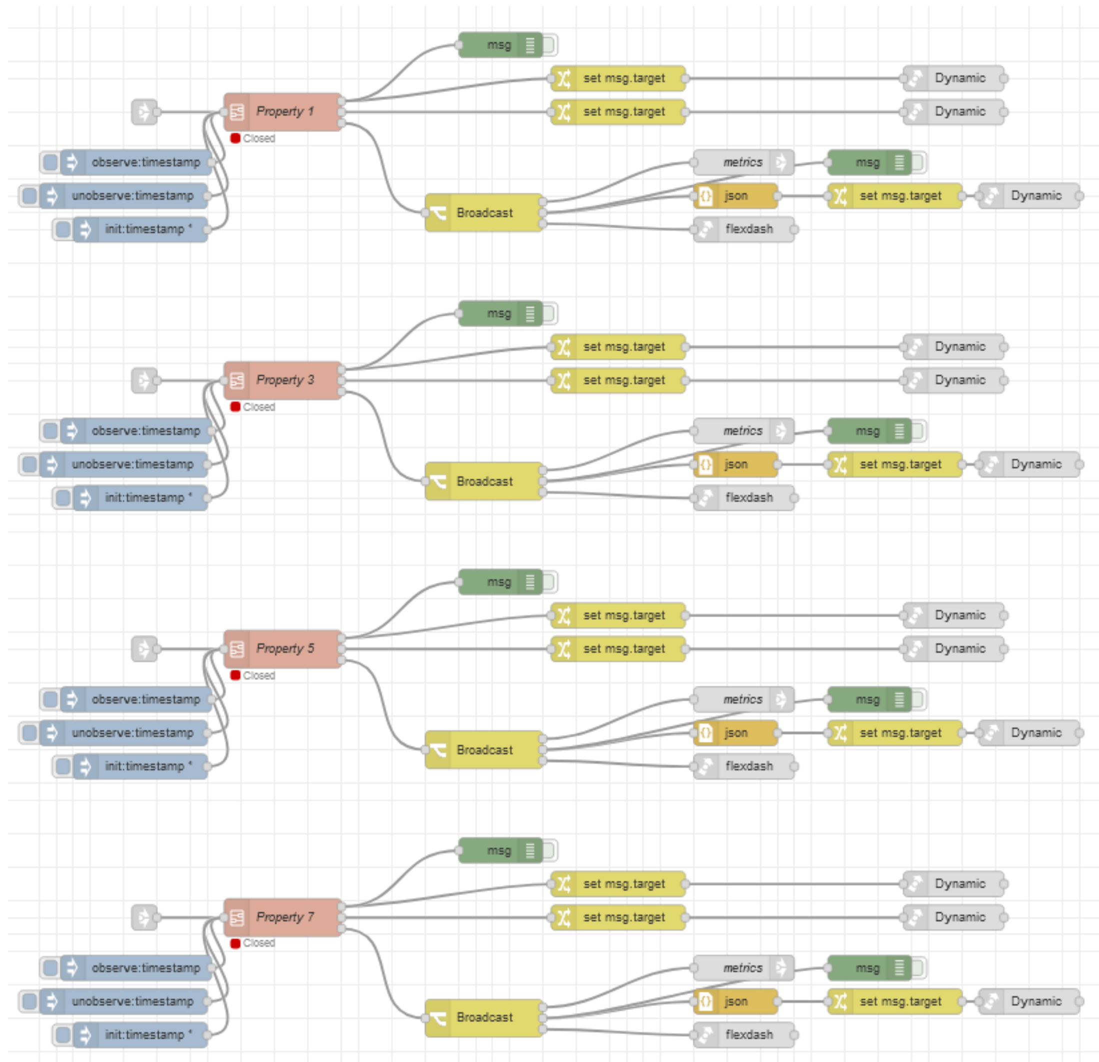

Figure A.1: Test result of 5.1, Property groups 1, 3, 5 and 7

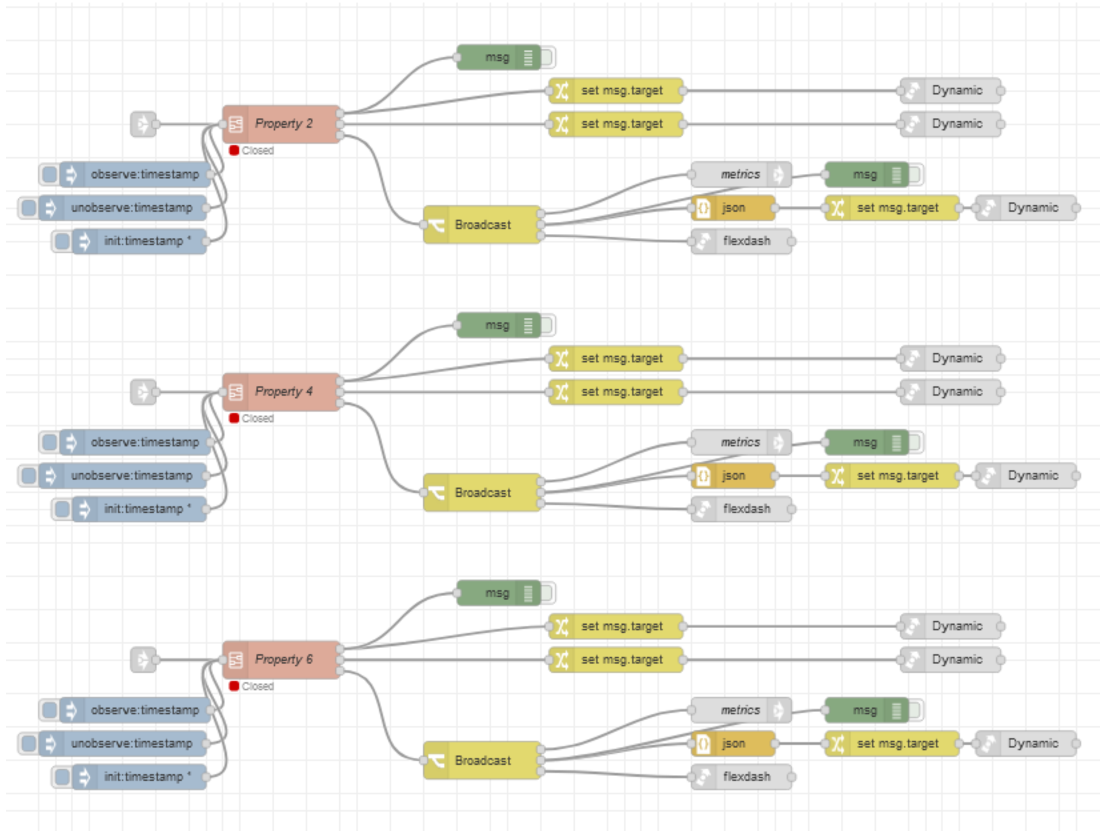

Figure A.2: Test result of 5.1, Property groups 2, 4 and 6

Edit subflow instance: Property handler (3)

Edit subflow template | Delete

Properties

Name	Property 1
PropertyName	CurrentOperatingPressure
PropertyLink	http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CurrentOperatingPressure
HistoryLength	14400
PropertyLinkEvt	http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CurrentOperatingPressureEvt

Figure A.3: Test result of 5.1, Property 1 configuration

Edit subflow instance: Property handler (3)

Edit subflow template | Delete

Properties

Name: Property 2

PropertyName: ResponseTime

PropertyLink: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTime

HistoryLength: 14400

PropertyLinkEvt: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTimeEvt

Figure A.4: Test result of 5.1, Property 2 configuration

Edit subflow instance: Property handler (3)

Edit subflow template | Delete

Properties

Name: Property 3

PropertyName: ValvePosition

PropertyLink: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ValvePosition

HistoryLength: 14400

PropertyLinkEvt: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ValvePositionEvt

Figure A.5: Test result of 5.1, Property 3 configuration

Edit subflow instance: Property handler (3)

Edit subflow template | Delete

Properties

Name: Property 4

PropertyName: ResponseTimeMovingAvg

PropertyLink: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTimeMovingAvg

HistoryLength: 14400

PropertyLinkEvt: http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTimeMovingAvgEvt

Figure A.6: Test result of 5.1, Property 4 configuration

Edit subflow instance: Property handler (3)
Edit subflow template
Delete
Properties
Name Property 5
PropertyName CylinderStatus
PropertyLink http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CylinderStatus
HistoryLength 14400
PropertyLinkEvt http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CylinderStatusEvt

Figure A.7: Test result of 5.1, Property 5 configuration

Edit subflow instance: Property handler (3)
Edit subflow template
Delete
Properties
Name Property 6
PropertyName Cycle
PropertyLink http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/Cycle
HistoryLength 14400
PropertyLinkEvt http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/CycleEvt

Figure A.8: Test result of 5.1, Property 6 configuration

Edit subflow instance: Property handler (3)
Edit subflow template
Delete
Properties
Name Property 7
PropertyName ResponseTimePred
PropertyLink http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTimePred
HistoryLength 14400
PropertyLinkEvt http://smart.festo.com/id/instance/aas/5140_0142_3091_4340/OperationalData/ResponseTimePredEvt

Figure A.9: Test result of 5.1, Property 7 configuration

A.2 Prosys test results

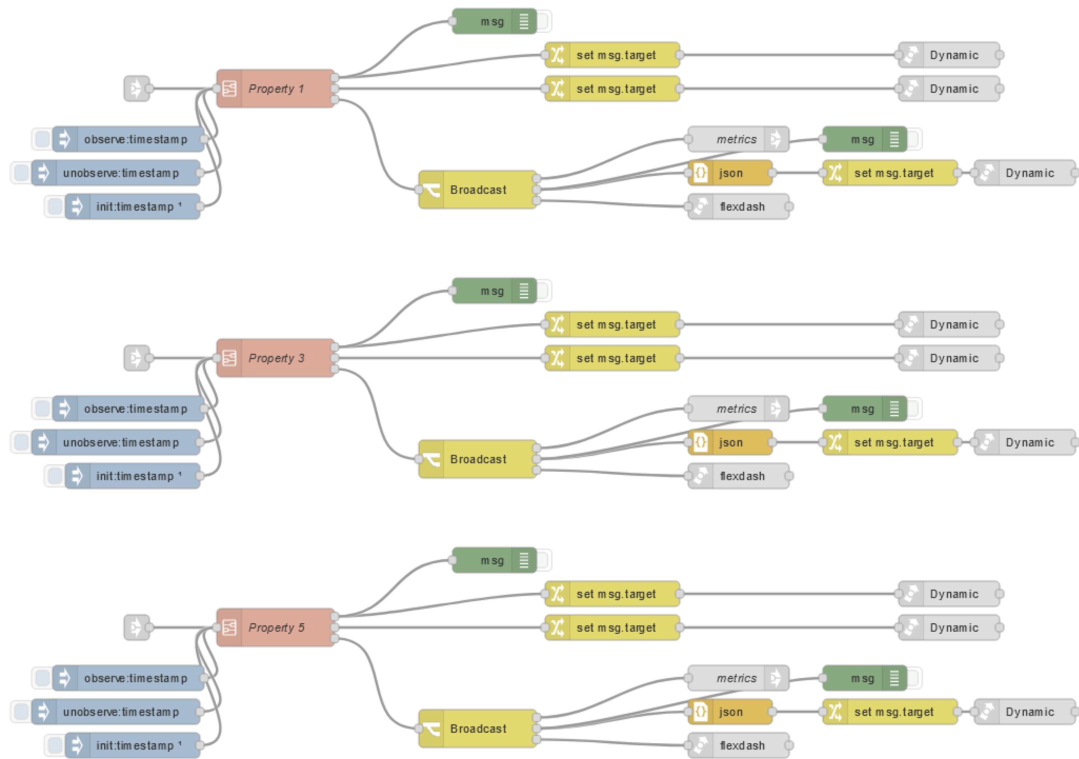

Figure A.10: Test result of 5.2, Property groups 1, 3 and 5

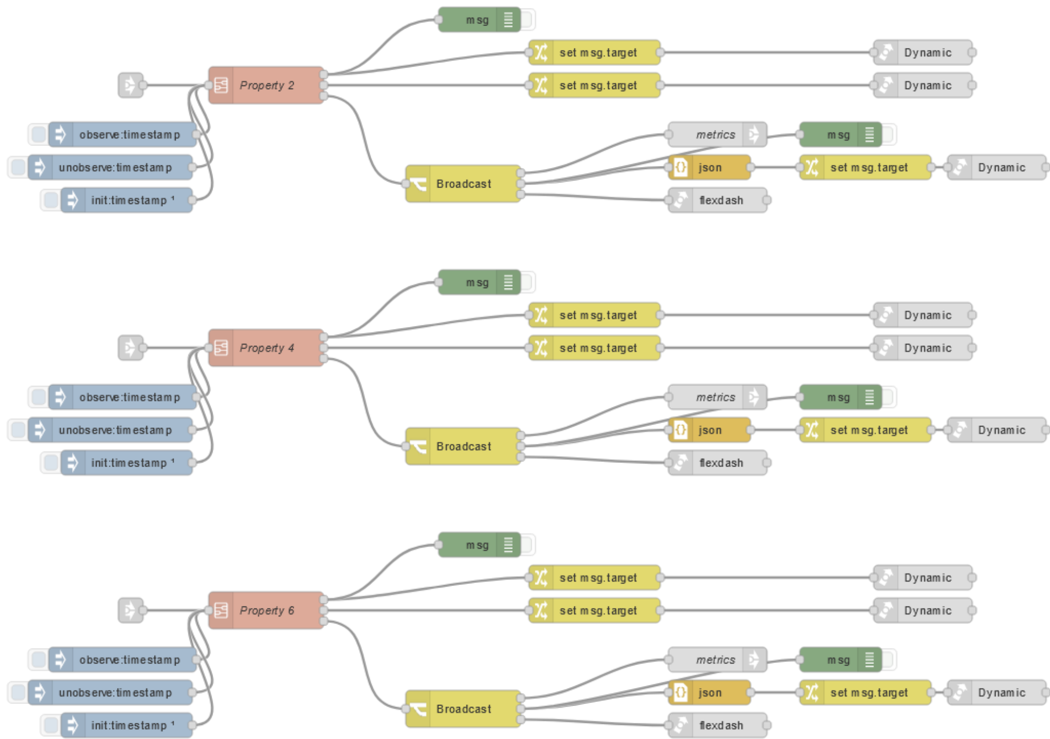

Figure A.11: Test result of 5.2, Property groups 2, 4 and 6

Properties

Name	Property 1
PropertyName	Counter
PropertyLink	https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/Counter
HistoryLength	14400
PropertyLinkEvt	https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/CounterEvt

Figure A.12: Test result of 5.2, Property 1 configuration

Properties

Name: Property 2
PropertyName: RandomNumber
PropertyLink: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/RandomNumber
HistoryLength: 14400
PropertyLinkEvt: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/RandomNumberEvt

Figure A.13: Test result of 5.2, Property 2 configuration

Properties

Name: Property 3
PropertyName: Sawtooth
PropertyLink: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/Sawtooth
HistoryLength: 14400
PropertyLinkEvt: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/SawtoothEvt

Figure A.14: Test result of 5.2, Property 3 configuration

Properties

Name: Property 4
PropertyName: Sinusoid
PropertyLink: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/Sinusoid
HistoryLength: 14400
PropertyLinkEvt: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/SinusoidEvt

Figure A.15: Test result of 5.2, Property 4 configuration

Properties

Name: Property 5

PropertyName: Square

PropertyLink: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/Square

HistoryLength: 14400

PropertyLinkEvt: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/SquareEvt

Figure A.16: Test result of 5.2, Property 5 configuration

Properties

Name: Property 6

PropertyName: Triangle

PropertyLink: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/Triangle

HistoryLength: 14400

PropertyLinkEvt: https://example.prosys.com/ids/aas/9574_3103_7012_8644/OperationalData/TriangleEvt

Figure A.17: Test result of 5.2, Property 6 configuration

A.3 Hydraulic Plant Simulator test results

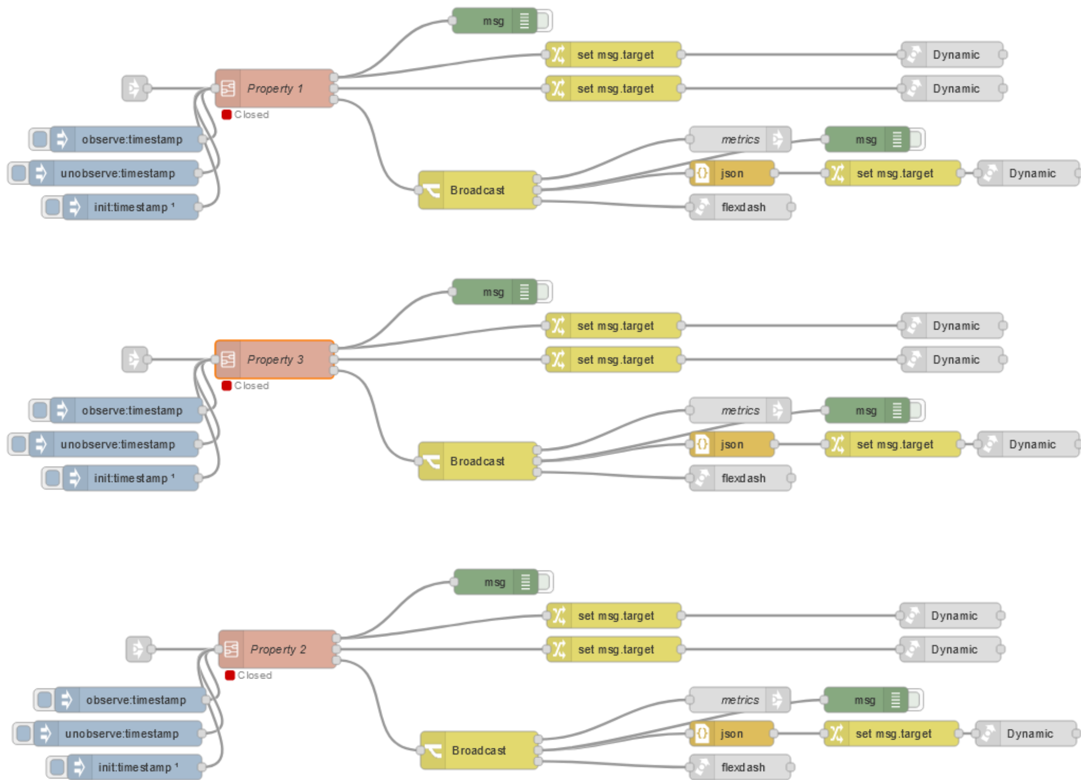

Figure A.18: Test result of 5.3, Property groups 1, 2 and 3

Properties

Name	Property 1
PropertyName	CLO.OilTank
PropertyLink	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.OilTank
HistoryLength	14400
PropertyLinkEvt	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.OilTankEvt

Figure A.19: Test result of 5.3, Property 1 configuration

Properties

Name	Property 2
PropertyName	CLO.Pumps
PropertyLink	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.Pumps
HistoryLength	14400
PropertyLinkEvt	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.PumpsEvt

Figure A.20: Test result of 5.3, Property 2 configuration

Properties

Name	Property 3
PropertyName	CLO.DeliveryLine.OilPressure
PropertyLink	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.DeliveryLine.OilPressure
HistoryLength	14400
PropertyLinkEvt	https://nova.hydraulic.simulation.com/ids/aas/6383_5151_1112_3966/OperationalData/CLO.DeliveryLine.OilPressureEvt

Figure A.21: Test result of 5.3, Property 3 configuration

2023
Development of automated NOVAAS Connectors
Henrique Sá
NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY