

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Interoperability in IIoT using IETF SDF

Jorge Daniel de Almeida Sousa



Master's degree in Informatics and Computing Engineering

Supervisor: Prof. Filipe Sousa

Second Supervisor: Carlos Resende

June 30, 2025

Interoperability in IIoT using IETF SDF

Jorge Daniel de Almeida Sousa

Master's degree in Informatics and Computing Engineering

Approved by:

President: Prof. José Magalhães Cruz

Referee: Prof. Ângelo Martins

Referee: Prof. Filipe Sousa

June 30, 2025

Resumo

O crescimento da Internet das Coisas (IoT) levou ao desenvolvimento de múltiplos protocolos de comunicação e modelos de dados. Isto criou um problema de interoperabilidade porque dispositivos que usam diferentes protocolos de comunicação e modelos de dados não conseguem perceber-se. O IETF SDF apresenta-se como uma solução para a existência de vários modelos de dados, permitindo representá-los num único formato. Deste modo, todos os dispositivos conseguem perceber a informação uns dos outros.

Construída com base no IETF SDF e no protocolo MQTT, a NextGenGW é uma gateway que permite a intercomunicação entre dispositivos que comunicam por protocolos diferentes, através da associação desenvolvida entre o IETF SDF e o MQTT.

O problema da interoperabilidade também se espalhou para o campo da Internet das Coisas Industrial (IIoT), onde é comum ter diferentes máquinas no chão de fábrica, cada uma utilizando o seu protocolo de comunicação e modelo de dados. Contudo, a usabilidade do IETF SDF não foi avaliada para estes cenários, uma vez que este foi desenvolvido para cenários comuns de IoT. Consequentemente, a NextGenGW também não foi desenhada para este propósito. Este é precisamente o problema em que esta tese se foca: avaliar se o IETF SDF e a NextGenGW são adequados a contextos industriais.

Ao longo deste trabalho analisamos os vários níveis de interoperabilidade e exploramos vários protocolos de comunicação industriais e se estes definem um modelo de dados. Destes, escolhemos o OPC UA para o estudo de compatibilidade com o IETF SDF. Se conseguirmos usar o IETF SDF para representar dados de OPC UA, podemos provar que o mesmo é adequado a cenários industriais.

Aquando da exploração por conversores existentes entre IETF SDF e o modelo de informação do OPC UA, foi encontrado um, mas apenas de IETF SDF para OPC UA. Assim, desenvolvemos o nosso conversor no sentido oposto que, apesar de não suportar todas as funcionalidades do OPC UA, demonstrou ser um bom ponto de partida, gerando documentos de IETF SDF a partir de Nós de OPC UA. Para avaliar a associação MQTT-IETF SDF proposta pela NextGenGW e as suas operações suportadas com o OPC UA, foi desenvolvida uma integração em que uma mensagem MQTT gera uma ação num servidor OPC UA. Apesar de apenas um subconjunto de operações do OPC UA ter sido avaliado, estas provaram ser exequíveis com tempos de execução e utilização de memória aceitáveis.

Abstract

The rise of the Internet of Things (IoT) led to the development of multiple communication protocols and data models. This created interoperability problems because devices using different communication protocols and data models can't understand each other. IETF SDF shows up as a solution for the existence of various data models, allowing for the representation of these data models in one. This way, every device can understand the information of the others.

Built on top of IETF SDF and the MQTT protocol, NextGenGW is a gateway that allows the intercommunication between devices that communicate via different protocols, through bindings between IETF SDF and MQTT.

The interoperability problem also spread to the field of Industrial Internet of Things (IIoT), where it is common to have different machines on the shop floor, each using its own communication protocol and data format. However, the usability of IETF SDF was not assessed for industrial scenarios, since it was designed for regular IoT use cases. Consequently, NextGenGW wasn't designed for these scenarios. This is the problem that this thesis focuses on: assessing the suitability of IETF SDF and NextGenGW in industrial contexts.

Throughout this document, we analyze the various levels of interoperability and explore several industrial communication protocols and whether they define a data model. From these, we choose OPC UA as the one to be used in the compatibility study with IETF SDF. If we can use IETF SDF to represent OPC UA data, we can prove that it is suitable for industrial scenarios.

When exploring existing converters between IETF SDF and the OPC UA information model, one was found, but it was only available in the IETF SDF to OPC UA direction. Therefore, we developed our converter in the opposite direction, which, even though it doesn't support all OPC UA features, proved to be a good stepping stone, generating IETF SDF documents from OPC UA Nodes. To evaluate the MQTT-IETF SDF bindings proposed by NextGenGW and its supported operations with OPC UA, an integration was developed in which a message sent via MQTT triggers an action in an OPC UA server. Even though only a subset of OPC UA operations was tested, it proved to be feasible with acceptable execution times and memory usage.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This thesis contributes to SDG 9, specifically to target 9.2. These contributions are described in Table 1.

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization, and foster innovation

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.2	Improving interoperability between devices contributes to a more efficient shop floor	Number of devices using different protocols connected and communicating

Table 1: UN SDGs contributions

Acknowledgements

I would like to thank my family for the support and for encouraging me to be better and always aim for more. I thank all my friends who made this journey alongside me, gifting me with so many memories, which will surely be remembered.

Thank you all,

Jorge Sousa

*“In the midst of winter, I found there was, within me, an invincible summer.
And that makes me happy.”*

Albert Camus

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem definition	2
1.3	Research questions	2
1.4	Structure	3
2	State of the art	4
2.1	IIoT protocols	4
2.1.1	MQTT	6
2.1.2	DNP3	7
2.1.3	Ethernet/IP	8
2.1.4	OPC UA	9
2.1.5	Modbus	10
2.1.6	CoAP	12
2.1.7	Summary	13
2.2	Interoperability problem	14
2.2.1	Industrial Data Ontology (IDO)	17
2.2.2	Smart Applications REference Ontology (SAREF)	17
2.3	IIoT interoperability solutions	17
2.3.1	An Interoperability Middleware for IIoT	18
2.3.2	Enabling Industry 4.0 Communication Protocol Interoperability: An OPC UA Case Study	20
2.3.3	TSN-compatible Industrial Wired/Wireless Multi-protocol Conversion Mechanism and Module	22
2.4	Conclusion	24
3	Enabling technologies	25
3.1	IETF SDF	25
3.1.1	IETF SDF groups	26
3.1.2	IETF SDF document structure	27
3.1.3	IETF SDF names and namespaces	30
3.1.4	IETF SDF tools	33
3.2	NextGenGW	33
3.2.1	System architecture	34
3.2.2	MQTT-IETF SDF bindings	35
3.3	OPC UA	37
3.3.1	OPC UA information model fundamentals	37
3.3.2	OPC UA common attributes	39

3.3.3	OPC UA standard ObjectTypes	40
3.3.4	OPC UA standard VariableTypes	41
3.3.5	OPC UA standard Methods	41
3.3.6	OPC UA standard ReferenceTypes	41
3.3.7	OPC UA standard DataTypes	41
3.4	Conclusion	42
4	Solution	43
4.1	IETF SDF theoretical mappings	43
4.2	IETF SDF - OPC UA information model conversions	44
4.2.1	IETF SDF to OPC UA information model	45
4.2.2	OPC UA information model to IETF SDF	47
4.3	NextGenGW integration	51
5	Results	57
5.1	OPC UA-IETF SDF conversion results	57
5.2	OPC UA-NextGenGW test setup	58
5.3	OPC UA-NextGenGW result analysis	59
6	Conclusion	63
6.1	Contributions	63
6.2	Future work	64
	References	65
A	IETF SDF to OPC UA information model	70
B	OPC UA information model to IETF SDF	85

List of Figures

2.1	The 5 levels of interoperability (from [54])	15
2.2	Nutrient solution system overview (from [50])	19
2.3	System architecture (from [12])	23
2.4	VLAN configuration frame (from [12])	24
2.5	Conversion module test scenario (from [12])	24
3.1	Relations between IETF SDF groups (from [38])	27
3.2	NextGenGW system architecture (from [51])	35
3.3	OPC UA <i>Object</i> Model (from [21])	37
3.4	OPC UA <i>Node</i> Model (from [21])	37
3.5	OPC UA <i>Reference</i> Model (from [21])	38
3.6	OPC UA <i>Variable</i> defined by a <i>VariableType</i> (from [21])	38
4.1	IETF SDF to OPC UA conversion process	45
4.2	IETF SDF document generation overview	48
4.3	<code>parse_all_object_nodes()</code> implementation steps	49
4.4	<code>parse_object_type()</code> implementation steps	50
4.5	NextGenGW integration following its architecture	52
4.6	Implemented integration of OPC UA with NextGenGW	53
5.1	IETF SDF modeled environment sensor	57
5.2	READ vs POST operation execution times (s)	60
5.3	READ vs POST single operation execution times (ms)	60
5.4	READ vs POST operation memory usage (%)	61
5.5	READ vs POST operation memory usage (absolute values in MB)	61
5.6	READ vs POST operation peak CPU usage (%)	62

List of Tables

1	UN SDGs contributions	iii
2.1	Number of results per source for SQ1	5
2.2	Coverage of IIoT protocols per source	5
2.3	IIoT protocols summary	14
2.4	Degrees of maturity per interoperability dimension (from [54])	16
2.5	Transformation levels and its requirements (from [54])	16
2.6	Number of results per search query per source	18
2.7	VAB primitives [6]	20
2.8	VAB primitives to HTTP verbs mappings (from [32])	21
2.9	VAB to OPC UA type mappings (from [32])	21
2.10	VAB primitives to HTTP verbs mappings (from [32])	22
3.1	IETF SDF requirements	26
3.2	IETF SDF information block qualities	28
3.3	IETF SDF existing converters	33
3.4	OPC UA <i>Node</i> common <i>Attributes</i> (from [22])	39
3.5	OPC UA <i>Object</i> common <i>Attributes</i> (from [22])	39
3.6	OPC UA <i>Variable</i> common <i>Attributes</i> (from [22])	40
3.7	OPC UA <i>VariableType</i> common <i>Attributes</i> (from [22])	40
3.8	OPC UA <i>Method</i> common <i>Attributes</i> (from [22])	40
3.9	OPC UA standard <i>Methods</i> (from [22])	41
4.1	IETF SDF to OPC UA information model theoretical mappings	43
4.2	IETF SDF to OPC UA information model evaluated mappings	44
4.3	OPC UA types supported by our converter	49
4.4	NextGenGW - OPC UA mappings	51
6.1	OPC UA to IETF SDF next features	64

Listings

3.1	IETF SDF information block example	28
3.2	IETF SDF namespaces block example	28
3.3	IETF SDF definitions block example	29
3.4	IETF SDF model example	29
3.5	IETF SDF global name example	30
3.6	IETF SDF namespace definition	31
3.7	IETF SDF namespace reference	31
3.8	IETF SDF resolved namespace reference	31
3.9	IETF SDF <i>sdfRef</i> example	31
3.10	IETF SDF <i>sdfRequired</i> example	32
3.11	IETF SDF <i>sdfRequired</i> shorter form	32
3.12	IETF SDF <i>sdfRequired</i> alternative representation	33
3.13	IoT node discovery response	35
3.14	Object-level interaction request payload	36
3.15	Property, event or action-level interaction request payload	36
4.1	IETF SDF information block construction	48
4.2	Message payload to read the values of <i>Temperature</i> and <i>Humidity</i>	53
4.3	SDFObject class	54
4.4	SDFProperty class	55
4.5	SDFAction class	55
4.6	Message payload to write the values of <i>Temperature</i> and <i>Humidity</i>	56
5.1	READ operation payload	58
5.2	POST operation payload	59
A.1	IETF SDF example document for conversion	70
A.2	YANG file generated from IETF SDF document	71
A.3	OPC UA Model Design file generated from YANG file	72
A.4	OPC UA NodeSet2 XML file from OPC UA Model Design file	74
A.5	IETF SDF changed example document for conversion	76
A.6	YANG file generated from updated IETF SDF document	77
A.7	OPC UA Model Design file generated from updated YANG file	78
A.8	OPC UA NodeSet2 XML file from updated OPC UA Model Design file	79
A.9	OPC UA types supported by our converter	81
B.1	IETF SDF modeled environment sensor	85
B.2	OPC UA information model to IETF SDF resulting file	87

Abbreviations and Acronyms

GUI	Graphical User Interface
RAMI4.0	Reference Architecture Model for Industrie 4.0
CAGR	Compound Annual Growth Rate
AI	Artificial Intelligence
IoT	Internet of Things
IIoT	Industrial Internet of Things
I4.0	Industry 4.0
IETF	International Engineering Task Force
SDF	Semantic Definition Format
MQTT	Message Queuing Telemetry Transport
JSON	JavaScript Object Notation
LwM2M	Lightweight Machine-to-Machine
IPSO	Internet Protocol for Smart Objects
OCF	Open Connectivity Foundation
DTDL	Digital Twins Definition Language
IEEE	Institute of Electrical and Electronics Engineers
ACM	Association for Computing Machinery
DNP3	Distributed Network Protocol 3
CIP	Common Industrial Protocol
OPC UA	Open Platforms Communication Unified Architecture
CoAP	Constrained Application Protocol
DDS	Data Distribution Service
HTTP	Hypertext Transfer Protocol
AASs	Asset Administration Shells

AAS	Asset Administration Shell
TSN	Time-Sensitive Networking
SCADA	Supervisory Control and Data Acquisition
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
FTP	File Transfer Protocol
SNMP	Simple Network Management Protocol
Ethernet/IP	Ethernet/Industrial Protocol
ERP	Enterprise Resource Planning
MES	Manufacturing Execution System
PLCs	Programmable Logic Controllers
PLC	Programmable Logic Controller
TLS	Transport Layer Security
API	Application Programming Interface
SenML	Sensor Measurement Lists
CBOR	Concise Binary Object Representation
XML	Extensible Markup Language
EXI	Efficient XML Interchange
TSN	Time-Sensitive Networking
SiOME	Siemens OPC UA Modeling Editor
ROS2	Robot Operating System 2
RAMI	Reference Architecture Model Industry 4.0
VAB	Virtual Automation Bus
RTT	Round Trip Time
OT	Operational Technology
OSCORE	Object Security for Constrained RESTful Environments
ACE	Authentication and Authorization for Constrained Environments
IDO	Industrial Data Ontology
SAREF	Smart Applications REference Ontology
SAREF4INMA	SAREF for Industry and Manufacturing

YANG Yet Another Next Generation

WSL Windows Subsystem for Linux

QoS Quality of Service

Chapter 1

Introduction

This chapter introduces this thesis by framing its theme in Section 1.1 and describing the problem to be solved in Section 1.2, together with the questions that guide this research in Section 1.3. Section 1.4 gives an overview of the remainder of this report's structure.

1.1 Context

The Internet of Things (IoT) has been growing at a fast pace. Billions of smart devices are expected to be involved in the IoT paradigm [2], and device-to-device communications surpassed person-to-person communications in 2020, when there were 11.7 billion active IoT device connections in a total of 21.7 billion active connections worldwide. Furthermore, a growth of 13% in IoT device communications is expected from 2020 to 2025 [36], while a Compound Annual Growth Rate (CAGR) of 14% in the IoT market is expected from 2025 to 2030, mainly driven by Artificial Intelligence (AI) integration, cloud integration, and sustainability sectors like energy, utilities, and smart infrastructure. [31]

Despite all its benefits, the ubiquity of IoT brought some challenges. One is that several communication protocols and data formats were developed, resulting in a lack of interoperability between them. Some scenarios require energy-efficient protocols, while others require higher bandwidths or lower latencies. Solving this interoperability problem may result in an economic growth of 30% to 89% in factories, work sites, and retail environments. [51] [37]

To tackle the IoT interoperability problem, Fraunhofer AICOS developed NextGenGW, a software middleware gateway that homogenizes the different protocols and data formats using standards to avoid vendor lock-in. They are the Message Queuing Telemetry Transport (MQTT) communication protocol and the International Engineering Task Force (IETF) Semantic Definition Format (SDF), a data format designed to homogenize different data formats and to be used in the creation and maintenance of data and interaction models that describe Things, which are physical objects that are available for interaction over a network.

However, the requirements in IoT and Industrial Internet of Things (IIoT) are not the same, and both NextGenGW and IETF SDF target the IoT field. IIoT is a core pillar of the currently

ongoing fourth industrial revolution, and the fact that its compatibility with **IETF SDF** is not ensured motivated this thesis.

1.2 Problem definition

The development of multiple communication protocols and data formats poses a challenge in the **IoT** and **IIoT** fields. **IETF SDF** presents itself as a possible solution for this problem. However, it was designed for **IoT** scenarios, with current support for Internet Protocol for Smart Objects (**IPSO**), Open Connectivity Foundation (**OCF**), Web of Things [52], Yet Another Next Generation (**YANG**), Digital Twins Definition Language (**DTDL**), and Matter, which are not broadly used in industrial applications [40].

The homogenization of data formats brought by **IETF SDF** could be incredibly useful in **IIoT**, where different data models are and will continue to be used. It is essential to assess if various protocols could be homogenized via **IETF SDF**, making a study on this subject important.

NextGenGW is, to the best of our knowledge, the only solution that binds **IETF SDF** to a communication protocol - **MQTT** - providing an interoperability solution at both the data modeling level, that comes with **IETF SDF**, and at the communication level, coming from the bindings between **MQTT** and **IETF SDF**.

NextGenGW is also aiming for generic **IoT** applications, i.e., there are no evident efforts to apply it to industrial scenarios, which might bring additional requirements.

To be used in industrial scenarios, NextGenGW translation mechanisms must comply with **IIoT** communication protocol requirements and objectives. To assess this, it is crucial to identify and analyze the communication protocols and data models commonly used in **IIoT**. In such analysis, identifying the **IIoT** use case commonly associated with each protocol and data model, their adoption rate, and the existence of open-source libraries is important in determining which ones are likely compatible with **IETF SDF** and NextGenGW.

1.3 Research questions

Considering the problems defined in Section 1.2, the focus of this work is to investigate different **IIoT** protocols and data models and check their compatibility with **IETF SDF**.

Therefore, this work addresses the following research questions:

- **RQ1:** What is the current landscape of **IIoT** application protocols and data models, namely what is their adoption rate, use cases, and community support?
- **RQ2:** Is **IETF SDF** compliant with **IIoT**? If not, what are its limitations, and can it be adapted to such an application domain?
- **RQ3:** Is **IETF SDF** binding with **MQTT** proposed by NextGenGW a viable solution for the **IIoT** interoperability issue?

1.4 Structure

Other than this chapter, this document is composed of five more chapters. In Chapter 2, we start by going through common industrial communications protocols, describing their advantages, disadvantages, open-source implementations, and whether they define any data model. From these protocols, one is selected for the compatibility study with IETF SDF. Then, we present the different levels of interoperability, followed by related works that present solutions to the interoperability problem described in Section 1.2.

In Chapter 3, a thorough explanation of IETF SDF and of the selected protocol is performed, along with an overview of NextGenGW. This knowledge will be crucial to understand the work developed in Chapter 4, where a mapping between the chosen protocol and IETF SDF is proposed and implemented through a converter, as well as OPC UA integration proposals with NextGenGW. In Chapter 5, we evaluate the converter results and the performance of the implemented integration of OPC UA with NextGenGW.

Finally, Chapter 6 concludes this work, indicating this thesis's contributions and the future steps to improve the developed solution.

Chapter 2

State of the art

Before assessing the suitability of **IETF SDF** in industrial scenarios, we must first conduct an investigation on common **IIoT** communication protocols and data models. In Section 2.1, we present the research process to find these protocols, followed by a description of each one of the relevant findings, pointing out their advantages, disadvantages, open-source implementations, and whether they define data models.

To get a deeper understanding of the interoperability problem in **IIoT**, in Section 2.2, we go through the interoperability levels found in the literature, ranging from the technical to the dynamic levels. From the levels mentioned, we highlight the semantic one, where ontologies are used, so we reviewed some relevant industrial ones. This is followed by another investigation, in Section 2.3, this time on solutions to solve the interoperability problem where multiple devices employ different communication protocols.

The sources used in all the investigations were Institute of Electrical and Electronics Engineers (**IEEE**), Association for Computing Machinery (**ACM**), ScienceDirect, and Scopus, and an exploratory literature review was followed.

2.1 IIoT protocols

This section documents the process of finding common industrial communication protocols and data models, answering the research question **RQ1**. By the end of this chapter, one of these protocols is chosen to be part of the **IETF SDF** compatibility study. We have not limited our search to the communication protocols that define a data model for the information they transport. This means our search query returned communication protocols irrelevant to the **IETF SDF** compatibility study. We have used such a broad query with the intention of capturing the most used protocols in the literature, independently of their association with a standard data model, because this is still an interesting output. The search query used was:

- SQ1: ("Document Title": protocol) AND ("All Metadata": IIoT)

Data Source	Number of Results
IEEE	109
ACM	17
ScienceDirect	74
Scopus	687
Total	887

Table 2.1: Number of results per source for SQ1

Table 2.1 shows the number of results retrieved. After removing duplicates and evaluating eligibility, only 10 papers seemed relevant. The inclusion criteria justify the discrepancy between the number of results found and those considered relevant. It consists of:

- **IC1:** The articles reference industrial communication protocols.
- **IC2:** The articles must be published between 2019 and 2024.
- **IC3:** The articles must be written in English.
- **IC4:** The referenced communication protocols must be used in IIoT scenarios.

Throughout the selected papers, many protocols were referenced, reinforcing the existence of the interoperability problem. From these protocols, some filtering had to be applied. We discarded some that were domain-specific, where the domain didn't apply to this thesis use case, as well as IoT protocols, due to our strict interest in IIoT protocols. There were also references to camera interface and field protocols, which were discarded for the same reason. Table 2.2 summarizes the filtered protocols referenced per consulted source.

Protocol	[60]	[59]	[10]	[42]	[4]	[29]	[1]	[14]	[26]	[30]
MQTT	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓
DNP3	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Ethernet/IP	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
OPC UA	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗
Modbus	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓
CoAP	✗	✗	✓	✓	✓	✗	✓	✗	✗	✓

Table 2.2: Coverage of IIoT protocols per source

Next, we go through each of the selected communication protocols and briefly describe them, their advantages, disadvantages, and existing open-source implementations. By the end of this section, Table 2.3 summarizes this information and adds a column to indicate if a protocol defines a data model.

2.1.1 MQTT

MQTT [48] is a lightweight publish/subscribe messaging protocol. It is one of the most popular protocols, and it can be used in regular **IoT** applications, but also in the industry, such as automotive, manufacturing, and telecommunications.

2.1.1.1 Advantages

MQTT presents several benefits. Some of the most relevant are:

- **Reliability:** **MQTT** defines three Quality of Service (**QoS**) options, allowing for a trade-off between reliability and overhead. It is possible to choose a lower **QoS** to avoid higher overheads, compromising message receipt acknowledgments. On the other hand, we can opt for reliability by choosing a higher **QoS**.
- **Scalability:** **MQTT** can scale to connect millions of devices.
- **Security:** **MQTT** uses Transport Layer Security (**TLS**) for message encryption and provides authentication mechanisms through modern authentication protocols, like OAuth.
- **Lightweight:** To run an **MQTT** client, the resources needed are minimal since the clients are very small. Furthermore, the message headers are also small to optimize network bandwidth.

2.1.1.2 Disadvantages

In [60], the authors list some disadvantages of using **MQTT**. Below, we present the ones that we consider the most important:

- **Timeliness:** **MQTT** may not be suitable for scenarios with strict real-time requirements.
- **Centralized broker:** **MQTT** relies on a centralized broker to exchange messages, creating a single point of failure.
- **Missing discovery mechanism:** **MQTT** does not come with a built-in discovery mechanism. Users need to use external libraries to get this functionality.

2.1.1.3 Open-source implementations

Eclipse Mosquitto [18] is an open-source implementation of a message broker and client for the **MQTT** protocol. It supports versions 5.0, 3.1, and 3.1.1, and it is written in C++, making it suitable to run on embedded devices and microcontrollers.

2.1.2 DNP3

Distributed Network Protocol 3 (**DNP3**) [45] is an application layer open protocol used in the communication between Supervisory Control and Data Acquisition (**SCADA**) systems and remote devices in industries like water and electricity [59]. This protocol defines the structure and format of **DNP3** messages through an object-based model. This model defines types, properties, and behaviors to represent typical **SCADA** components. **DNP3** can use either Transmission Control Protocol (**TCP**) or User Datagram Protocol (**UDP**) in the transport layer.

2.1.2.1 Advantages

DNP3 has the following features:

- **Reliability and robustness:** The protocol has error detection and error correction mechanisms as well as data integrity checks, ensuring data quality in harsh environments.
- **Scalability:** The support of various data types and the possibility to handle thousands of connections make **DNP3** scalable and suitable for scenarios with lots of devices.
- **Security:** **DNP3** has built-in support for encryption, authentication, and data integrity checks.
- **Time Synchronization:** Essential for applications requiring time coordination and sequencing, **DNP3** allows time synchronization among devices through action synchronization and precise timestamps.
- **Historical Data Retrieval:** Checking past data makes troubleshooting and diagnostics easier while allowing past events and trend analysis.

2.1.2.2 Disadvantages

DNP3 provides great features, but these don't come without drawbacks. We highlight the following:

- **Complexity:** **DNP3** is complex to implement, which can result in more complicated maintenance and longer development times.
- **Limited Bandwidth:** An overhead in the network may be introduced when using **DNP3**, especially if there are multiple devices connected.
- **Real-time Constraints:** **DNP3** is not the most suitable protocol for hard real-time constrained applications.

2.1.2.3 Open-source implementations

OpenDNP3 [28] is a library implementing the **DNP3** protocol stack written in C++ 11. However, this project reached its end of life in September 2022 and is currently archived on GitHub. Another open-source implementation of this protocol besides this repository could not be found.

2.1.3 Ethernet/IP

Ethernet/Industrial Protocol (**Ethernet/IP**) [49] [59] is an application layer, open communication protocol built over Ethernet used in industrial automation scenarios. It can use **TCP** or **UDP** in the transport layer and allows devices to communicate with each other in an industrial network. **Ethernet/IP** brings Common Industrial Protocol (**CIP**) to Ethernet, thus making use of its object-oriented data representation through which devices represent their functionalities and data. A unique identifier is assigned to these objects, which other devices can access and manipulate. **Ethernet/IP** supports device discovery, configuration, and diagnostics, and uses explicit messaging to communicate. It also supports real-time monitoring and industrial process control through I/O messaging. Finally, a standardized definition of device profiles describing their capabilities can be specified, contributing to easier integration of new devices and increased interoperability.

2.1.3.1 Advantages

The fact that **Ethernet/IP** is built on top of standard Ethernet brings some advantages. Let's take a look at the main ones:

- **Interoperability:** **Ethernet/IP** is compatible with standard Ethernet, allowing integration and communication with devices that use this technology.
- **High bandwidth:** **Ethernet/IP** provides high-speed communication, providing fast data transfer between devices.
- **Scalability:** **Ethernet/IP** supports many devices and integrating new ones is easy.
- **Support for common protocols:** The support of industry common protocols such as Simple Network Management Protocol (**SNMP**), Hypertext Transfer Protocol (**HTTP**), and File Transfer Protocol (**FTP**) allows for easier integration with other enterprise systems.

2.1.3.2 Disadvantages

Some Ethernet disadvantages are also in **Ethernet/IP**. Here are the most concerning ones:

- **Complexity:** **Ethernet/IP** may be complex to configure to ensure reliable communication.
- **Security:** **Ethernet/IP** doesn't have security features by itself; therefore, additional security measures must be taken to ensure secure connections.
- **Timeliness:** Even though **Ethernet/IP** provides high bandwidth, it still lacks deterministic behavior. Additional measures, such as prioritization mechanisms, must be implemented to solve this issue.
- **Cost:** To implement **Ethernet/IP**, switches, networking hardware, and cables are needed, making it costly to implement.

2.1.3.3 Open-source implementations

opENer [27] is a portable open-source implementation of **Ethernet/IP**. It supports multiple I/O and explicit connections and is written in C. The default version targets a POSIX operating system.

EIPScanner [13] is another implementation of **Ethernet/IP** written in C++ 20. The authors highlight the support for explicit and implicit messaging (the latter is only available in point-to-point) and discovery services. Additionally, it also supports **CIP** standard objects.

2.1.4 OPC UA

Open Platforms Communication Unified Architecture (**OPC UA**) [20] is a platform-independent industrial standard created in 2008 that provides an architecture for interoperability. It combines all the OPC Classic functionalities in one framework [10].

OPC UA defines an information model to represent structure, behavior, and semantics; a message model for inter-device communication; a communication model for end-to-end data transmission; and a conformance model to ensure interoperability between systems.

OPC UA supports client-server and publisher-subscriber architectures and supports multiple security features, such as session encryption, message signing, sequenced packets, authentication, user control, and auditing [14] [42].

With **OPC UA**, communication across automation pyramid levels is possible, from sensors and actuators to Enterprise Resource Planning (**ERP**) and Manufacturing Execution System (**MES**) systems [14].

It was the only recommendation for the communication layer in the Reference Architecture Model for Industrie 4.0 (**RAMI4.0**) [19].

2.1.4.1 Advantages

OPC UA comes with multiple features. Here's some [60]:

- **Interoperability:** **OPC UA** allows for seamless communication between different devices and systems. It focuses on the transport and the semantic layers [14].
- **Vendor-neutrality:** **OPC UA** is not associated with any hardware or software manufacturer. This enables users to choose from various compliant devices, allowing for custom setups.
- **Wide adoption:** **OPC UA** has been growing a lot in the industry, and, therefore, many compliant devices are available. The wide adoption also ensures that the protocol continues to be supported by the OPC Foundation.
- **Scalability:** **OPC UA** can be used in small or large industrial environments, from single machines to distributed systems, making adding new devices to the setup easier.

2.1.4.2 Disadvantages

Some **OPC UA** strengths bring some disadvantages as the ones listed below [60]:

- **Complexity:** **OPC UA** can be challenging to implement due to the various configuration options available. Users with little experience may find it complex because they lack an understanding of the underlying concepts.
- **Version compatibility:** The evolution of **OPC UA** resulted in multiple versions, and this can introduce complexity when integrating devices that support a specific version only or with legacy systems.
- **Security:** It was stated that **OPC UA** is focused on data exchange and communication; thus, security features may be compromised. However, security is a highlighted feature in the **OPC UA**'s official website, where it is said that it supports session encryption, message signing, and authentication, among others.
- **Performance:** **OPC UA** may introduce overhead to the system, especially in environments with high data throughput and low-bandwidth networks. The system design must be made carefully to minimize this overhead.

2.1.4.3 Open-source implementations

UA-.NETStandard [25] is an implementation of the **OPC UA** protocol written in the C# programming language. Using the .NET Standard stack, the code can be run on all common platforms, such as Linux, iOS, Android, and Windows, including its **IoT/Embedded** editions. They also offer a bare-bones implementation of the publisher-subscriber version of the protocol [23].

Free OPC UA [35] has some protocol implementations written in Python and C++ 11. For the C++ version, which would be the most interesting due to the possibility of deployment in embedded devices, the authors state that documentation is lacking, even though there are several examples. However, the lack of documentation is a downside.

open62541 [44] is an OPC UA open-source implementation written in C. It can be used to build OPC UA clients and servers and is platform-independent.

2.1.5 Modbus

Modbus [47] is a long-established open-source application-layer protocol originally published in 1979 by Modicon. It is used for communication between field devices and Programmable Logic Controllers (PLCs) [59] [26].

Modbus follows a master-slave architecture, uses a polling-based mechanism, and communicates via serial lines, specifically RS232, RS422, and RS485 [59]. It is a light protocol with low memory consumption. In standard Modbus, one can connect up to 247 devices [4].

Later, a version of Modbus named Modbus TCP, capable of running over Ethernet, was created. The difference here is that this version follows a client-server architecture and a request-response mechanism. Similar to standard Modbus, a client can query a server for data or ask to execute a command using a frame formatted in a specific way. The server then responds with the requested data or confirmation that the command was run [30].

2.1.5.1 Advantages

Modbus has many advantages:

- **Lightness:** Modbus is a lightweight protocol with low memory consumption, which can be ideal for embedded devices with low memory specifications.
- **Wide adoption:** Modbus is a highly adopted protocol in the industry, resulting in numerous compatible devices and a lot of documentation.
- **Interoperability:** Modbus TCP operates over standard Ethernet and is therefore compatible with most Ethernet hardware.
- **Costs:** Due to being widely adopted and the low memory usage, Modbus can be run on cheap hardware.

2.1.5.2 Disadvantages

Having been developed a couple of years ago, Modbus has some disadvantages derived from the time of its development and its original purpose. Below, we list some of them:

- **Security:** Standard Modbus was not built with security in mind, so it is lacking in this field. However, Modbus TCP supports **TLS**, which improves over the standard version [26].
- **Performance:** Modbus is not suitable for scenarios with high transmission rates. In [59], it was noted that Modbus was not picked for that use case because they required at least 100 kilo samples per second.
- **Scalability:** Standard Modbus is limited to 247 devices per installation, making it less appealing for environments with many devices. In this case, one would have to join devices in groups of at most 247 devices, and another solution would need to be used to facilitate communication between the groups.

2.1.5.3 Open-source implementations

Since Modbus is a popular, open-source, widely adopted protocol, finding implementations was not problematic.

eModbus [15] provides server, client, and bridge functionalities for Modbus RTU, ASCII, and TCP. It was designed for ESP32, has a non-blocking Application Programming Interface (**API**),

and all common Modbus function codes. However, the authors state that enthusiasts built the project, so it may not be suitable for industrial deployments. It is written in the C++ programming language.

libmodbus [34] is a Modbus library written in the C programming language that can be run on Windows, macOS, Linux, and FreeBSD. It can be run in microcontrollers using Embox RTOS. It supports communication via serial port or Ethernet. This library is under an LGPL license, and if one does not want the limitations of this license, a paid license must be acquired.

2.1.6 CoAP

Constrained Application Protocol (**CoAP**), defined in RFC 7252 [58], is an application layer protocol designed for constrained networks and devices. It uses the REST mechanism [10] [30]. The base **CoAP** supports only request-response communication, though unicast or multicast publisher-subscriber can be enabled via extensions [42] [1].

For the transport layer, **CoAP** uses **UDP**, and for the network layer, it can use either IPv6 or 6LoWPAN [4].

CoAP defines multiple data types and can use Sensor Measurement Lists (**SenML**), JavaScript Object Notation (**JSON**), Concise Binary Object Representation (**CBOR**), Extensible Markup Language (**XML**), Efficient XML Interchange (**EXI**), and popular data models for sensors [42].

2.1.6.1 Advantages

CoAP offers excellent features for constrained environments, ranging from low latency to security. We point out some of them below:

- **Performance:** Since **CoAP** is connectionless, it can achieve lower latencies and improved performance [4].
- **Security:** **CoAP** can use DTLS to improve security, as well as Object Security for Constrained RESTful Environments (**OSCORE**) and Authentication and Authorization for Constrained Environments (**ACE**) for the application layer [42].
- **Timeliness:** Due to the use of **UDP**, **CoAP** has the potential to be real-time capable when complemented with a time-slot mechanism [42].

2.1.6.2 Disadvantages

CoAP, published in 2014, is a relatively recent protocol. Therefore, it may not be as complete or robust as some older protocols. Below, we list **CoAP**'s main disadvantages:

- **Immature sensor data models:** Compared to the data modeling capabilities offered by **OPC UA**, **CoAP** sits behind it [42].
- **Reliability:** Using **UDP** in the transport layer can introduce packet loss, compromising the protocol's reliability [42].

2.1.6.3 Open-source implementations

libcoap [7] is a C implementation of **CoAP**, allowing the implementation of servers, clients, and **CoAP** Resource Discovery servers. Moreover, DTLS can be used through frameworks like GnuTLS, OpenSSL, etc. It is designed to run on embedded devices and high-end computer systems with a POSIX OS, making moving the developed code to the target platform easy.

2.1.7 Summary

In Table 2.3, we present a summary of the advantages, disadvantages, open-source implementations, and whether the studied protocols define a data model.

Results show that these protocols share many advantages, from reliability to security, as well as disadvantages, mainly related to complexity and performance constraints.

From these protocols, only the three with data models defined are eligible for the **IETF SDF** compatibility study. They are **DNP3**, Ethernet/IP, and OPC UA.

Given all the advantages presented and the maturity of its information model, OPC UA was the selected protocol to proceed in the study. A decisive factor was that this protocol was the only one recommended for the communication layer in the **RAMI4.0** [19]. In Chapter 3, we will go into further details about this protocol.

Protocol	Advantages	Disadvantages	Implementations	Data Model
MQTT	Reliability Scalability Security Lightweight	Timeliness Centralized broker Missing discovery mechanism	Eclipse Mosquitto	N
DNP3	Reliability Scalability Security Time synchronization Historical data retrieval	Complexity Limited bandwidth Real-time constraints	OpenDNP3 (archived)	Y
Ethernet/IP	Interoperability High bandwidth Scalability Common protocol support	Complexity Security Timeliness Cost	opENer EIPScanner	Y
OPC UA	Interoperability Vendor-neutrality Wide adoption Scalability	Complexity Version compatibility Performance	UA-.NetStandard Free OPC UA open62541	Y
Modbus	Lightness Wide adoption Interoperability Cost	Security Performance Scalability	eModbus libmodbus	N
CoAP	Performance Security Timeliness	Immature sensor data models Reliability Use in the industry	libcoap	N

Table 2.3: IIoT protocols summary

2.2 Interoperability problem

In this section, we address the five levels of interoperability developed for a framework proposed in [54] to assess the current status of a company in its journey toward interoperability. The five levels are based on the "Levels of Conceptual Interoperability Model" work [57] and they are presented in Figure 2.1.

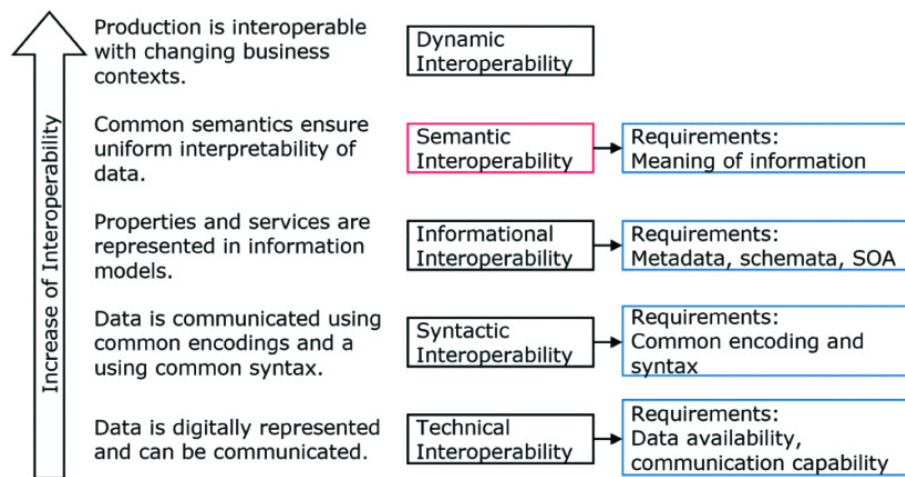


Figure 2.1: The 5 levels of interoperability (from [54])

At the Technical Interoperability level, the lowest one, no semantics or data formats are defined. It is only expected to have assets ready to be integrated into the information and communication infrastructure.

At the Syntactic Interoperability level, data formats are defined, such as **JSON** or **XML**. Still, the meaning of data is not contemplated, i.e., the meaning of **XML** tags. At this level, the content may be vendor-specific; therefore, the receiving device must know what it is expected to receive to understand it.

At the Informational Interoperability level, schemas are used to formally describe the structure of data. The data providers are also the schema or metadata providers, and these schemas can be interpreted at runtime.

At the Semantic Interoperability level, the meaning of data is specified. Typical approaches involve ontologies, which formally represent knowledge of a specific domain as a set of concepts, categories, and relationships between them.

At the Dynamic Interoperability level, a system can automatically adapt to context changes without manual intervention. However, this is an ambitious goal that still needs further research.

Based on these levels, the proposed framework describes four dimensions of interoperability: technical interoperability, syntactic interoperability, informational interoperability, and semantic interoperability. Across each category, various degrees were defined to provide a more detailed evaluation. A summary is presented in Table 2.4.

Index	A: Technical interoperability	B: Syntactic interoperability	C: Informational interoperability	D: Semantic interoperability
0	no digital or digitized data	no connectivity	no information models	no semantics
1	unstructured data	connectivity using established protocols	information models for specific assets or applications	common semantics for specific assets or applications
2	structured data	common syntax	standardized factory-wide information model	standardized factory-wide semantics
3	structured data compliant with functional requirements			

Table 2.4: Degrees of maturity per interoperability dimension (from [54])

Finally, the authors propose four transformation levels and their requirements, as shown in Table 2.5. In the first transformation level, information can be exchanged without a specific syntax, and production lines are static and inflexible. Transformation level 2 adds full syntactic interoperability to the features of transformation level 1. At this level, machines can communicate with one another, with applications being able to access others' assets. Transformation level 3 introduces factory-wide information models, and assets can be easily integrated or replaced. Finally, the goal of transformation level 4 is the discovery and access of assets and their capabilities. Note that this can be done either with a factory-wide information model (C_2) or not (C_0).

Transformation Level	1 technical	2 syntactic	3 informational	4 semantic
Dimension A	A3	A3	A3	A3
Dimension B	B1	B2	B2	B2
Dimension C	C0	C0	C2	C2 / C0
Dimension D	D0	D0	D0	D2

Table 2.5: Transformation levels and its requirements (from [54])

We will now focus on the semantic level, which we considered to be the most interesting for this thesis. Semantic interoperability is a fundamental feature of Industry 4.0 (I4.0). It enables connected devices to understand the exchanged data, building a path to achieve the desired interoperability.

Achieving semantic interoperability could lead to new data-driven insights and optimizations, and improve manufacturing operations due to data being promptly and accurately exchanged. An overview of semantic interoperability in I4.0 shows that common approaches to achieving semantic interoperability involve ontologies [3].

Ontologies can be used to describe a system's components in a way that devices understand what others are communicating. Then, rules can be applied to ontologies, enabling semantic reasoning and the inference of new information from data. Examples of industrial ontologies are Smart Applications REference Ontology (SAREF) and Industrial Data Ontology (IDO). We will provide a brief overview of each one of these below.

2.2.1 Industrial Data Ontology (IDO)

IDO is the initial part of the ISO standard 23726 Ontology-Based Interoperability.

IDO is a Web Ontology Language (OWL) ontology suited for industrial data and information. It can be used to build vocabularies and asset models, benefiting from OWL DL to infer implicit knowledge and quality assurance.

This work is still under development, but exploring it once it is more mature can be interesting. A draft can be found in [5].

2.2.2 Smart Applications REference Ontology (SAREF)

SAREF is a collection of ontologies to enable semantic interoperability across multiple domains through a "shared model of consensus." It is published as a set of open standards by the ETSI Technical Committee Smart Machine-to-Machine Communications [17].

SAREF comprises a core ontology, reference ontology patterns to help and guide in the utilization and extension of SAREF, and several extensions to serve different domains. SAREF Core specifies the smart applications domain's base concepts and their relations.

SAREF for Industry and Manufacturing (SAREF4INMA) is the extension that fits this project the most since it was developed for industrial and manufacturing scenarios, more specifically to solve interoperability issues leading to product traceability to the corresponding production equipment, batch, material, and manufactured time. It was published in 2019 and extends the base version of SAREF with 24 additional classes, 20 object properties, and 11 data type properties [16].

2.3 IIoT interoperability solutions

Now that we are familiar with some industrial protocols and with the levels of interoperability, we will look for solutions for the interoperability problem, caused by the development of various protocols and data formats. The goal is to find out if there is a common approach in the solutions and how they compare to NextGenGW.

The search queries used should be able to capture articles describing these solutions. However, we are still expecting to find other results, possibly related to different levels of interoperability that are not so relevant for this section. The search queries employed were:

- **SQ1:** ("Document Title":interoperability) AND ("Document Title":industry 4.0)
- **SQ2:** ("Document Title":iiot) AND ("All Metadata":interoperability)

The number of results obtained across the data sources and for each query is presented in Table 2.6

Data Source	SQ1	SQ2	Total
IEEE	12	2	14
ACM	1	2	3
ScienceDirect	6	2	8
Scopus	43	8	51
Total	62	14	76

Table 2.6: Number of results per search query per source

The inclusion criteria used were the following:

- **IC1:** The articles are written in English.
- **IC2:** The articles were published between the years 2020 and 2024.
- **IC3:** The solutions described in the articles must involve the intercommunication between at least 2 different protocols.
- **IC4:** The protocols used in the solutions must be in the scope of **IIoT**.

After removing duplicates and applying this criterion, we ended up with three articles, which all revolve around middleware gateways. Next, we will describe what was the approach that the authors took to solve the communication issue between different protocols.

2.3.1 An Interoperability Middleware for IIoT

Here, the authors propose a middleware gateway for a system that uses three communication protocols: Data Distribution Service (**DDS**), **MQTT**, and **OPC UA** [50]. The system has eight level sensors, one pH sensor, and an electrical conductivity sensor. Regarding actuators, the system uses a valve, five pumps, and an air compressor. It is shown in Fig. 2.2.

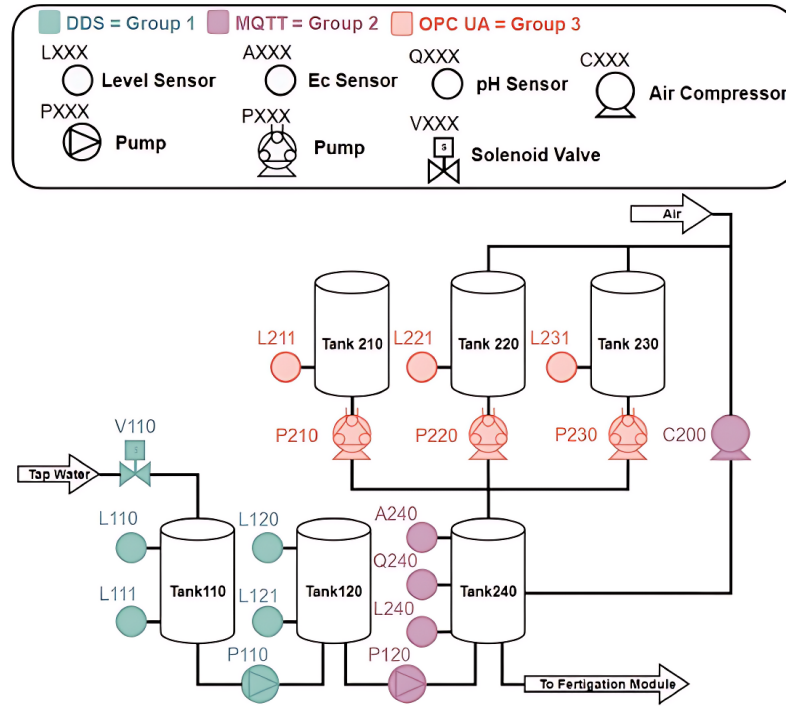


Figure 2.2: Nutrient solution system overview (from [50])

This system uses IIoTOnTo, a proposed ontology that combines multiple models with **IEEE** standardized ontologies, introducing industrial concepts that were missing from these. This is the system's entry point, where an automation engineer, typically, can instantiate the components of the industrial plant like sensors, actuators, **PLCs**, etc. Then, two files are generated: one JSON file and one YAML file, which will be used later to configure the protocol translators.

Asset Administration Shells (**AASs**) are also employed, which allow for the digital representation of assets and support **OPC UA** data communication. Siemens OPC UA Modeling Editor (**SiOME**) is used to create the digital representation of the assets, resulting in an XML file to be used by open62541, an open-source library implementation of **OPC UA** written in C. This library takes the XML file and generates the **OPC UA** server code needed for the Asset Administration Shell (**AAS**).

After the modeling phase, we can move to the system execution part. Since the system was designed to support **MQTT** and **DDS**, the authors used Eclipse Mosquitto, an open-source server implementation for **MQTT**, and Robot Operating System 2 (**ROS2**), a set of software libraries to build robot applications, which uses **DDS** for message serialization and transport. The two files generated in the previous part, JSON and XML, are now used to configure the protocol translator scripts. A **ROS2** workspace was created to store and run the protocol translators and device simulation scripts.

For the user interface, SCADA-like software was adopted using Node-RED to create a dashboard for real-time monitoring and simple command execution, such as running simulation scripts.

The **SCADA** system gathers data from the devices via the **OPC UA** server, keeping track of their current state.

The final system consists of an **OPC UA** server that communicates with the protocol translators - these translators take the source protocol and translate it to **OPC UA** -, a user interface to monitor and execute commands, an **AAS** editor to represent the assets digitally, and software to create models based on the IIoTOnTo ontology.

The authors tested the system twenty times, each lasting twenty minutes, by running the simulation scripts. The results demonstrated low average CPU and RAM usage - below 10% and 17%, respectively - as well as low protocol translation times. As expected, the time between a publisher sending a message and a subscriber receiving it was lower when no protocol translation had to be made.

This solution is identical to NextGenGW, but it has other capabilities such as asset modeling using **AASs** and interaction with ontologies, which do not fit NextGenGW's purpose at the moment. However, as noted in the paper, the authors didn't test the system for scalability by increasing the number of connected devices, leaving us with uncertainty in this regard.

2.3.2 Enabling Industry 4.0 Communication Protocol Interoperability: An OPC UA Case Study

This solution takes advantage of the **OPC UA** popularity and adoption to integrate with protocols, especially legacy ones. Moreover, **OPC UA** is the recommended communication protocol among machines, robots, and appliances by Reference Architecture Model Industry 4.0 (**RAMI**). For this purpose, they developed an intermediate language, a platform-independent type system, and a set of communication primitives. They demonstrate the results between the **HTTP** and the **OPC UA** protocols, using a gateway speaking the intermediate language based on the *Create, Read, Update, Delete*, and *Invoke* Virtual Automation Bus (**VAB**) primitives, described in Table 2.7 [32].

Primitive	Description
Ref create(elementId, name, value)	Creates a new property in a VAB element with parent element <i>elementId</i> , with name <i>name</i> and value <i>value</i> , returning a reference to the created element.
Ref delete(elementId, name)	Deletes a property named <i>name</i> of the VAB element with <i>elementId</i> .
Any retrieve(elementId, name)	Gets the value of a property named <i>name</i> of an element with id <i>elementId</i> .
void update(elementId, name, value)	Changes the value of a property with name <i>name</i> of element with id <i>elementId</i> to the new value <i>value</i> .
Any invoke(elementId, name, parameter)	Invokes operation with name <i>name</i> with parameter array <i>parameter</i> of element with id <i>elementId</i> .

Table 2.7: VAB primitives [6]

Based on the Industry 4.0 **VAB** architecture, their solution defines a type system to which the protocols' types must be mapped. **VAB** defines the *String*, *Boolean*, *Integer*, *Double*, *Map<String, Object>*, and *List<Object>* types. *Maps* and *Lists* may be nested arbitrarily.

Mapping the **VAB** primitives to **HTTP** verbs is almost direct. Table 2.8 shows the mappings used by the authors.

VAB Primitive	HTTP Verb
create(elementId, name, value)	HTTP PUT
delete(elementId, name)	HTTP DELETE
retrieve(elementId, name)	HTTP GET
update(elementId, name, value)	HTTP PUT
invoke(elementId, name, parameter)	HTTP POST

Table 2.8: VAB primitives to HTTP verbs mappings (from [32])

Now, the type mapping between **OPC UA** and **VAB** and the mapping between **VAB** primitives and **OPC UA** services need to be done. **OPC UA** defines twenty-five built-in data types, but the authors only picked the most relevant to the **VAB** specification. Table 2.9 describes the type mappings performed.

VAB Type	OPC UA Type	Node Class
String	PropertyType	Variable with data type UA_String
Boolean	PropertyType	Variable with data type UA_Boolean
Integer	PropertyType	Variable with data type UA_Integer
Double	PropertyType	Variable with data type UA_Double
Map<String, Object>	VABMapType	Object
List<Object>	VABListType	Object

Table 2.9: VAB to OPC UA type mappings (from [32])

To represent the Map and List **VAB** types, the authors defined special *ObjectTypes*, named *VABMapType* and *VABListType*. The elements of a *Map* are children of the *Object* node, which can have an arbitrary number of children. The parent-child relationship is defined using the *HasEntry* reference type, which is a subset of the *HasComponent* type since the elements of the map do not need to be ordered. The Map keys are annotated with the node's *BrowseName*.

The elements of a *List*, in this case, are ordered. Therefore, the parent-child relationship is defined using the *HasElement* type, a subtype of the *HasOrderedComponent* type. An element's index is annotated as the node's *BrowseName*.

To modify the attributes of a node, a unique *NodeId* is necessary. The authors took the *BrowsePath*, which is an ordered collection of *BrowseNames*, and transformed it into a *NodeId*, thanks to the **OPC UA** service *TranslateBrowsePathsToNodeIds*.

Finally, only the mappings between **VAB** primitives and **OPC UA** services are missing. The authors performed the ones described in Table 2.10.

VAB Primitive	OPC UA Service
create(elementId, name, value)	addNodes(id); write(nodeId, value)
delete(elementId, name)	deleteNode(nodeId)
retrieve(elementId, name)	read(nodeId, value)
update(elementId, name, value)	write(nodeId, value)
invoke(elementId, name, parameter)	call(nodeId, InArgs, OutArgs)

Table 2.10: VAB primitives to HTTP verbs mappings (from [32])

To evaluate the proposed solution, the authors used open62541. The **HTTP** client was built using Java, based on the open-source Apache Tomcat. The use case modeled various sensors of a gas tank of a wet glue manufacturing plant from the DigiPro4BaSys project. Essentially, on one side, they had the Industry 4.0 application and the **OPC UA** server on the other, having the gateway in the middle. They measured the Round Trip Time (**RTT**) for **HTTP** and **OPC UA** using an average of 500 samples and compared the results. The **HTTP** GET corresponds to the **OPC UA** read, and the **HTTP** PUT corresponds to the **OPC UA** write.

The results demonstrated that additional delay was introduced by **HTTP** on **OPC UA** services. Moreover, the translation service from *BrowsePaths* to *NodeIds* contributed significantly to the overhead. Finally, the implementation of the **OPC UA** server can also contribute to the delay, especially the way the server resolves *BrowsePaths*. However, this solution fulfilled its purpose, enabling communication between **HTTP** and **OPC UA**.

This is an interesting solution that we can easily compare to NextGenGW. The intermediate language proposed in this paper corresponds to **IETF SDF** in NextGenGW; then we have different protocols interacting with one another, just like we have **MQTT** and the NextGenGW-supported protocols. However, this solution focuses specifically on the intercommunication between **OPC UA** and **HTTP**, while with NextGenGW we can support as many protocols as we want by developing a translator between the desired protocol and **IETF SDF**.

2.3.3 TSN-compatible Industrial Wired/Wireless Multi-protocol Conversion Mechanism and Module

This solution aims to handle communication between wired and wireless protocols. The wired protocols supported are PROFINET, Ethernet/IP, and EtherCAT; the wireless protocols supported are ZigBee and LoRa. Additionally, they propose a Time-Sensitive Networking (**TSN**)-compatible frame with dynamic priorities [12]. **TSN** is a set of standards designed to provide guaranteed packet transport, low packet loss and packet variation, and limited latency through IEEE 802 networks [46].

The conversion module consists of three parts: the Protocol Layer, the Conversion Layer, and the Application Layer. There are two additional layers outside the module, each on opposite sides, while the conversion module is in the middle. They are the Operational Technology (OT) Device Layer, on one side, and the Edge Layer on the other.

The OT Device Layer is where devices such as PLCs and robots are located and where the data comes from. The Edge Layer contains devices like gateways and TSN switches. The data flow starts from an OT device, reaches the conversion module, and is sent to a device in the Edge Layer after processing.

The conversion module's Protocol Layer is responsible for implementing various protocols, with two interfaces: one connecting to the OT Device Layer and the other sending data to the upper layers. The Conversion Layer is where the actual conversion between protocols is performed. Lastly, the Application Layer contains several servers and interfaces. A TCP/IP server is set for data transmission; a configuration interface is set to allow the configuration of rules for protocol conversion and the configuration of the TSN-compatible frame. Figure 2.3 shows the system architecture.

The TSN-compatible frame adjusts the priority in the VLAN tag using an algorithm that takes as input a pre-assigned priority and the processing delay calculated on the conversion module. In the VLAN configuration frame, presented in Figure 2.4, the *timeout* field specifies how long the data frames are available in the conversion module. This value is also used in the algorithm to calculate dynamic priorities since it is assumed that the maximum processing delay equals the *timeout*.

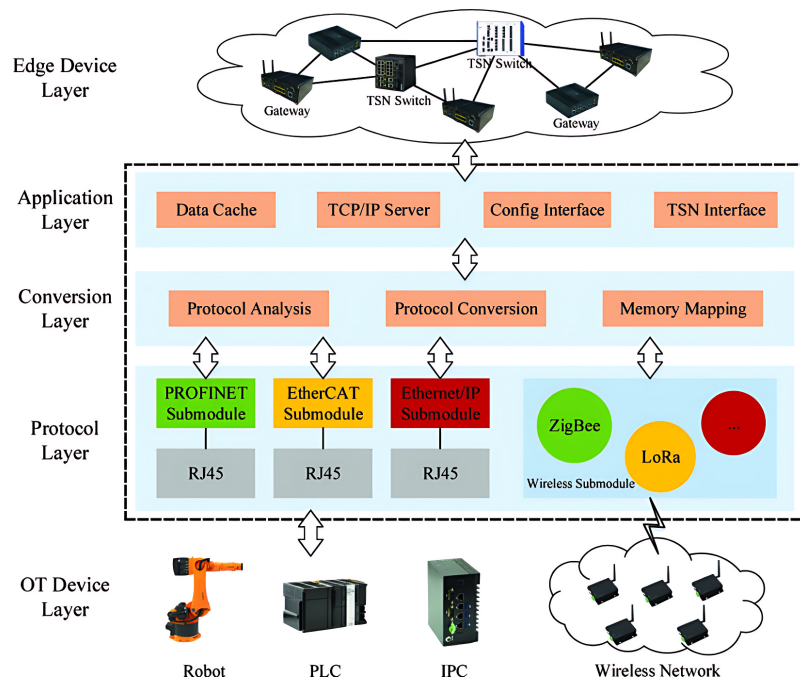


Figure 2.3: System architecture (from [12])

CMD	srcPro	timeout	vid	prePrio
-----	--------	---------	-----	---------

Figure 2.4: VLAN configuration frame (from [12])

The authors assembled the scenario in Figure 2.5 to test the solution. It has three I/O device lines, each having a Programmable Logic Controller (PLC) and an I/O device for Ethernet/IP, EtherCAT, and PROFINET. It also has two ZigBee and two LoRa nodes, one PC, one computer to monitor network traffic using Wireshark, and a TSN switch. An oscilloscope was used to measure the latency of protocol conversion.

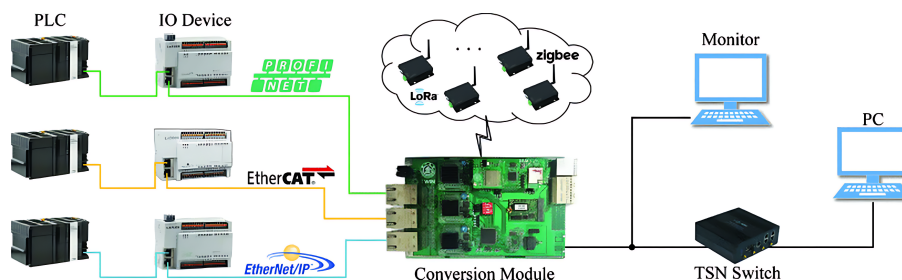


Figure 2.5: Conversion module test scenario (from [12])

After conducting 30 tests, the authors observed that the maximum delay was equal to 250ms. Attributing dynamic priorities through the TSN-compatible frame was also verified. After setting up the VLAN configuration rules for the protocols requiring deterministic communication, in this case, it was for PROFINET, it was observed that, after the processing delay became almost constant, the changing of the *timeout* field changed the priority in the VLAN.

This solution can be seen as a protocol translator with the added value of temporal guarantees provided by TSN. When compared to NextGenGW, this solution lacks semantic interoperability; however, they both support protocol conversion.

2.4 Conclusion

In this chapter, we described six communication protocols that are commonly used in the industry. From them, OPC UA was the selected one to proceed with in this study. Then, we went through the levels of interoperability, focusing on the semantic level and briefly introducing industrial ontologies. Finally, we reviewed different solutions targeting the interoperability problem.

Chapter 3

Enabling technologies

To address the compatibility of **IETF SDF** with industrial scenarios, we will take OPC UA, the chosen communication protocol in Chapter 2, check if there are already tools to convert its information model to **IETF SDF** and, if not, address this possibility and propose a conversion. Only then can we assume that **IETF SDF** can be applied to the industry; therefore, NextGenGW should also be capable of moving to industrial scenarios.

Before proposing any conversion, in Section 3.1, we will first dive into **IETF SDF**, exposing its purpose, core concepts, and relevant tools; then, in Section 3.2, we will describe NextGenGW, its architecture and functioning; finally, in Section 3.3, we will describe the main concepts of OPC UA and its information model. Only by understanding both **IETF SDF** and the OPC UA information model can we propose a mapping between the two. Understanding NextGenGW will be useful when addressing possible integrations.

3.1 IETF SDF

IETF SDF [[39] [38]] describes abstract data models and presents itself as a solution for the development of multiple data models in the **IoT** field. The goal is to align these data models by describing them in the same format.

IETF SDF focuses on describing *Things*, which are physical objects that may be interacted with over a network. It supports extensions for its base definition, allowing users to fine-tune it to fit their needs more closely. An example of this may be the creation of protocol bindings for specific scenarios.

IETF SDF uses **JSON** as its representation language, taking advantage of its features, from the existence of numerous parsers to the existence of a vast set of libraries and tools to produce and consume **JSON**.

The requirements that **IETF SDF** needs to fulfill are presented in Table 3.1. [38]

Index	Requirement
0	Support RESTful and action-based paradigms
1	Support data types
2	Transmit semantic information
3	Independent from the transport layer
4	Reusability of properties and components by multiple models

Table 3.1: IETF SDF requirements

In Subsection 3.1.1, we present the **IETF SDF** data model, going through each of its constituent groups. Then, in Subsection 3.1.2, we describe the structure of an **IETF SDF** document. In Subsection 3.1.3, we explain **IETF SDF** names and namespaces, and how interaction between them can be achieved. Finally, Subsection 3.1.4 presents existing tools to work with **IETF SDF**.

3.1.1 IETF SDF groups

IETF SDF is composed of six groups, and their interactions are presented in Figure 3.1. Both the *sdfThing* and *sdfObject* groups can be composed of zero or more *sdfProperty*, *sdfEvent*, and *sdfAction* groups. The main difference between them is that an *sdfThing* can contain other *sdfThing* definitions, while an *sdfObject* can't contain other *sdfObject* definitions. That's because *sdfThing* groups define more complex *Things*, composed of multiple objects.

We will explain *sdfData* groups now, because they can be used by all the others that are still to be described. *sdfData* groups define common modeling patterns, data constraints, and semantic concepts. In short, these groups define data types that can be reused in other **IETF SDF** groups, avoiding repetitions.

An *sdfAction* can be seen as method calls from a programmer's perspective. They are used to read, update, or observe a *Thing* state. *sdfAction* groups may have input and output data, which can be defined via *sdfData* groups. The input data can be seen as the arguments of a method, and the output data as the returned elements.

sdfEvent groups define meaningful *happenings* in a device model, i.e., movement detected or a door that opened.

sdfProperty groups model elements of *Things*, similar to fields in a class, from a programming perspective. Again, *sdfData* groups can be used in definitions of *sdfProperty* groups. **IETF SDF** provides qualities to constrain the structure and values of data in interactions modeled by *sdfProperty* groups, as well as qualities to associate semantics with the data. Associations with some protocol affordances can be achieved through named definition entries in *sdfProperty* groups, allowing for the application to obtain the state variable and, optionally, modify it.

The difference between *sdfProperty* changes and an *sdfEvent* is that events require ordering, reliability, and consistency. A state change may be overridden by another, but events need to be preserved even when other events occur.

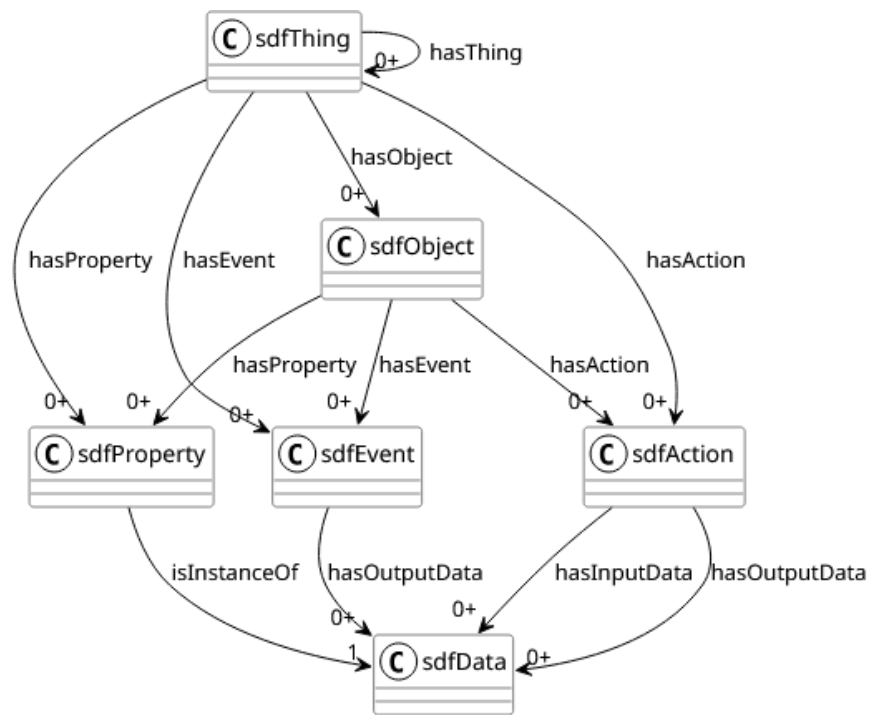


Figure 3.1: Relations between IETF SDF groups (from [38])

3.1.2 IETF SDF document structure

An **IETF SDF** document is composed of three blocks that can contain zero or more **JSON** name/-value pairs. Together, these three blocks define an **IETF SDF** model. It is possible to reference declarations and definitions from other **IETF SDF** documents.

3.1.2.1 Information block

The information block is used to provide metadata about the **IETF SDF** model itself, not about the device's behavior or properties. It's akin to a *header* or *profile* section. This block is typically found at the top of an **IETF SDF** file.

Its purpose is to describe who made the model, what it's about, its version, and other descriptive or administrative details.

The qualities of Table 3.2 are part of the information block.

Quality name	Description
title	Human-readable name of the model
description	A longer human-readable explanation
version	Version of the model in semver format
modified	Latest modification time
copyright	Link or embedded copyright notice
license	Link to the license
features	Extension features used
\$comment	Source code comments

Table 3.2: IETF SDF information block qualities

The *features* quality is used to list names of **IETF SDF** extension features to properly process **IETF SDF** definitions. Listing 3.1 presents the information block of a simple smart door sensor.

```
{
  "info": {
    "title": "Smart Door Sensor",
    "version": "1.0.0",
    "description": "A model for a simple open/closed door sensor.",
    "copyright": "2024 Acme Corp",
    "license": "https://spdx.org/licenses/MIT.html"
  }
}
```

Listing 3.1: IETF SDF information block example

3.1.2.2 Namespaces block

The namespaces block defines a set of prefixes and their corresponding URIs used for referencing reusable models or definitions in a consistent way. It also defines the default namespace.

The default namespace references an entry defined in the namespaces block by its short name.

Multiple **IETF SDF** documents can contribute to the same namespace if their default namespace is the same.

If no default namespace is specified, the definitions in that **IETF SDF** document are local and can't be accessed by other documents. Listing 3.2 shows a namespace block with two namespaces, with one being selected as the default one.

```
{
  "namespace": {
    "ns1": "https://example.com/ns1",
    "myNamespace": "https://example.com/myNamespace"
  },
}
```

```

"defaultNamespace": "ns1"
}

```

Listing 3.2: IETF SDF namespaces block example

3.1.2.3 Definitions block

The definitions block enables the definition of reusable **IETF SDF** groups to be referenced elsewhere in the model. It contains one or more groups defined by a Class Name keyword (*sdfObject*, *sdfProperty*). There can only be one group per keyword; therefore, we put all definitions in the group under that keyword, avoiding repeated keywords.

Each definition is a named set of qualities, and it can contain other definitions. In Listing 3.3, a definition of a simple *sdfObject* with one *sdfProperty* is presented.

```

{
  "sdfObject": {
    "myObject": {
      "sdfProperty": {
        "myProperty": {
          "type": "boolean"
        }
      }
    }
  }
}

```

Listing 3.3: IETF SDF definitions block example

3.1.2.4 Top-level affordances and sdfData

Even though *sdfProperty*, *sdfEvent*, *sdfAction*, and *sdfData* groups can be defined within an *sdfObject* or *sdfThing*, they can still be defined at the top level of the **IETF SDF** document. This way, they are not associated with any *sdfObject* or *sdfThing*, and their main purpose is to be reused via the *sdfRef* mechanism.

To sum up the information provided up until now, we present an example of an **IETF SDF** model in Listing 3.4. There, we can see a simple representation of an *sdfObject* named Switch with an *sdfProperty* "value" of type boolean and three *sdfActions*: on, off, and toggle.

```

{
  "info": {
    "title": "Example document for SDF (Semantic Definition Format)",
    "version": "2019-04-24",
    "copyright": "Copyright 2019 Example Corp. All rights reserved.",

```

```
"license": "https://example.com/license"
},
"namespace": {
  "cap": "https://example.com/capability/cap"
},
"defaultNamespace": "cap",
"sdfObject": {
  "Switch": {
    "sdfProperty": {
      "value": {
        "description":
          "The state of the switch; false for off and true for on.",
        "type": "boolean"
      }
    },
    "sdfAction": {
      "on": {
        "description":
          "Turn the switch on; equivalent to setting value to true."
      },
      "off": {
        "description":
          "Turn the switch off; equivalent to setting value to false."
      },
      "toggle": {
        "description":
          "Toggle the switch; equivalent to setting value to its complement."
      }
    }
  }
}
```

Listing 3.4: IETF SDF model example

3.1.3 IETF SDF names and namespaces

3.1.3.1 Global names

IETF SDF documents can contribute and reference elements from a global namespace. Global names are *https://* URIs followed by fragment identifiers.

Listing 3.5 is an example of a global name: [38]

```
https://example.com/capability/cap#/sdfObject/Switch
```

Listing 3.5: IETF SDF global name example

The first part, the absolute-URI, of the global name is the same as the default namespace. The second part, the fragment identifier, is constructed from a JSON pointer to the element defined in the **IETF SDF** document. The # character separates both parts.

A definition for a name always targets the default namespace. That's why it is also called the target namespace.

To reference a name of the same **IETF SDF** document, we can use an # followed by a JSON pointer. If the name is from an external **IETF SDF** document, then a prefix is needed.

If we have a namespace prefix defined [38], say:

```
"namespace": {
  "foo": "https://example.com/"
}
```

Listing 3.6: IETF SDF namespace definition

We can have a reference to that namespace:

```
"sdfRef": "foo:#/sdfData/temperatureData"
```

Listing 3.7: IETF SDF namespace reference

This will result in:

```
"https://example.com/#/sdfData/temperatureData"
```

Listing 3.8: IETF SDF resolved namespace reference

Name references only occur in two scenarios:

- when copying elements with *sdfRef* values
- when pointing to elements with *sdfRequired* value elements

Both *sdfRef* and *sdfRequired* are explained next.

3.1.3.2 sdfRef

This keyword is used to reference a definition and copy all its qualities and enclosed definitions. Consider the following reference in Listing 3.9: [38]

```
"temperatureProperty": {
  "sdfRef": "#/sdfData/temperatureData"
}
```

Listing 3.9: IETF SDF *sdfRef* example

This creates a new definition with all the qualities and definitions of the */sdfData/temperatureData* property. It is possible to have additional members on the map. If there is an element equal to a referenced one, the **JSON Merge Patch** algorithm is used to resolve conflicts. It is also possible to have nested references, which all need to be resolved.

When all references are processed, we end up with a resolved model.

3.1.3.3 sdfRequired

The *sdfRequired* keyword is used to define which declarations are mandatory in an *sdfThing* or *sdfObject*. The value is an array of references, where each one indicates a declaration that needs to be represented.

Take Listing 3.10 as an example: [38]

```
"sdfObject": {
  "temperatureWithAlarm": {
    "sdfRequired": [
      "#/sdfObject/temperatureWithAlarm/sdfProperty/currentTemperature",
      "#/sdfObject/temperatureWithAlarm/sdfEvent/overTemperatureEvent"
    ],
    "sdfData": {
      "temperatureData": {
        "type": "number"
      }
    },
    "sdfProperty": {
      "currentTemperature": {
        "sdfRef": "#/sdfObject/temperatureWithAlarm/sdfData/temperatureData",
        "writable": false
      }
    },
    "sdfEvent": {
      "overTemperatureEvent": {
        "sdfOutputData": {
          "sdfRef": "#/sdfObject/temperatureWithAlarm/sdfData/temperatureData"
        }
      }
    }
  }
}
```

Listing 3.10: IETF SDF *sdfRequired* example

Or in a shorter form:

```
"sdfRequired": ["currentTemperature", "overTemperatureEvent"]
```

Listing 3.11: IETF SDF *sdfRequired* shorter form

Or even:

```
"overTemperatureEvent": {
  "sdfRequired": [true],
  "...": "..."
}
```

Listing 3.12: IETF SDF *sdfRequired* alternative representation

3.1.4 IETF SDF tools

OneDM, the original owner of **IETF SDF**, has a GitHub repository [40] with references to multiple **IETF SDF** tools. These include **IETF SDF** converters, none targeting OPC UA, which are presented in Table 3.3. This is crucial for this thesis: if any conversion between OPC UA and **IETF SDF** already exists, then its suitability for industrial scenarios is proven.

Converter	Description
IPSO /Lightweight Machine-to-Machine (LwM2M)	Converts to/from IETF SDF
OCF	Converts IETF SDF to OCF
Web of Things	Converts IETF SDF to Web of Things
YANG	Converts to/from IETF SDF
DTDL	Converts to/from IETF SDF (early implementation)
Matter	Converts to/from IETF SDF

Table 3.3: IETF SDF existing converters

3.2 NextGenGW

NextGenGW [51] is a middleware gateway solution to enhance interoperability in **IoT** environments. It targets the syntactic and semantic sides of device communication to tackle the challenges posed by the heterogeneity of **IoT** devices.

NextGenGW uses **IETF SDF** combined with the **MQTT** protocol. By using these technologies, it is expected to support scenarios with multiple communication protocols and to enable communication between them. **IETF SDF**, being an abstract data modeling language by nature, is seen as a great candidate for multiple protocol support. This feature may allow for the translation of information models employed by industrial communication protocols to **IETF SDF**.

NextGenGW currently supports two communication protocols: IoTivity and LwM2M. This thesis aims to assess its integration with an industrial communication protocol, marking its shift to the industrial field instead of the regular IoT scenarios.

3.2.1 System architecture

NextGenGW employs a communication endpoint per communication protocol, where this protocol is abstracted. The endpoints interact with the gateway via two components: a broker and a translator. These form the Abstraction Layer. The broker is responsible for communicating with the IoT node; the translator converts the communication protocol's data to IETF SDF and vice-versa. The data in IETF SDF format is passed to the *DownStream Manager*, making it available to external applications via an MQTT broker. External application requests are delegated to the correct translator by the *DownStream Manager*; then, the broker communicates with the IoT end node to execute the desired operation.

With reliability and scalability in mind, NextGenGW supports the use of Extenders. These can be devices with lower computing power that extend NextGenGW's communication range and allow for IoT nodes in shadowed areas to connect with it.

When there are only devices with limited capabilities near the IoT nodes, the Abstraction Layer can be deployed in Extenders that are close to the limited devices. The third-party applications are deployed in Edge Servers and, if communication between the Extenders and third-party applications is not possible due to the network architecture, an MQTT broker can act as a bridge between both parties.

To interact with IoT nodes, third-party applications need to know that these exist. NextGenGW provides a discovery mechanism to solve this. A third-party application publishes a request to the "discovery" MQTT topic. Then, the MQTT client in the Abstraction Layer, which is subscribed to this topic, receives the request and forwards it to each translator and broker so they can pass it to the IoT end nodes. The response is returned in IETF SDF format to the MQTT client that creates a topic for each resource available. Figure 3.2 provides a visual representation of NextGenGW's architecture.

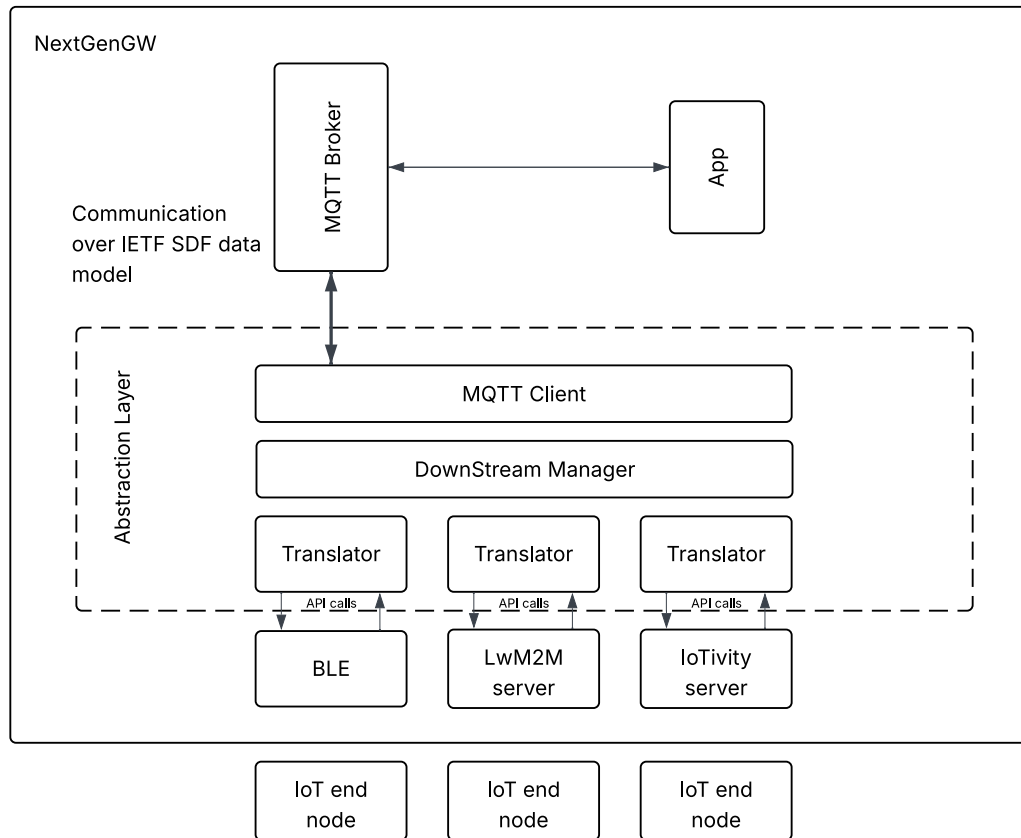


Figure 3.2: NextGenGW system architecture (from [51])

3.2.2 MQTT-IETF SDF bindings

The **MQTT** broker starts with the "announce" and "discovery" topics. Whenever a request with no payload is received in the "discovery" topic, the discovery mechanism for all **IoT** nodes is started. The discovered ones are added to the system and shared with the third-party applications via the "announce" topic, following the format of the **JSON** payload in **IETF SDF** format presented in Listing 3.13. **IoT** nodes with announcement capabilities are also announced in the "announce" topic.

```
{
  "<thing_identifier>": {
    "sdfObject": {
      "<object_name>": {
        "sdfProperty": {
          "<prop_name>": "<prop_value>", (...)
        },
        "sdfAction": {
          "<action_name>": "<action_value>", (...)
        },
        "sdfEvent": {
          "<event_name>": "<event_value>", (...)
        }
      }
    }
  }
}
```

```

        }
      }, (... )
    }
  }
}

```

Listing 3.13: IoT node discovery response

Each connected **IoT** node is associated with dedicated **MQTT** topics following the logic below, allowing for interaction at the object, property, action, and event level: [51]

- <thing identifier>/<object name>
- <thing identifier>/<object name>/<prop name>
- <thing identifier>/<object name>/<action name>
- <thing identifier>/<object name>/<event name>

It is possible to interact at these levels by publishing a message to these topics with a payload that defines the type of operation to be performed and the data that the operation targets. The operations supported are: READ, POST, PUT, DELETE, OBSERVE_START, and OBSERVE_STOP. Listing 3.14 shows the structure of an object-level interaction request payload.

```

{
  "operation": "<request>",
  "data": {
    "sdfProperty": {
      "<prop_name>": "<prop_value>", (... )
    },
    "sdfAction": {
      "<action_name>": "<action_value>", (... )
    }
  }
}

```

Listing 3.14: Object-level interaction request payload

The payload for a property, event, or action-level interaction follows the structure in Listing 3.15.

```

{
  "operation": "<request>",
  "data": "<value>"
}

```

Listing 3.15: Property, event or action-level interaction request payload

The response for an interaction is sent in the response topic specified along with the published message, available in **MQTT** version 5.

3.3 OPC UA

OPC UA was the selected protocol for the **IIoT** compatibility study, given the reasons already pointed out at the end of Chapter 2. The use cases, advantages, and disadvantages of this protocol were already described in Subsection 2.1.4 of the same chapter. Therefore, this section will strictly focus on its information model.

3.3.1 OPC UA information model fundamentals

The OPC UA information model follows a graph-based paradigm constituted by *Nodes* that are connected via *References*. It defines a concept called the *AddressSpace*, which is simply a way for OPC UA *Servers* to represent *Objects* to *Clients*, using the OPC UA Object Model to accomplish this. Figure 3.3 presents this model.

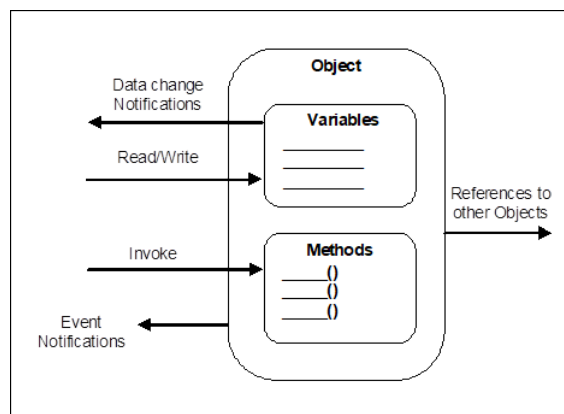


Figure 3.3: OPC UA *Object* Model (from [21])

Nodes represent elements of this model in the *AddressSpace*. Each *Node* is of a given *Node-Class* that represents an element of the Object Model. A *Node* is described by *Attributes* and is connected to other *Nodes* by *References*, as shown in Figure 3.4.

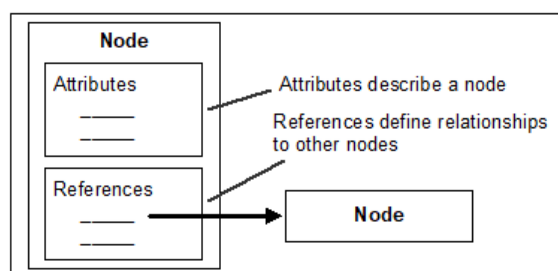


Figure 3.4: OPC UA *Node* Model (from [21])

Attributes describe *Nodes* and can be accessed by *Clients* through *Services*.

References relate *Nodes* to each other and can also be accessed by *Clients*. The *Node* containing the *Reference* is called the *SourceNode*, and the referenced *Node* is called the *TargetNode*. To identify a *Reference*, OPC UA uses the *SourceNode*, the *TargetNode*, and the *ReferenceType*. That's why a *Node* can't have more than one *Reference* of the same type pointing to the same *TargetNode*. Figure 3.5 presents this Relation Model.

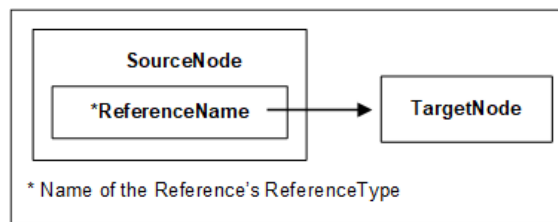


Figure 3.5: OPC UA *Reference* Model (from [21])

Variables represent values and can be of two types: *Properties* and *DataVariables*. *Properties* characterize *Objects*, *DataVariables*, and other *Nodes*. They differ from *Attributes* in the sense that these are defined for every *Node* of a *NodeClass*, while *Properties* can be server-defined. *Property* values don't change often and a *Property* can't contain other *Properties*.

DataVariables, on the other hand, can contain other *Variables* - other *DataVariables* or *Properties*. *DataVariables* are used to represent something from the real world, i.e., a pressure measurement, while *Properties* are usually used to represent values that describe a *Node*, i.e., an engineering unit of a *Variable*.

OPC UA *Objects* and *Variables* must have a type definition. For that purpose, *References* of type *HasTypeDefinition* are used. Whenever a more specific type exists, the OPC UA server can use the *BaseObjectType* and the *BaseDataVariableType*. These are described in Subsection 3.3.3 and Subsection 3.3.4, respectively.

Figure 3.6 shows how a *Variable* is linked to its type definition.

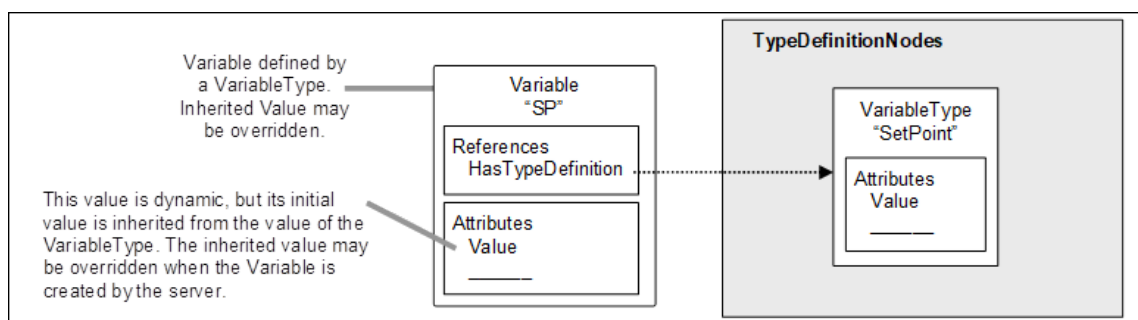


Figure 3.6: OPC UA *Variable* defined by a *VariableType* (from [21])

OPC UA has an *Event Model* to support subscription and emission of events. The occurrence of *Events* is reported through *Event Notifications*. The *Events* are not exposed directly in the *Address Space* and, to subscribe to an *Event*, *Objects* and *Views* can be used. The *EventNotifier*

Attribute of a *Node* indicates whether a *Node* can subscribe to *Events*. An *Event* is of a specific *EventType*.

Methods are functions that can be invoked by *Clients* and processed in *Servers* that return the result to the caller *Client*. These *Methods* are stateless in the sense that they don't have a state of their own. However, they can change the state of their owning *Object*.

Methods can have multiple input arguments and output results. They are represented through a *Node* of the *Method NodeClass*, where the definition of the input arguments, result, and behavior is defined. *Methods* are invoked by the *Call Service*.

3.3.2 OPC UA common attributes

OPC UA defines common attributes for *Nodes* of every *NodeClass*. These are generic attributes, like the *DisplayName* and the *Description*. They are explained in Table 3.4.

Attribute	Value
DisplayName	LocalizedText that each server should provide identical to the <i>BrowseName</i> of the <i>Node</i> for the en locale.
Description	Optional server-specific <i>Node</i> description.
NodeClass	The <i>Node</i> 's <i>NodeClass</i> .
NodeId	Represented by <i>BrowseNames</i> .
WriteMask	Optional <i>Attribute</i> that, if specified, sets non-server-specific <i>Attributes</i> to non-writable.
UserWriteMask	Optional <i>Attribute</i> following the same rules as the <i>WriteMask</i> .
RolePermissions	Optional server-specific permissions.
UserRolePermissions	Optional permissions for the current Session. It depends on the <i>RolePermissions Attribute</i> and the current Session. It is server-specific.
AccessRestrictions	Optional server-specific access restrictions.

Table 3.4: OPC UA *Node* common *Attributes* (from [22])

A single *Attribute* is defined for all OPC UA *Objects*. It is presented in Table 3.5.

Attribute	Value
EventNotifier	Specifies if a <i>Node</i> can be used to subscribe to <i>Events</i> .

Table 3.5: OPC UA *Object* common *Attributes* (from [22])

OPC UA *Variable* attributes must be defined according to Table 3.6, and *variableTypes* according to Table 3.7.

Attribute	Value
MinimumSamplingInterval	Optional <i>Attribute</i> that specifies a server-specific minimum sampling interval.
AccessLevel	The access level for type definition <i>Variables</i> is server-specific; other <i>Variables</i> should allow reading; other settings are server-specific.
UserAccessLevel	Server-specific <i>Attribute</i> . It is assumed that all <i>Variables</i> can be accessed by at least one User.
Value	The value of <i>InstanceDeclaration Variables</i> is server-specific. The others represent the value described in the text.
ArrayDimensions	If the <i>ValueRank</i> does not specify a dimension, this <i>Attribute</i> can be set to null or missing; this is server-specific. If a <i>ValueRank</i> is specified, this <i>Attribute</i> must be defined.
Historizing	Server-specific <i>Attribute</i> .
AccessLevelEx	If this <i>Attribute</i> is defined, then the bits 8, 9, and 10 must be set to 10, representing atomic read and write operations on an individual <i>Variable</i> ; arrays can be partially written.

Table 3.6: OPC UA *Variable* common *Attributes* (from [22])

Attribute	Value
Value	Optional server-specific default value.
ArrayDimensions	If the <i>ValueRank</i> does not specify a dimension, this <i>Attribute</i> can be set to null or missing; this is server-specific. If a <i>ValueRank</i> is specified, this <i>Attribute</i> must be defined.

Table 3.7: OPC UA *VariableType* common *Attributes* (from [22])

Finally, Table 3.8 shows the common *Attributes* for OPC UA *Methods*.

Attribute	Value
Executable	All <i>Methods</i> should be executable (this value is set to "True") unless it is defined differently in the <i>Method</i> definition.
UserExecutable	Server-specific <i>Attribute</i> . It is expected that <i>Methods</i> can be executed by at least one User.

Table 3.8: OPC UA *Method* common *Attributes* (from [22])

3.3.3 OPC UA standard *ObjectTypes*

OPC UA defines standard *ObjectTypes*. These *ObjectTypes* can be extended with additional components. The *BaseObjectType* is a base for the other *Objects* since they all inherit directly or

indirectly from it. This *Type* shouldn't be used by *Servers*, being more specific *Types* preferred.

ObjectTypes are used to represent *Server* capabilities and *EventTypes*. The number of items defined in OPC UA is large, so we will not mention them here. They are described in detail in [22].

3.3.4 OPC UA standard VariableTypes

Standard *VariableTypes* are also specified. Similar to the *BaseObjectType*, there is a *BaseVariableType* here that serves as a base for the other *VariableTypes*. However, only the *PropertyType* and the *BaseDataVariableType* directly inherit from the *BaseVariableType*.

The *PropertyType* defines the type for all *Properties*. Subtyping this *Variable* is not allowed.

The *BaseVariableType* defines the type of *DataVariables* that don't have a more concrete type definition available. This is the base *VariableType* for *DataVariables VariableTypes*.

There are more *VariableTypes* defined in this protocol that can be found in [22].

3.3.5 OPC UA standard Methods

OPC UA defines four standard *Methods*. They are briefly presented in Table 3.9.

Name	Description
GetMonitoredItems	Gets information about the monitored items of a subscription.
ResendData	Gets the current values of the data monitored items of a <i>Subscription</i> where the <i>MonitoringMode</i> is set to <i>Reporting</i> .
SetSubscriptionDurable	Sets a <i>Subscription</i> into a mode where the data and event queues are persisted if the <i>Client</i> is disconnected or the <i>Server</i> is restarted.
RequestServerStateChange	Allows a <i>Client</i> to request a <i>Server</i> state change.

Table 3.9: OPC UA standard *Methods* (from [22])

3.3.6 OPC UA standard ReferenceTypes

OPC UA defines various *ReferenceTypes*, such as *References*, *Organizes*, *HasChild*, and *Aggregates*. The complete list and detailed description can be found in [22].

3.3.7 OPC UA standard DataTypes

OPC UA defines multiple *DataTypes*. It includes an abstract *BaseDataType* and common data types like *Boolean*, *Number*, *String*, *DateTime*, *Enumeration*, and *Structure*. Other *DataTypes* defined are the *ByteString*, *Image*, *Guid*, *LocalizedText*, *NodeId*, *QualifiedName*, and *XmlElement*.

3.4 Conclusion

We are now familiar with both the OPC UA information model and **IETF SDF**. With this knowledge, we are able to assess if they are compatible. In that case, we can search for any existing conversion and, if none is found, we can propose our own. Knowing how NextGenGW is implemented, we can also propose integrations between it and OPC UA. This is the work developed in Chapter 4.

Chapter 4

Solution

This chapter introduces our proposed solution to assess the compatibility between the OPC UA information model and **IETF SDF**. Section 4.1 presents the theoretical mappings between these models, where we associate their elements conceptually. Section 4.2 details our effort in creating a translation between the OPC UA information model and **IETF SDF**, addressing the research question **RQ2**. Section 4.3 answers the research question **RQ3** by presenting how interactions with an OPC UA server are performed using the **MQTT-IETF SDF** bindings proposed by NextGenGW.

4.1 IETF SDF theoretical mappings

Considering the study performed on **IETF SDF** (see 3.1) and the OPC UA information model (see 3.3), we can draw parallels between their building blocks. Table 4.1 presents these mappings.

IETF SDF group	OPC UA mapping
<i>sdfThing</i>	OPC UA server IP address plus <i>Node</i> 's namespace index
<i>sdfObject</i>	<i>ObjectType Node</i> and <i>Object Node</i>
<i>sdfAction</i>	<i>Method Node</i>
<i>sdfEvent</i>	<i>Event Node</i>
<i>sdfProperty</i>	<i>Property</i> or <i>DataVariable Node</i>
<i>sdfData</i>	<i>DataType Node</i>

Table 4.1: IETF SDF to OPC UA information model theoretical mappings

Both *sdfObject* groups and OPC UA *Object Nodes* represent generic objects, which can be, for example, physical devices such as motors, pumps, or sensors. *sdfObject* groups can have child *sdfAction*, *sdfProperty*, and *sdfEvent* groups, similar to OPC UA *Object Nodes* that can have references to OPC UA *Property*, *DataVariable*, *Method*, and *Event Nodes*. *sdfProperty* groups and OPC UA *Property* and *DataVariable Nodes* characterize *Objects*. *DataVariable Nodes* may have child *Property Nodes*, but the same could be achieved with an *sdfProperty* group with a complex type. *sdfAction* groups declare *Actions* to read and update the state of a *Thing* and can

be compared to OPC UA *Method Nodes*, although these may be more complete and support more interactions. They both can have input and output arguments. *sdfEvent* groups model occurrences, defining the output of the modeled *Event*. OPC UA *Event Nodes* also model occurrences and have an *EventType* type definition. *sdfData* groups and OPC UA *DataType Nodes* define data types.

All of these mappings are intuitive and conceptually similar, but there is one **IETF SDF** group missing: the *sdfThing*. We decided to map the *sdfThing* to the *Node*'s OPC UA server IP address plus the *Node*'s namespace index. The rationale behind it is that OPC UA servers define *Objects*, which would make the OPC UA server the *sdfThing*, the only group that is higher than the *sdfObject* in the **IETF SDF** hierarchy. Instead of only saving the OPC UA server IP address, we found it useful to also save the *Node*'s namespace index, since the OPC UA *Node* is part of one namespace. In scenarios consisting of only one OPC UA server, its IP address may be omitted, using the *Node*'s namespace index only.

From all this information, it seems that **IETF SDF** and OPC UA are compatible. The only concept that seems more misaligned between the two is the *sdfThing*, which doesn't map to an OPC UA *Node*, unlike the other **IETF SDF** groups.

From the mappings presented, we will evaluate only the subset presented in Table 4.2. This leaves *sdfEvent* groups, and OPC UA *Event* and *DataVariable Nodes* for future analysis.

IETF SDF group	OPC UA mapping
<i>sdfThing</i>	OPC UA server IP address with <i>Node</i> 's namespace index
<i>sdfObject</i>	<i>ObjectType Node</i> and <i>Object</i> of type <i>ObjectType</i>
<i>sdfAction</i>	<i>Method Node</i>
<i>sdfProperty</i>	<i>Property Node</i>
<i>sdfData</i>	<i>DataType Node</i>

Table 4.2: IETF SDF to OPC UA information model evaluated mappings

4.2 IETF SDF - OPC UA information model conversions

For the conversion between **IETF SDF** and the OPC UA information model, we explored the use of the existing converters provided by OneDM (see Section 3.1.4). These converters only allow an indirect conversion from **IETF SDF** to the OPC UA information model, which is detailed in Subsection 4.2.1. For this reason, we had to explore a different strategy for the conversion in the opposite direction, which is detailed in Subsection 4.2.2. The conversion in the **IETF SDF** to OPC UA direction isn't too relevant for NextGenGW. This is because NextGenGW only needs to generate **IETF SDF** documents from the data available in the end nodes at the initialization phase, when it is discovering them. We still present this conversion as a contribution to the community. Moreover, the conversion results are helpful to confirm the mappings proposed and to create a converter in the opposite direction.

4.2.1 IETF SDF to OPC UA information model

While exploring the converters provided by OneDM, we found it was possible to use one of them to convert **IETF SDF** to the OPC UA information model in a multi-step conversion approach. This approach consists of converting **IETF SDF** to **YANG** using *sdf-yang-converter* - the converter provided by OneDM - and then converting **YANG** to the OPC UA information model, using *pyang*. We address each of these steps separately in 4.2.1.1 and 4.2.1.2. Figure 4.1 provides an overview of this process, where some extra steps are presented after the use of *pyang*, which are explained in 4.2.1.2.

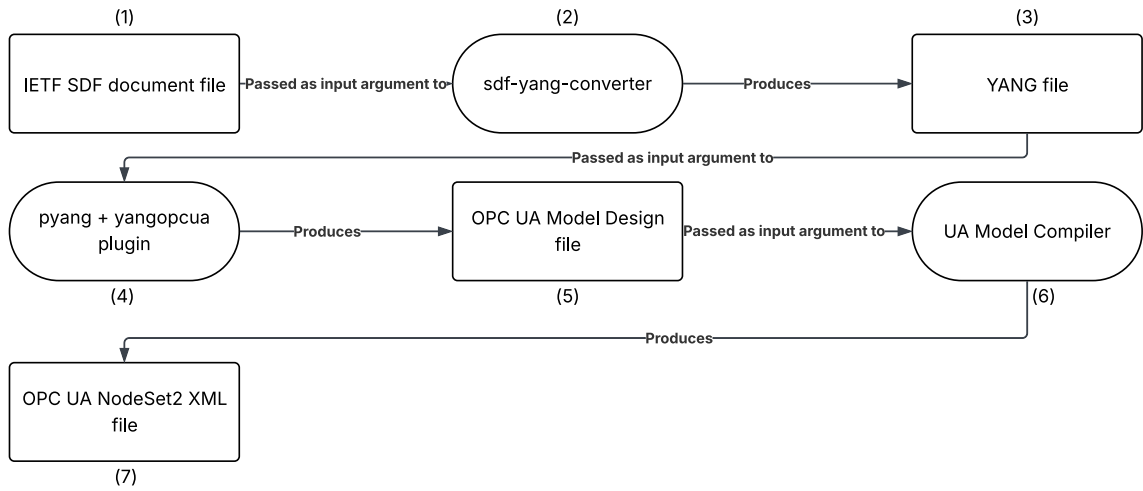


Figure 4.1: IETF SDF to OPC UA conversion process

4.2.1.1 IETF SDF to YANG

IETF SDF and **YANG** follow the same hierarchy-based paradigm, making it easier to map one to another. The *sdf-yang-converter* (2) tool [33] handles bi-directional conversions, and relies on some dependencies: it uses the *yang* repository [41] to load the context of **YANG** files; *libyang* [11] to parse **YANG** files; *nlohmann/json* [43] to parse **JSON** files; *pboettch/json-schema-validator* [9] to validate the generated **IETF SDF JSON** files, using a validation schema.

After setting up all the required dependencies and the converter, we are ready to run it. We will pass the example **IETF SDF** file (1) provided in the **IETF SDF** specification [39] as input argument to *sdf-yang-converter*. This file can be consulted in Appendix A.1.

Running the converter, we obtained the generated **YANG** file (3) in Appendix A.2, which was correctly generated. However, we immediately noticed that the conversion took considerable time, taking around 4 minutes. To reduce the execution time, we tried to turn off the validations performed after the output files were generated, but we didn't find it helpful. Therefore, the high execution time is due to the converter implementation itself.

Next, in Subsection 4.2.1.2, we will convert the generated **YANG** file to an OPC UA model file.

4.2.1.2 YANG to OPC UA information model

After the generation of the **YANG** file, we move on to the next step, which is converting **YANG** to the OPC UA information model. This step consists of 2 steps: the first takes a **YANG** file and generates an OPC UA Model Design XML file. This kind of file was designed to offer a better user experience, since it is more human-readable than the NodeSet2 XML file that OPC UA servers consume. In the second step, we take this file and generate a NodeSet2 XML file.

Firstly, we use *pyang* [8], a **YANG** validator and converter which is extensible through plugins. *yangopcua* [53] offers one to ease the conversion from **YANG** to the OPC UA information model. However, it is noted by the author that this plugin does not come with *pyang* by default due to the need for complete validation.

Using *pyang* (4), we pass the **YANG** file as input, and it generates the OPC UA Model Design XML file (5), which can be found in Appendix A.3. This takes much less time than the previous step, being almost instant.

OPC UA servers consume NodeSet2 XML files, and here's when the next step comes in. The OPC Foundation developed a tool, the *UA Model Compiler* [24], which takes an OPC UA Model Design XML file and generates C# and ANSI C code, a NodeSet2 XML file, as well as other files that won't be needed for this use case. Passing the generated OPC UA Model Design XML file as input to *UA Model Compiler* (6), we get the OPC UA NodeSet2 XML file (7) in Appendix A.4, concluding the conversion process.

4.2.1.3 Conversion analysis

Concluding the conversion from **IETF SDF** to **YANG**, we can analyze the results obtained and the mappings between **IETF SDF** and OPC UA.

In the first conversion step, the generated **YANG** file is complete, that is, all the information from the **IETF SDF** document is correctly represented. In the second conversion step, the generated OPC UA Model Design XML file lacks some data, specifically the *sdfAction* groups.

After checking the generated files, we confirmed that they were lost during the **YANG** to OPC UA Design Model conversion. This seems to be a limitation of the OPC UA plugin used with *pyang*, which doesn't process **YANG** *action* statements, and these are generated when *sdfAction* groups are contained inside an *sdfObject* group; if the *sdfAction* groups are declared outside the *sdfObject*, they generate **YANG** *rpc* statements that are correctly processed. Since we want to analyze the resulting **IETF SDF** to the OPC UA information model mappings, we changed the **IETF SDF** document to have the *sdfAction* groups placed in the root, as presented in Appendix A.5. Following the procedure described throughout this section, we get the generated **YANG** file with the *rpc* statements in Appendix A.6, and the OPC UA Design Model file with the processed *sdfAction* groups in Appendix A.7. Finally, we generate the OPC UA NodeSet2 XML file in Appendix A.8.

Comparing the initial **IETF SDF** document and the final OPC UA NodeSet2 XML file, we obtained the expected mappings presented in Table 4.1, with the exception of the *sdfThing*, which was not used in this conversion.

In conclusion, the usage of existing tools to translate **IETF SDF** to the OPC UA information model presents a set of limitations. The four-minute overhead for the simple **IETF SDF** file used might be too much. It is expected to obtain even higher execution times for more complex files. For use cases where a conversion from **IETF SDF** to the OPC UA information model is needed, which is not the case of this thesis, the best approach would be to develop a custom solution to directly convert one to the other. The lack of support for converting *sdfAction* groups nested in *sdfObject* groups is another limitation found.

4.2.2 OPC UA information model to IETF SDF

Considering the mappings obtained from the **IETF SDF** to the OPC UA information model conversion in Subsection 4.2.1, we will propose a converter in the opposite direction. This converter aims to be integrated with frameworks or utilities that communicate with OPC UA servers. Therefore, it will be implemented using an OPC UA client. This means that instead of receiving an OPC UA NodeSet2 XML file as input, parsing it, and generating **IETF SDF** documents, we take advantage of the built-in parsing capacities and APIs of OPC UA clients to generate **IETF SDF**. The conversion process and the supported features are explained in 4.2.2.1.

4.2.2.1 Implementation details

In our solution, we use the OPC UA client that comes with *asyncua* [56], a Python module that provides asynchronous OPC UA server and client implementations. At this stage, the goal wasn't to support all OPC UA features. Next, we will explain the functioning of our converter along with the supported features.

The sequence diagram of Figure 4.2 shows a high-level overview of the conversion process. The OPC UA client (1) instantiates our converter, which is represented as the SDFGenerator (2). It takes the OPC UA *Objects Node* from the OPC UA server (3) and recursively generates an **IETF SDF** document for each child *Node* of *NodeClass Object*.

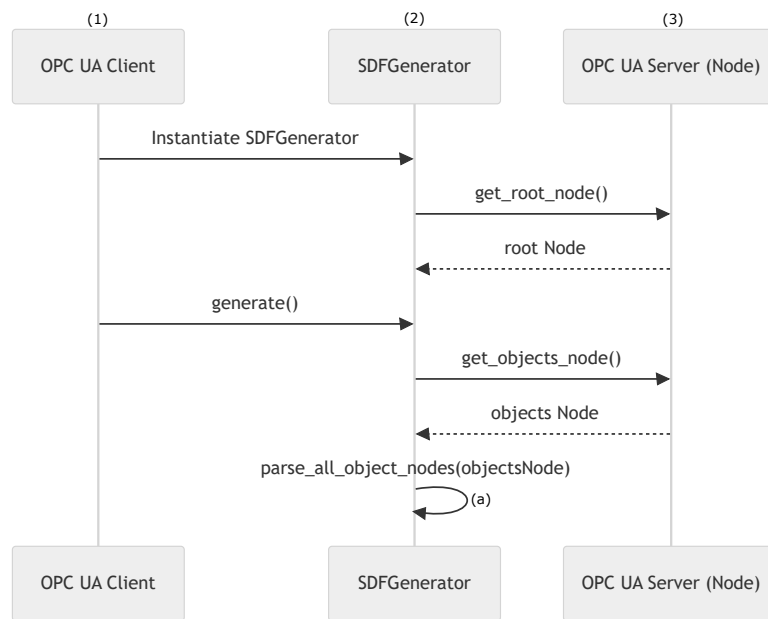


Figure 4.2: IETF SDF document generation overview

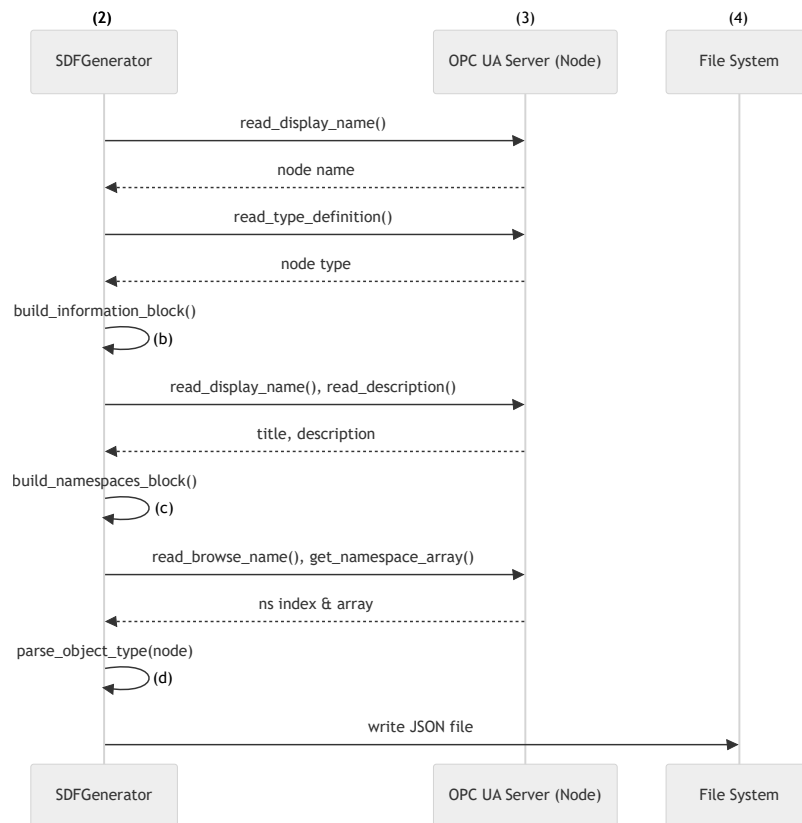
Sequence diagram in Figure 4.3 describes what the `parse_all_object_nodes()` (a) does. It starts by building the information block (b), which follows the implementation presented in Listing 4.1.

```

'info': {
    'title': <NodeDisplayName>,
    'description': <NodeDescription>,
    'version': strftime('%Y-%m-%d', gmtime()),
    'copyright': strftime('Copyright %Y Fraunhofer AICOS. All rights reserved.',
        gmtime()),
    'license': ''
}
  
```

Listing 4.1: IETF SDF information block construction

Next is the generation of the namespaces block (c). We took an approach where we only define two global variables: one holding the default namespace value, and the other holding the *types* namespace that we will explain later. Currently, all **IETF SDF** generated files only have these namespaces. Having more namespaces could be implemented in the future.

Figure 4.3: `parse_all_object_nodes()` implementation steps

Finally, we get to the definitions block, the most relevant one, which is built in the `parse_object_type()` function (d). Before going through its generation process, we will first explain how OPC UA *DataType Nodes* are handled, as well as indicate the ones supported by our solution. For now, these are the types in Table 4.3. However, additional data types can be added in the future using the *sdfData* group capabilities to model complex types.

Boolean	SByte	Byte	Int16	UInt16
Int32	UInt32	Int64	UInt64	Float
Double	String	DateTime	Guid	ByteString
NodeId	ExpandedNodeId	StatusCode	QualifiedName	LocalizedText

Table 4.3: OPC UA types supported by our converter

The **IETF SDF** definitions of these types were manually written. A file named `sdfdata-types.sdf.json` was created, and the type definitions were placed inside an *sdfData* group, with constraints applied to each data type to match the OPC UA type definitions better. This file can be found in Appendix A.9. You can see that the *types* default namespace of this **IETF SDF** documents is `https://example.com/types/`. This is the namespace that is always placed in the namespaces block of all **IETF SDF** documents.

Back to the definitions block, the `parse_object_type()` function takes the *ObjectType* of an *Object Node* and parses its *Property* and *Method Nodes*. This process is presented in the sequence diagram of Figure 4.4.

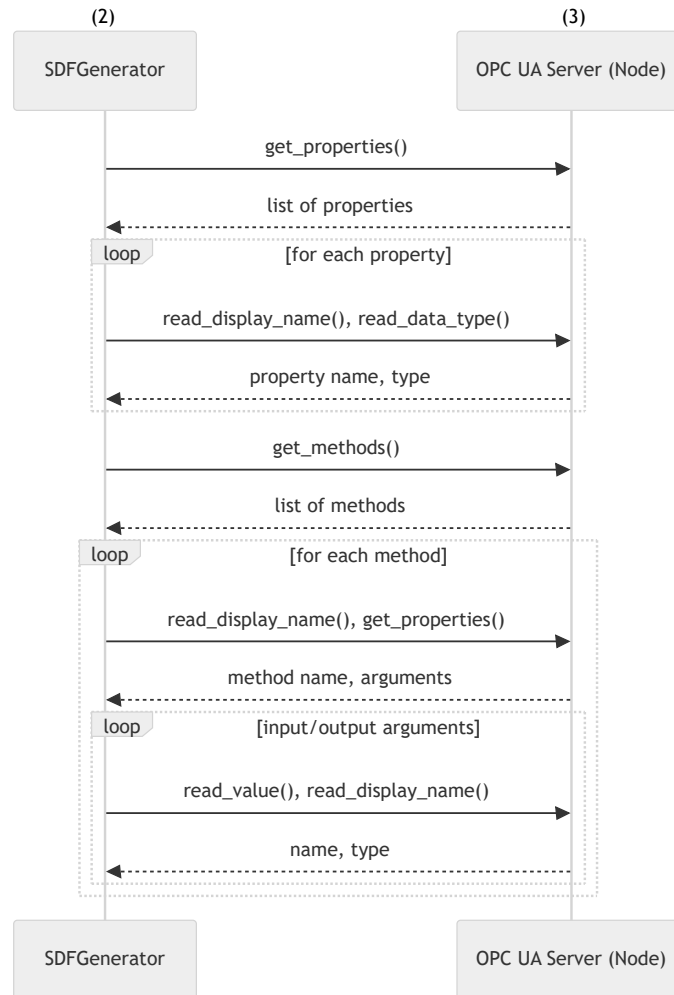


Figure 4.4: `parse_object_type()` implementation steps

For each *Property Node*, it returns the *Node's DisplayName* and, if the type is supported, it returns an *sdfRef* to the type definition in the `sdfdata-types.sdf.json` file; if the type is not supported, it returns null. Note that the *Property Nodes* of a *Node's parent Nodes* are recursively fetched. For each *Method Node*, a function returns the *Node's DisplayName* and, if the *Node* has input arguments, they are processed and placed inside the *sdfAction's sdfInputData* group; if the *Node* has output arguments, they are processed and placed inside the *sdfAction's sdfOutputData* group. Each entry in the *sdfInputData* and *sdfOutputData* groups has an *sdfRef* statement pointing to its type definition, if it belongs in the set of supported types.

In summary, our converter generates *sdfObject* groups from OPC UA *Object Nodes*, *sdfProperty* groups from OPC UA *Property Nodes*, and *sdfAction* groups from OPC UA *Method Nodes*.

It generates *sdfInputData* and *sdfOutputData* groups from *Method Nodes* input and output parameters. The supported OPC UA types are the ones in Table 4.3. Conversion results are presented in Chapter 5.

4.3 NextGenGW integration

Our converter proved that **IETF SDF** is capable of modeling the selected OPC UA features. Now, we will associate the NextGenGW-supported operations with the logical correspondent OPC UA interactions. NextGenGW defines six operations: READ, POST, PUT, DELETE, OBSERVE_START, and OBSERVE_STOP. The mappings that we propose are shown in Table 4.4 and, from these, the READ and POST operations will be evaluated for compatibility.

NextGenGW Operation	OPC UA interaction
READ	Read <i>Node</i> value
POST	Write to <i>Node</i> value
PUT	Create new <i>Node</i>
DELETE	Delete <i>Node</i>
OBSERVE_START	Create subscription
OBSERVE_STOP	Delete subscription

Table 4.4: NextGenGW - OPC UA mappings

Following NextGenGW's architecture, a possible integration would consist of placing an OPC UA client beside the IoTivity server that would communicate with outside devices, which would be OPC UA servers. Note that, in NextGenGW's architecture diagram, we see two servers, but no clients. This is because, in OPC UA, machines and devices are the OPC UA servers, so, on NextGenGW's side, we must have an OPC UA client. An external application would send a message that would be received by the **MQTT** client. Then, the *DownStream Manager* would forward it to the correct translator, which would take the data in **IETF SDF** format and trigger the appropriate action in the OPC UA client, which would be a read or a write operation. This integration is represented in Figure 4.5.

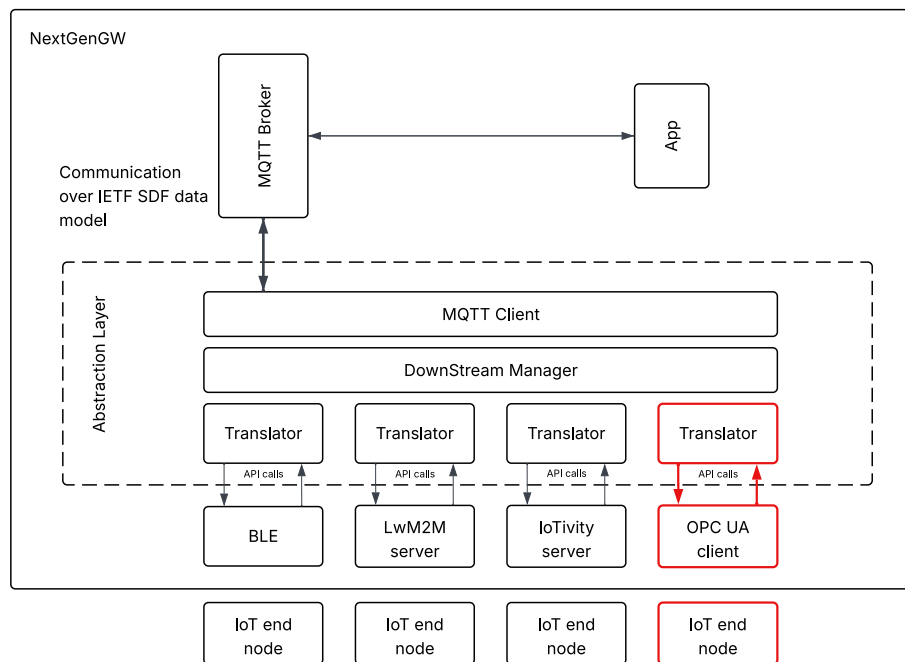


Figure 4.5: NextGenGW integration following its architecture

A more straightforward approach can be taken while still evaluating the theoretical mappings proposed. This approach removes the *Downstream Manager* from the NextGenGW architecture and focuses only on the OPC UA server. This removes the complexity of a regular NextGenGW integration, allowing for faster results. Due to time limitations, this was the approach implemented. Figure 4.6 presents this integration.

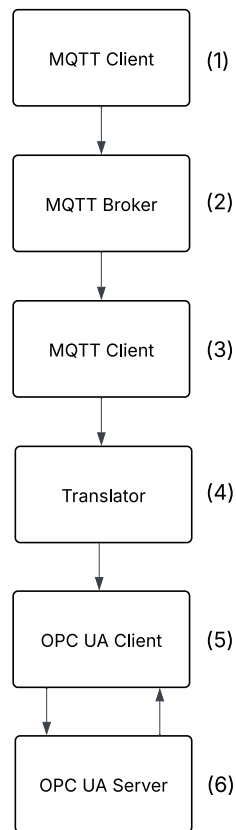


Figure 4.6: Implemented integration of OPC UA with NextGenGW

An **MQTT** client (3) receives a message sent by an **MQTT** publisher (1) via an **MQTT** broker (2). The message topic and payload are parsed, specifying the target object, property, or action, and the operation to be performed. This information is processed by the translator (4), and an OPC UA client (5) is used to perform the operation against the OPC UA server (6). Note that we are not taking the response from the OPC UA server and sending it back to the **MQTT** publisher (1), since we only wanted to evaluate if OPC UA operations are supported by NextGenGW.

Let's consider the following example. In the OPC UA server, we instantiate an *Object* of type *EnvironmentSensorType* named *EnvSensor* in namespace 1. The *EnvironmentSensorType* consists of two *Property Nodes* - Temperature and Humidity - and one *Method Node* - SetThresholds - with two input arguments - TemperatureThreshold and HumidityThreshold - and one output argument - Status. Therefore, to interact with this *Object*, the **MQTT** subscriber and the **MQTT** publisher must subscribe and publish to the **MQTT** topic named `1/EnvSensor`, respectively.

Now, imagine that we want to read the value of the OPC UA *Temperature* and *Humidity Property Node* values. The **MQTT** publisher must send a message to the topic `1/EnvSensor` with the payload in Listing 4.2.

```

{
  "operation": "READ",
  "data": {

```

```

        "sdfProperty": {
            "Temperature": "",
            "Humidity": ""
        }
    }
}

```

Listing 4.2: Message payload to read the values of *Temperature* and *Humidity*

When the message arrives, the **MQTT** client first parses the topic to assess the desired interaction level. Remember that, in NextGenGW, we can interact at the object, action, event, or property levels. In this implementation, we currently support object and property interaction levels. Given the `1/EnvSensor` topic, we are before an interaction at the object level. Then, the message payload is evaluated, determining the operation to be performed - we are only supporting READ and POST - and the operation's targets. In this example, the operation is READ, and the targets are the *Temperature* and *Humidity* *sdfProperty* entries.

Now, it's time to handle the operation. On the first operation since the **MQTT** client start, an in-memory representation of all OPC UA *Objects* at the connected OPC UA server is done, keeping it cached for the following operations. This is an implementation that uses three classes: *SDFObject*, *SDFProperty*, and *SDFAction*. All these classes save the *NodeId* and *DisplayName* of the corresponding OPC UA *Nodes*, as we will be needing them later.

The *SDFObject* class is presented in Listing 4.3. It contains two lists to save its *sdfProperty* and *sdfAction* groups, and functions to define the comparison criteria, which are used, for instance, when finding an element in a list, and printing format for a better debugging experience.

```

class SDFObject:

    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.sdfProperty: List[SDFProperty] = []
        self.sdfAction: List[SDFAction] = []

    def __repr__(self):
        props = "\n".join(
            [f"    Property - id: {p.id}, name: {p.name}, type: {p.type}, value: {p.value}" for p in self.sdfProperty]
        )
        actions = "\n".join(
            [f"    Action - id: {a.id}, name: {a.name}, inputs: {a.inputArguments}, outputs: {a.outputArguments}" for a in self.sdfAction]
        )
        return f"Object {self.name}:\n{props}\n{actions}"

    def __str__(self):

```

```
        return self.__repr__()

    def __eq__(self, value):
        return self.name == value.name

    def addSDFProperty(self, prop: SDFProperty):
        self.sdfProperty.append(prop)

    def addSDFAction(self, action: SDFAction):
        self.sdfAction.append(action)
```

Listing 4.3: SDFObject class

Then, in Listing 4.4, we have the *SDFProperty* class.

```
class SDFProperty:

    def __init__(self, id, name, type, value):
        self.id = id
        self.name = name
        self.type = type
        self.value = value

    def __eq__(self, value):
        return self.name == value.name
```

Listing 4.4: SDFProperty class

And, finally, the *SDFAction* class in Listing 4.5, which saves any input or output arguments that the OPC UA *Method* may have.

```
class SDFAction:

    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.inputArguments = []
        self.outputArguments = []

    def addInputArgument(self, arg):
        self.inputArguments.append(arg)

    def addOutputArgument(self, arg):
        self.outputArguments.append(arg)

    def __eq__(self, value):
        return self.name == value.name
```

Listing 4.5: SDFAction class

Since we obtain the *Object's* name in the **MQTT** topic, we can look it up in the generated representations, a list of *SDFObject* instances. That's why we overrode the `__eq__` function of the *SDFObject* class. If found, we check its *sdfProperty* list for the values that we want to read, which, in this case, are the *Humidity* and *Temperature*. If found, we are now able to read their values. For that, we use an OPC UA client that has a method to read a *Node's* value. However, we need the *NodeId*, and that's why we save it in every class instance. A message is printed with the read value.

The process for a POST operation is the same, with the difference that we write a value to the *Node*. An example payload is shown in Listing 4.6.

```
{
  "operation": "POST",
  "data": {
    "sdfProperty": {
      "Temperature": 10.0,
      "Humidity": 20.0
    }
  }
}
```

Listing 4.6: Message payload to write the values of *Temperature* and *Humidity*

Chapter 5

Results

In this chapter, we present the results of our OPC UA-IETF SDF converter together with the results of the association between NextGenGW-supported operations and OPC UA interactions. Taking the integration with NextGenGW described in Section 4.3, we will evaluate it for some parameters: execution time, memory usage, and CPU usage. Low execution times are important to ensure that little overhead is added with the use of NextGenGW; low memory and CPU usage are desired to allow NextGenGW to be deployed in machines with limited capabilities.

The OPC UA-IETF SDF conversion results are presented in Section 5.1. The test setup used to evaluate the translation between NextGenGW-supported operations and OPC UA interactions is described in Section 5.2, and Section 5.3 presents the analysis of the gathered results.

5.1 OPC UA-IETF SDF conversion results

To test our solution, we used the *opcua-modeler* utility from Free OPC UA [55], a Graphical User Interface (GUI) utility to design OPC UA *AddressSpaces*. Even though it is no longer maintained, we found it helpful to have a visual representation of OPC UA *Nodes* and *References*, and, since it comes with an OPC UA server - which is the one that comes with *asyncua* - we can create our own *Nodes* and connect our converter's OPC UA client to this OPC UA server to generate the IETF SDF documents.

To test all of our converter's features, we modeled the IETF SDF document in Appendix B.1 that represents an environment sensor with two *sdfProperty* groups to represent humidity and temperature values, and one *setThreshold sdfAction* to set temperature and humidity level thresholds. It receives these thresholds as input data and returns the request acknowledgment as output. The *opcua-modeler* representation is presented in Figure 5.1.

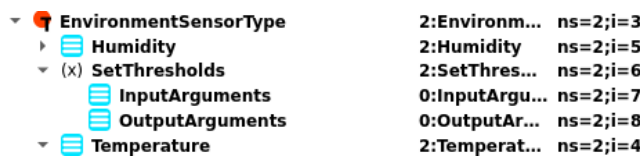


Figure 5.1: IETF SDF modeled environment sensor

After running our converter against this OPC UA server, we generated 75 **IETF SDF** documents. That is because the OPC UA server loads the OPC UA default node set, and the converter traverses all the *Object Nodes*, generating **IETF SDF** documents for them. In the future, we will implement the functionality of only generating **IETF SDF** documents for specified *Nodes*. Looking for the file named `sdfobject-EnvironmentSensor.sdf.json`, we observe that its content, in Appendix B.2, is as expected according to the mappings of Section 4.1. We have an *sdfObject* group with the two entries in the *sdfProperty* group: one named *Temperature* and the other named *Humidity*, both of type float. Then, we have the *sdfAction* group with the *SetThresholds* entry with its two input arguments - *tempThreshold* and *humidityThreshold* - in the *sdfInputData* group, and the output argument - *status* - in the *sdfOutputData* group.

5.2 OPC UA-NextGenGW test setup

The machine where the tests were run was equipped with an AMD Ryzen 7 7700X 8-Core CPU @4.50 base GHz with 16 threads, and 32GB of RAM. The tests ran in a Windows Subsystem for Linux (WSL) environment with Ubuntu 22.04.3 LTS installed, where 16GB of memory was available.

To automate the test runs, a simple Python script was developed to launch an OPC UA server and an **MQTT** client to get the READ and POST requests. The script receives the number of OPC UA *Property Nodes* to be created as an argument. After creating a test *ObjectType Node* and an *Object Node* of that type named *MyTestObject*, the requested number of *Property Nodes* is created following the `TestProperty<N>` naming convention, where *N* is the number of the *Property*. The value of each *Property Node* is also set to *N*. When creating these *Property Nodes*, the payload of the **MQTT** message to be sent is built. This will contain the operation to be performed and all the *Property Nodes* names to be read or written. Listing 5.1 shows an example of a payload for a READ operation.

```
{
  "operation": "READ",
  "data": {
    "sdfProperty": {
      "TestProperty0": "",
      "TestProperty1": "",
      "TestProperty2": "",
      "TestProperty3": "",
      "TestProperty4": "",
      "TestProperty5": "",
      "TestProperty6": "",
      (...)
      "TestPropertyN": ""
    }
  }
}
```

Listing 5.1: READ operation payload

The payload for a POST operation is similar, the only difference being the existence of a value to be written. Listing 5.2 provides an example of a payload for a POST operation.

```
{
  "operation": "POST",
  "data": {
    "sdfProperty": {
      "TestProperty0": 0,
      "TestProperty1": 1,
      "TestProperty2": 2,
      "TestProperty3": 3,
      "TestProperty4": 4,
      "TestProperty5": 5,
      "TestProperty6": 6,
      (...)
      "TestPropertyN": N
    }
  }
}
```

Listing 5.2: POST operation payload

Next, an **MQTT** client is instantiated and subscribes to the *1/MyTestObject/#* topic, where it will receive the READ and POST requests. It will connect to the OPC UA server and listen for messages in the subscribed topic.

Finally, we use the `publish.single()` function from the *paho.mqtt.publish* Python module to send a message with the generated payload to the existing topic. The **MQTT** client will process the message and execute the requested readings or writings.

A shell script was then created to call this Python script with different argument values. We tested the READ and POST operations for 10, 100, 1,000, and 10,000 *Property Nodes*. After each run, we get the execution time, the peak CPU usage, and the memory usage.

The execution time was measured with the `time()` function from the `time` Python module in the code, and the peak CPU usage and memory usage were measured using the GNU `time` utility (not the built-in shell command). The results are presented in Section 5.3.

5.3 OPC UA-NextGenGW result analysis

We will start by analyzing the execution time for the READ and POST operations for different numbers of operations performed. Figure 5.2 shows the obtained results, where the measurements

for both operations were almost the same and, as expected, the execution time increased with the number of operations.

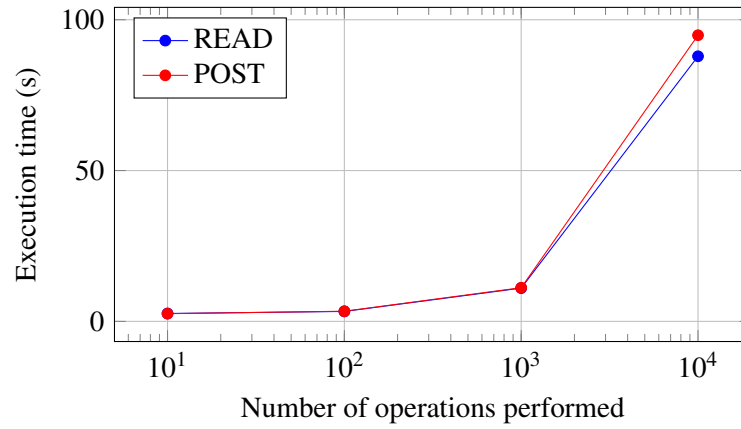


Figure 5.2: READ vs POST operation execution times (s)

The execution time ranges from 2.55 seconds for the writing of 10 *Property Nodes* to 94.87 seconds for the writing of 10,000 *Property Nodes*. Note that these values are cumulative for all the operations performed. Therefore, if we divide the execution time by the number of *Property Nodes*, we get the average execution time for a single operation. Figure 5.3 presents these results.

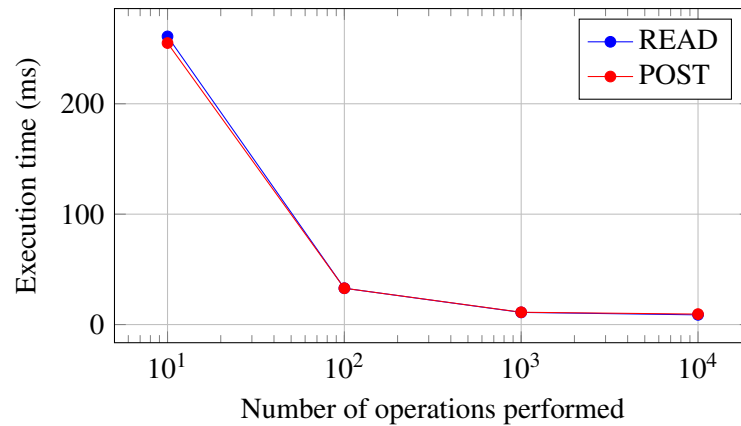


Figure 5.3: READ vs POST single operation execution times (ms)

We get interesting results here with the execution time per operation decreasing when more operations are performed. We believe that this is due to the overhead introduced by asynchronous operations when setting up their event loop. When more operations are performed, this overhead is diluted, decreasing the execution time per operation.

Figure 5.4 presents the memory usage (%) in both operations for the same different number of *Property Nodes*. In this case, the values overlap due to the implementation of both operations being alike, and memory usage increased with the number of operations performed, as expected. This is due to the internal representation of all OPC UA *Object Nodes* and their child *Nodes*. When creating more *Property Nodes*, these will take more memory. Still, the memory usage remained

relatively low, hitting a maximum at 1.96% for 10,000 operations performed, resulting in the creation of 10,000 *Property Nodes*.

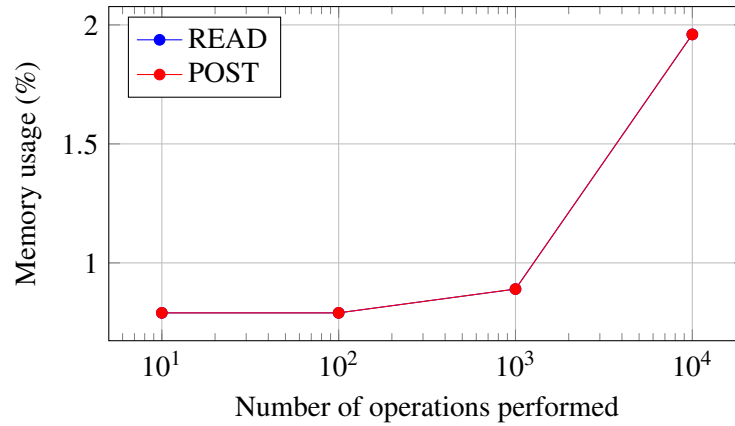


Figure 5.4: READ vs POST operation memory usage (%)

Despite the low memory usage, we have to consider that there were 16GB of memory available. Therefore, we present the absolute values of memory consumption in Figure 5.5 so they can be interpreted for machines with less memory, which is common in IIoT scenarios. Again, only one line is visible because it is overlapping the other.

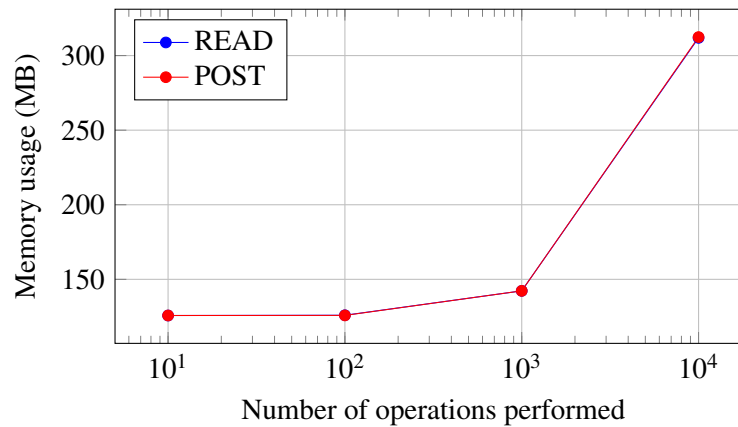


Figure 5.5: READ vs POST operation memory usage (absolute values in MB)

Finally, we will analyze the peak CPU usage (%) for the same scenarios presented until now. Figure 5.6 presents these results. From 10 to 1,000 operations performed, the measurements remained steady, hitting 71% for 10,000 READ operations. The high CPU usage presented is due to how this implementation was coded.

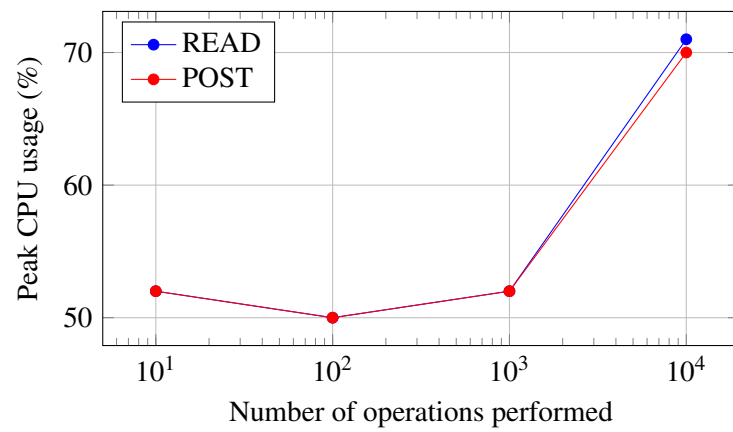


Figure 5.6: READ vs POST operation peak CPU usage (%)

In conclusion, this integration with NextGenGW uses little memory, peaking at 312.404 MB for 10,000 operations performed, but is CPU-intensive, peaking at 71% for 10,000 READ operations performed. The execution times per operation are acceptable, being in the millisecond order of magnitude. However, an effort to improve the CPU usage should be made.

Chapter 6

Conclusion

This chapter concludes this thesis by presenting its contributions in Section 6.1 and the future improvements for both our OPC UA-IETF SDF converter and OPC UA-NextGenGW solution in Section 6.2.

6.1 Contributions

The goal of this thesis was to assess whether IETF SDF could be used to solve the interoperability problem in industrial scenarios. To accomplish this goal, we stipulated that we would need to be able to represent the data model of an industrial protocol in IETF SDF. In Chapter 2, we started by going through the levels of interoperability and presenting a comprehensive study on IIoT communication protocols, describing their advantages, disadvantages, and open source implementations, answering the research question RQ1. From the results, we selected OPC UA for the IETF SDF compatibility study, given its mature information model and the recommendation to be used in the communication layer in the RAMI4.0.

Then, in Chapter 4, we searched for any existing conversion between the OPC UA information model and IETF SDF. We found one multi-step conversion that converts IETF SDF to YANG, followed by a conversion from YANG to the OPC UA information model. This conversion, however, lacks support for some IETF SDF functionalities, such as having *sdfAction* entries nested in *sdfObject* groups.

Since a conversion in the opposite direction was not found, we proposed our own converter. Even though it doesn't support all OPC UA features due to missing implementations, it is a stepping stone for a more complete solution. We achieved conversions of OPC UA *Object*, *Method*, and *Property Nodes* with support for a set of OPC UA data types. Even though this converter still needs improvements, we show that IETF SDF is capable of representing core features of the OPC UA information model, answering the research question RQ2. A continuation of this study would be needed to assess the compatibility of other OPC UA features.

Still in Chapter 4, the MQTT-IETF SDF bindings proposed by NextGenGW were also addressed for an OPC UA integration, answering the research question RQ3. This work revealed

that the READ and POST operations are supported in this integration with acceptable execution times and memory usage, but are highly CPU-intensive. However, the remaining OPC UA operations would need to be tested to evaluate if they are supported by NextGenGW.

6.2 Future work

The next steps for this work revolve around supporting more OPC UA features in our OPC UA-IETF SDF converter, making it more complete and robust. In Table 6.1, we present the features that we consider to be the most important ones, along with some implementation hints.

Feature	Implementation hint
Support for OPC UA <i>DataVariables</i>	OPC UA <i>DataVariables</i> may be represented in <i>sdfData</i> groups of type <i>Object</i> that would contain nested OPC UA <i>Variables</i> .
Support for OPC UA <i>Events</i>	OPC UA <i>Events</i> could be mapped to <i>sdfEvent</i> groups. These group definitions would be created from the <i>Event</i> 's <i>EventType</i> .
Support for <i>sdfRequired</i>	The <i>sdfRequired</i> may be implemented by checking the <i>References</i> of type <i>HasModellingRule</i> .
Support for IETF SDF <i>minItems</i> and <i>maxItems</i>	The IETF SDF <i>minItems</i> and <i>maxItems</i> qualities may be implemented by checking the <i>arrayDimensions</i> attribute of an OPC UA <i>Node</i> .
Automatic generation of simple OPC UA data types	OPC UA simple types were converted to IETF SDF manually. This conversion could be automatic by parsing the <i>DataType Nodes</i> .
Support for complex OPC UA data types	<i>sdfData</i> groups could be generated from OPC UA complex type definitions.

Table 6.1: OPC UA to IETF SDF next features

The other major step would be the support for all OPC UA operations that were not evaluated in this work. Further optimizations could be made in the developed integration to improve the CPU usage and reduce the memory consumption further. Different OPC UA clients could be used in the implementation, and their performance could be evaluated. For instance, an OPC UA client written in C++ may be more memory-efficient.

Finally, our integration with NextGenGW could be evaluated for multiple OPC UA server connections. The developed integration would be tested for reading or writing multiple variables in multiple OPC UA servers. This is likely to increase the memory usage, since, in our implementation, an OPC UA client would be spawned for every OPC UA server, resulting in a one-to-one connection.

References

- [1] Ali Alamery, Mohanad Abd Shehab, Omar Abdulkareem Mahmood, Yousif I Hammadi, Ahmed Aziz, and Mohammed Saleh Ali Muthanna. Internet of things protocols (iotp) literature review. In *Proceedings of the 7th International Conference on Future Networks and Distributed Systems, ICFNDS '23*, page 730–734, New York, NY, USA, 2024. Association for Computing Machinery.
- [2] G. Aloï, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, and C. Savaglio. Enabling iot interoperability through opportunistic smartphone-based mobile gateways. *Journal of Network and Computer Applications*, 81:74–84, 2017.
- [3] Fatima Zahra Amara, Meriem Djezzar, Mounir Hemam, Sanju Tiwari, and Mohamed Madani Hafidi. Unlocking the power of semantic interoperability in industry 4.0: A comprehensive overview. In Fernando Ortiz-Rodriguez, Boris Villazón-Terrazas, Sanju Tiwari, and Carlos Bobed, editors, *Knowledge Graphs and Semantic Web*, pages 82–96, Cham, 2023. Springer Nature Switzerland.
- [4] T. Anitha, S. Manimurugan, S. Sridhar, S. Mathupriya, and G. Charlyn Pushpa Latha. A review on communication protocols of industrial internet of things. In *2022 2nd International Conference on Computing and Information Technology (ICCIT)*, pages 418–423, 2022.
- [5] POSC Caesar Association. Draft of industrial data ontology formerly known as “iso 15926 part 14 ontology: DI profile for iso 15926-2”, 2025. Available at https://rds.posccaesar.org/WD_IDO.pdf, last accessed on March 2025.
- [6] BaSyx. Virtual automation bus, 2024. Available at https://wiki.basysx.org/en/latest/content/user_documentation/concepts%20and%20architecture/vab/index.html, last accessed on December 2024.
- [7] Olaf Bergmann. libcoap, 2025. Available at <https://github.com/obgm/libcoap>, last accessed on December 2024.
- [8] Martin Bjorklund. pyang, 2025. Available at <https://github.com/mbj4668/pyang>, last accessed on June 2025.
- [9] Patrick Boettcher. Json schema validator, 2025. Available at <https://github.com/pboettch/json-schema-validator>, last accessed on June 2025.
- [10] Marcella Cavalcanti, Hugo Costelha, and Carlos Neves. Industry 4.0 machine-to-machine communication protocols and architectures on the shop floor. In *Progress in Digital and Physical Manufacturing*, pages 222–234, Cham, 2023. Springer International Publishing.
- [11] CESNET. libyang, 2021. Available at <https://github.com/CESNET/libyang/tree/v1.0.225>, last accessed on June 2025.

- [12] Yingxiu Chen, Qimin Xu, Jinglong Zhang, Lei Xu, Lingzhi Li, and Cailian Chen. Tsn-compatible industrial wired/wireless multi-protocol conversion mechanism and module. In *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2022.
- [13] Nimbus Controls. Eipscanner, 2025. Available at <https://github.com/nimbuscontrols/EIPScanner>, last accessed on December 2024.
- [14] Bilgin Umut Deveci, Hilal Bas, Emre Ummak, Ozlem Albayrak, and Perin Unal. A thorough analysis and comparison of data communication protocols used in industry 4.0: the case of smart-cnc. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 199–206, 2022.
- [15] eModbus. emodbus, 2025. Available at <https://github.com/eModbus/eModbus>, last accessed on December 2024.
- [16] ETSI. Saref4inma: an extension of saref for the industry and manufacturing domain, 2025. Available at <https://saref.etsi.org/saref4inma/v1.1.2/>, last accessed on March 2025.
- [17] ETSI. The smart applications reference ontology (saref), 2025. Available at <https://onedm.org/sdflanguage/>, last accessed on March 2025.
- [18] Eclipse Foundation. Eclipse mosquito, 2025. Available at <https://github.com/eclipse-mosquitto/mosquitto>, last accessed on December 2024.
- [19] OPC Foundation. Opc ua in the reference architecture model rami 4.0, 2015. Available at <https://opccconnect.opcfoundation.org/2015/06/opc-ua-in-the-reference-architecture-model-rami-4-0/>, last accessed on June 2025.
- [20] OPC Foundation. Unified architecture, 2024. Available at <https://opcfoundation.org/about/opc-technologies/opc-ua/>, last accessed on November 2024.
- [21] OPC Foundation. Opc 10000-3: Ua part 3: Address space model, 2025. Available at <https://reference.opcfoundation.org/Core/Part3/v105/docs/>, last accessed on March 2025.
- [22] OPC Foundation. Opc 10000-5: Ua part 3: Information model, 2025. Available at <https://reference.opcfoundation.org/Core/Part5/v105/docs/>, last accessed on April 2025.
- [23] OPC Foundation. Ua-iiot-starterkit, 2025. Available at <https://github.com/OPCFoundation/UA-IIoT-StarterKit>, last accessed on December 2024.
- [24] OPC Foundation. Ua-modelcompiler, 2025. Available at <https://github.com/OPCFoundation/UA-ModelCompiler>, last accessed on June 2025.
- [25] OPC Foundation. Ua-.netstandard, 2025. Available at <https://github.com/OPCFoundation/UA-.NETStandard>, last accessed on December 2024.
- [26] Prakash Ganesan, S Gunasekaran, Ar Ashwin Balaji, S Jini Fathima, and S Suryaprakash. Comparative analysis on industrial iot communication protocols and its future directives. In *2021 International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, pages 1–6, 2021.

- [27] EIP Stack Group. Opener, 2025. Available at <https://github.com/EIPStackGroup/OpENer>, last accessed on December 2024.
- [28] Step Function I/O. Open dnp3, 2025. Available at <https://github.com/dnp3/opendnp3>, last accessed on December 2024.
- [29] Mahbubul Islam, Hossain Md. Mubashshir Jamil, Samiul Ahsan Pranto, Rupak Kumar Das, Al Amin, and Arshia Khan. Future industrial applications: Exploring lpwan-driven iot protocols. *Sensors*, 24(8), 2024.
- [30] Samer Jaloudi. Communication protocols of an industrial internet of things environment: A comparative study. *Future Internet*, 11(3), 2019.
- [31] Raghav Kadian. State of enterprise iot in 2025: Market recovery, ai integration, and upcoming regulations, 2025. Available at <https://iot-analytics.com/state-of-enterprise-iot/>, last accessed on June 2025.
- [32] Subash Kannothe, Frank Schnicke, and Pablo Oliveira Antonino. Enabling industry 4.0 communication protocol interoperability: An opc ua case study. In *7th Conference on the Engineering of Computer Based Systems*, ECBS 2021, New York, NY, USA, 2021. Association for Computing Machinery.
- [33] Jana Kiesewalter. Sdf-yang-converter, 2021. Available at <https://github.com/jkiesewalter/sdf-yang-converter/tree/main>, last accessed on June 2025.
- [34] libmodbus. libmodbus, 2025. Available at <https://github.com/stephane/libmodbus>, last accessed on December 2024.
- [35] Free OPC UA Library. Free opc ua, 2025. Available at <https://github.com/FreeOpcUa>, last accessed on December 2024.
- [36] K. L. Lueth. State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time, 2020. Available at <https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/>[hyphens], last accessed on October 2024.
- [37] K. L. Lueth. State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time, 2020. Available at <https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/iot-value-set-to-accelerate-through-2030-where-and-how-to-capture-it>, last accessed on October 2024.
- [38] Carsten Bormann Michael Koster and Ari Keränen. Semantic definition format (sdf) for data and interactions of things, 2025. Available at <https://onedm.org/sdflanguage/>, last accessed on March 2025.
- [39] Carsten Bormann Michael Koster and Ari Keränen. Semantic definition format (sdf) for data and interactions of things, 2025. Available at <https://datatracker.ietf.org/doc/draft-ietf-asdf-sdf/>, last accessed on March 2025.
- [40] One Data Model. One data model - related tools, 2024. Available at <https://github.com/one-data-model/tools/wiki/Related-tools>, last accessed on June 2025.

- [41] Yang Models. yang, 2025. Available at <https://github.com/YangModels/yang>, last accessed on June 2025.
- [42] Michael Nast, Hannes Raddatz, Benjamin Rother, Frank Golasowski, and Dirk Timmermann. A survey and comparison of publish/subscribe protocols for the industrial internet of things (iiot). In *Proceedings of the 12th International Conference on the Internet of Things, IoT '22*, page 193–200, New York, NY, USA, 2023. Association for Computing Machinery.
- [43] nlohmann. Json, 2025. Available at <https://github.com/nlohmann/json>, last accessed on June 2025.
- [44] open62541. open62541, 2025. Available at <https://github.com/open62541/open62541>, last accessed on December 2024.
- [45] DNP Org. Overview of dnp3 protocol, 2025. Available at <https://www.dnp.org/About/Overview-of-DNP3-Protocol>, last accessed on December 2024.
- [46] IEEE 802 Org. Ieee 802.1 tsn tg overview, 2018. Available at <https://www.ieee802.org/1/files/public/docs2018/detnet-tsn-farkas-tsn-overview-1118-v01.pdf>, last accessed on June 2025.
- [47] Modbus Org. Modbus application protocol specification, 2025. Available at https://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, last accessed on December 2024.
- [48] MQTT Org. Mqtt: The standard for iot messaging, 2024. Available at <https://mqtt.org>, last accessed on December 2024.
- [49] ODVA Org. Ethernet/ip quick start for vendors handbook, 2025. Available at https://www.odva.org/wp-content/uploads/2020/05/PUB00213R0_EtherNetIP_Developers_Guide.pdf, last accessed on December 2024.
- [50] Pedro Henrique Morgan Pereira, Gustavo Cainelli, Carlos Eduardo Pereira, Joao Paulo J. Da Costa, and Edison Pignaton De Freitas. An interoperability middleware for iiot. In *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*, pages 1–6, 2023.
- [51] Carlos Resende, Waldir Moreira, and Luís Almeida. Nextgengw - a software framework based on mqtt and semantic definition format. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 141–148, 2023.
- [52] Jan Romann. Bridging the gap to the web of things: On the conversion between wot data models and the semantic definition format, 2023. Available at <https://ceur-ws.org/Vol-3412/paper3.pdf>, last accessed on May 2025.
- [53] Devaraj Sambandan. yangopcua, 2024. Available at <https://github.com/DevarajSambandan/yangopcua>, last accessed on June 2025.
- [54] Erdem Tepe, Axel Busboom, and Michael Müller. A transformation framework for semantic interoperability in industry 4.0. In *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2022.
- [55] Free OPC UA. opcua-modeler, 2022. Available at <https://github.com/FreeOpcUa/opcua-modeler>, last accessed on June 2025.

- [56] Free OPC UA. asyncua, 2025. Available at <https://github.com/FreeOpcUa/opcua-asyncio>, last accessed on June 2025.
- [57] Wenguang Wang, Andreas Tolk, and Weiping Wang. The levels of conceptual interoperability model: applying systems engineering principles to m&s. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, San Diego, CA, USA, 2009. Society for Computer Simulation International.
- [58] C. Bormann Z. Shelby, K. Hartke. The constrained application protocol (coap), 2014. Available at <https://www.rfc-editor.org/rfc/pdf/rfc7252.txt.pdf>, last accessed on June 2025.
- [59] F. Zahran, B. Abu Zahra. It/ot convergence protocols: Dnp3, ethernet/ip, and modbus. *Proceedings - 2023 Congress in Computer Science, Computer Engineering, and Applied Computing, CSCE 2023*, pages 1743–1748, 2023.
- [60] F. Zahran, B. Abu Zahra. It/ot convergence protocols: Mqtt, opc, and rest. *Proceedings - 2023 Congress in Computer Science, Computer Engineering, and Applied Computing, CSCE 2023*, pages 1732–1737, 2023.

Appendix A

IETF SDF to OPC UA information model

```
1 {
2   "info": {
3     "title": "Example document for SDF (Semantic Definition Format)",
4     "version": "2019-04-24",
5     "copyright": "Copyright 2019 Example Corp. All rights reserved.",
6     "license": "https://example.com/license"
7   },
8   "namespace": {
9     "cap": "https://example.com/capability/cap"
10  },
11  "defaultNamespace": "cap",
12  "sdfObject": {
13    "Switch": {
14      "sdfProperty": {
15        "value": {
16          "description": "The state of the switch; false for off and true for on.",
17          "type": "boolean"
18        }
19      },
20      "sdfAction": {
21        "on": {
22          "description": "Turn the switch on; equivalent to setting value to true."
23        },
24        "off": {
25          "description": "Turn the switch off; equivalent to setting value to false"
26            .
27        },
28        "toggle": {
29          "description": "Toggle the switch; equivalent to setting value to its complement."
30        }
31      }
32    }
33  }
```

```
30     }
31   }
32 }
33 }
```

Listing A.1: IETF SDF example document for conversion

```
1 module Switch {
2   yang-version 1.1;
3   namespace "https://example.com/capability/cap";
4   prefix cap;
5
6   import sdf_extension {
7     prefix helper;
8   }
9
10  revision 2019-04-24;
11
12  container Switch {
13    helper:sdf-spec "sdfObject";
14
15    leaf value {
16      helper:sdf-spec "sdfProperty";
17      type boolean;
18      description
19        "The state of the switch; false for off and true for on.";
20    }
21
22    action off {
23      helper:sdf-spec "sdfAction";
24      description
25        "Turn the switch off; equivalent to setting value to false.";
26    }
27
28    action on {
29      helper:sdf-spec "sdfAction";
30      description
31        "Turn the switch on; equivalent to setting value to true.";
32    }
33
34    action toggle {
35      helper:sdf-spec "sdfAction";
36      description
37        "Toggle the switch; equivalent to setting value to its complement.";
38    }
39  }
40 }
```

Listing A.2: YANG file generated from IETF SDF document

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <opc:ModelDesign
3   xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:ua="http://opcfoundation.org/UA/"
6   xmlns:opc="http://opcfoundation.org/UA/ModelDesign.xsd"
7   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
8   xmlns="https://onedm.org/playground/#"
9   TargetNamespace="https://onedm.org/playground/#"
10  TargetVersion="2021-03-05">
11
12  <opc:Namespaces>
13    <opc:Namespace Name="playground" Prefix="pg" XmlPrefix="pg">https://onedm.
14      org/playground/#</opc:Namespace>
15    <opc:Namespace Name="sdf_extension" Prefix="helper" XmlPrefix="helper">http
16      ://opcfoundation.org/UA/default/</opc:Namespace>
17  </opc:Namespaces>
18
19  <opc:DataType SymbolicName="Frequencydata" BaseType="ua:Int64">
20  </opc:DataType>
21  <opc:DataType SymbolicName="Leveldata" BaseType="ua:Int64">
22  </opc:DataType>
23  <opc:DataType SymbolicName="Transitiontimedata" BaseType="ua:Double">
24  </opc:DataType>
25  <opc:ObjectType SymbolicName="LeveloptionsType" BaseType="ua:BaseObjectType">
26    <opc:Children>
27      <opc:Property SymbolicName="Leveloptions" DataType="ua:BaseObjectType"
28        AccessLevel="Read">
29        <opc>Description>implemented as a bitmap, modeled as an array with unique
30          items</opc>Description>
31      </opc:Property>
32    </opc:Children>
33  </opc:ObjectType>
34
35  <opc:ObjectType SymbolicName="MovestepmodeType" BaseType="ua:BaseObjectType">
36    <opc:Children>
37    </opc:Children>
38  </opc:ObjectType>
39
40  <opc:ObjectType SymbolicName="LevelType" BaseType="ua:BaseObjectType">
41    <opc:Children>
42      <opc:Property SymbolicName="Currentfrequency" DataType="ua:Unknown OPC UA
43        data type" AccessLevel="Read">
44      </opc:Property>

```

```

40      <opc:Property SymbolicName="Currentlevel" DataType="ua:Unknown OPC UA
41          data type" AccessLevel="Read">
42      </opc:Property>
43      <opc:Property SymbolicName="Defaultmoverate" DataType="ua:Int64"
44          AccessLevel="Read">
45      </opc:Property>
46      <opc:Property SymbolicName="Maxfrequency" DataType="ua:Unknown OPC UA
47          data type" AccessLevel="Read">
48      </opc:Property>
49      <opc:Property SymbolicName="Maxlevel" DataType="ua:Unknown OPC UA data
50          type" AccessLevel="Read">
51      </opc:Property>
52      <opc:Property SymbolicName="Minfrequency" DataType="ua:Unknown OPC UA
53          data type" AccessLevel="Read">
54      </opc:Property>
55      <opc:Property SymbolicName="Minlevel" DataType="ua:Unknown OPC UA data
56          type" AccessLevel="Read">
57      </opc:Property>
58      <opc:Property SymbolicName="Offtransitiontime" DataType="ua:Unknown OPC
59          UA data type" AccessLevel="Read">
60      </opc:Property>
61      <opc:Property SymbolicName="Onlevel" DataType="ua:Int64" AccessLevel="
62          Read">
63      <opc:DefaultValue> <uax:Int64>255</uax:Int64> </opc:DefaultValue>
64      </opc:Property>
65      <opc:Property SymbolicName="Onofftransitiontime" DataType="ua:Unknown OPC
66          UA data type" AccessLevel="Read">
67      <opc:DefaultValue> <uax:Unknown OPC UA data type>0</uax:Unknown OPC UA
68          data type> </opc:DefaultValue>
69      </opc:Property>
70      <opc:Property SymbolicName="Ontransitiontime" DataType="ua:Unknown OPC UA
71          data type" AccessLevel="Read">
72      </opc:Property>
73      <opc:Object Type SymbolicName="OptionsType" BaseType="ua:BaseObjectType">
74      <opc:Children>
75      <opc:Object SymbolicName="Leveloptions" TypeDefinition="
76          LeveloptionsType">
77      <opc:Object>
78      </opc:Children>
79      </opc:Object Type>
80
81      <opc:Object SymbolicName="Options" TypeDefinition="OptionsType">
82      <opc:Children>
83      <opc:References>
84      <opc:Reference IsInverse="true">
85      <opc:ReferenceType>SignalTo</opc:ReferenceType>
86      </opc:Reference>
87      </opc:References>
88      </opc:Children>

```

```

77         </opc:Object>
78         <opc:Property SymbolicName="Remainingtime" DataType="ua:Unknown OPC UA
           data type" AccessLevel="Read">
79             <opc:DefaultValue> <uax:Unknown OPC UA data type>0</uax:Unknown OPC UA
               data type> </opc:DefaultValue>
80         </opc:Property>
81     </opc:Children>
82 </opc:ObjectType>
83
84 <opc:Object SymbolicName="Level" TypeDefinition="LevelType">
85     <opc:Children>
86         <opc:References>
87             <opc:Reference IsInverse="true">
88                 <opc:ReferenceType>SignalTo</opc:ReferenceType>
89             </opc:Reference>
90         </opc:References>
91     </opc:Children>
92 </opc:Object>
93 </opc:ModelDesign>

```

Listing A.3: OPC UA Model Design file generated from YANG file

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <UANodeSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
   www.w3.org/2001/XMLSchema" LastModified="2025-06-25T22:31:18.1047439Z" xmlns="
   http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
3 <NamespaceUris>
4     <Uri>https://example.com/capability/cap</Uri>
5     <Uri>http://opcfoundation.org/UA/default</Uri>
6 </NamespaceUris>
7 <Models>
8     <Model ModelUri="https://example.com/capability/cap" Version="2019-04-24"
       PublicationDate="2025-06-25T22:31:18.1047439Z" ModelVersion="2019-04-24.0.0
       ">
9         <RequiredModel ModelUri="http://opcfoundation.org/UA/default/" />
10    </Model>
11 </Models>
12 <Aliases>
13     <Alias Alias="Boolean">i=1</Alias>
14     <Alias Alias="SByte">i=2</Alias>
15     <Alias Alias="Byte">i=3</Alias>
16     <Alias Alias="Int16">i=4</Alias>
17     <Alias Alias="UInt16">i=5</Alias>
18     <Alias Alias="Int32">i=6</Alias>
19     <Alias Alias="UInt32">i=7</Alias>
20     <Alias Alias="Int64">i=8</Alias>
21     <Alias Alias="UInt64">i=9</Alias>
22     <Alias Alias="Float">i=10</Alias>

```

```

23     <Alias Alias="Double">i=11</Alias>
24     <Alias Alias="DateTime">i=13</Alias>
25     <Alias Alias="String">i=12</Alias>
26     <Alias Alias="ByteString">i=15</Alias>
27     <Alias Alias="Guid">i=14</Alias>
28     <Alias Alias="XmlElement">i=16</Alias>
29     <Alias Alias="NodeId">i=17</Alias>
30     <Alias Alias="ExpandedNodeId">i=18</Alias>
31     <Alias Alias="QualifiedName">i=20</Alias>
32     <Alias Alias="LocalizedText">i=21</Alias>
33     <Alias Alias="StatusCode">i=19</Alias>
34     <Alias Alias="Structure">i=22</Alias>
35     <Alias Alias="Number">i=26</Alias>
36     <Alias Alias="Integer">i=27</Alias>
37     <Alias Alias="UInteger">i=28</Alias>
38     <Alias Alias="HasComponent">i=47</Alias>
39     <Alias Alias="HasProperty">i=46</Alias>
40     <Alias Alias="Organizes">i=35</Alias>
41     <Alias Alias="HasEventSource">i=36</Alias>
42     <Alias Alias="HasNotifier">i=48</Alias>
43     <Alias Alias="HasSubtype">i=45</Alias>
44     <Alias Alias="HasTypeDefinition">i=40</Alias>
45     <Alias Alias="HasModellingRule">i=37</Alias>
46     <Alias Alias="HasEncoding">i=38</Alias>
47     <Alias Alias="HasDescription">i=39</Alias>
48     <Alias Alias="HasCause">i=53</Alias>
49     <Alias Alias="ToState">i=52</Alias>
50     <Alias Alias="FromState">i=51</Alias>
51     <Alias Alias="HasEffect">i=54</Alias>
52     <Alias Alias="HasTrueSubState">i=9004</Alias>
53     <Alias Alias="HasFalseSubState">i=9005</Alias>
54     <Alias Alias="HasDictionaryEntry">i=17597</Alias>
55     <Alias Alias="HasCondition">i=9006</Alias>
56     <Alias Alias="HasGuard">i=15112</Alias>
57     <Alias Alias="HasAddIn">i=17604</Alias>
58     <Alias Alias="HasInterface">i=17603</Alias>
59     <Alias Alias="GeneratesEvent">i=41</Alias>
60     <Alias Alias="AlwaysGeneratesEvent">i=3065</Alias>
61     <Alias Alias="HasOrderedComponent">i=49</Alias>
62     <Alias Alias="HasAlarmSuppressionGroup">i=16361</Alias>
63     <Alias Alias="AlarmGroupMember">i=16362</Alias>
64     <Alias Alias="AlarmSuppressionGroupMember">i=32059</Alias>
65 </Aliases>
66 <UAObjectType NodeId="ns=1;i=1" BrowseName="1:SwitchType">
67     <DisplayName>SwitchType</DisplayName>
68     <References>
69         <Reference ReferenceType="HasProperty">ns=1;i=2</Reference>
70         <Reference ReferenceType="HasSubtype" IsForward="false">i=58</Reference>
71     </References>

```

```

72 </UAObjectType>
73 <UAVariable NodeId="ns=1;i=2" BrowseName="1:Value" ParentNodeId="ns=1;i=1"
    DataType="Boolean">
74   <DisplayName>Value</DisplayName>
75   <Description>The state of the switch; false for off and true for on.</
    Description>
76   <References>
77     <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
78     <Reference ReferenceType="HasModellingRule">i=78</Reference>
79     <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=1</Reference>
80   </References>
81 </UAVariable>
82 <UAObject NodeId="ns=1;i=3" BrowseName="1:Switch">
83   <DisplayName>Switch</DisplayName>
84   <References>
85     <Reference ReferenceType="HasProperty">ns=1;i=4</Reference>
86     <Reference ReferenceType="HasTypeDefinition">ns=1;i=1</Reference>
87   </References>
88 </UAObject>
89 <UAVariable NodeId="ns=1;i=4" BrowseName="1:Value" ParentNodeId="ns=1;i=3"
    DataType="Boolean">
90   <DisplayName>Value</DisplayName>
91   <Description>The state of the switch; false for off and true for on.</
    Description>
92   <References>
93     <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
94     <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=3</Reference>
95   </References>
96 </UAVariable>
97 </UANodeSet>

```

Listing A.4: OPC UA NodeSet2 XML file from OPC UA Model Design file

```

1  {
2  "info": {
3    "title": "Example document for SDF (Semantic Definition Format)",
4    "version": "2019-04-24",
5    "copyright": "Copyright 2019 Example Corp. All rights reserved.",
6    "license": "https://example.com/license"
7  },
8  "namespace": {
9    "cap": "https://example.com/capability/cap"
10 },
11 "defaultNamespace": "cap",
12 "sdfObject": {
13   "Switch": {
14     "sdfProperty": {
15       "value": {

```

```
16         "description": "The state of the switch; false for off and true for on.",
17         "type": "boolean"
18     }
19 }
20 }
21 },
22 "sdfAction": {
23     "on": {
24         "description": "Turn the switch on; equivalent to setting value to true."
25     },
26     "off": {
27         "description": "Turn the switch off; equivalent to setting value to false."
28     },
29     "toggle": {
30         "description": "Toggle the switch; equivalent to setting value to its
31             complement."
32     }
33 }
```

Listing A.5: IETF SDF changed example document for conversion

```
1 module Switch {
2     yang-version 1.1;
3     namespace "https://example.com/capability/cap";
4     prefix cap;
5
6     import sdf_extension {
7         prefix helper;
8     }
9
10    revision 2019-04-24;
11
12    container Switch {
13        helper:sdf-spec "sdfObject";
14        leaf value {
15            helper:sdf-spec "sdfProperty";
16            type boolean;
17            description
18                "The state of the switch; false for off and true for on.";
19        }
20    }
21
22    rpc off {
23        helper:sdf-spec "sdfAction";
24        description
25            "Turn the switch off; equivalent to setting value to false.";
26    }
```

```

27
28   rpc on {
29       helper:sdf-spec "sdfAction";
30       description
31           "Turn the switch on; equivalent to setting value to true.";
32   }
33
34   rpc toggle {
35       helper:sdf-spec "sdfAction";
36       description
37           "Toggle the switch; equivalent to setting value to its complement.";
38   }
39 }

```

Listing A.6: YANG file generated from updated IETF SDF document

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <opc:ModelDesign
3     xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:ua="http://opcfoundation.org/UA/"
6     xmlns:opc="http://opcfoundation.org/UA/ModelDesign.xsd"
7     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
8     xmlns="https://example.com/capability/cap"
9     TargetNamespace="https://example.com/capability/cap"
10    TargetVersion="2019-04-24">
11
12    <opc:Namespaces>
13        <opc:Namespace Name="capability" Prefix="cap" XmlPrefix="cap">https://
14            example.com/capability/cap</opc:Namespace>
15        <opc:Namespace Name="sdf_extension" Prefix="helper" XmlPrefix="helper">http
16            ://opcfoundation.org/UA/default/</opc:Namespace>
17    </opc:Namespaces>
18
19    <opc:Object Type SymbolicName="SwitchType" BaseType="ua:BaseObjectType">
20        <opc:Children>
21            <opc:Property SymbolicName="Value" DataType="ua:Boolean" AccessLevel="
22                Read">
23                <opc:Description>The state of the switch; false for off and true for on
24                .</opc:Description>
25            </opc:Property>
26        </opc:Children>
27    </opc:Object Type>
28
29    <opc:Object SymbolicName="Switch" TypeDefinition="SwitchType">
30        <opc:Children>
31            <opc:References>
32                <opc:Reference IsInverse="true">

```

```

29         <opc:ReferenceType>SignalTo</opc:ReferenceType>
30     </opc:Reference>
31 </opc:References>
32 </opc:Children>
33 </opc:Object>
34
35 <opc:Method SymbolicName="Off" ModellingRule="Mandatory">
36 </opc:Method>
37
38
39 <opc:Method SymbolicName="On" ModellingRule="Mandatory">
40 </opc:Method>
41
42
43 <opc:Method SymbolicName="Toggle" ModellingRule="Mandatory">
44 </opc:Method>
45
46 </opc:ModelDesign>

```

Listing A.7: OPC UA Model Design file generated from updated YANG file

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <UANodeSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
   www.w3.org/2001/XMLSchema" LastModified="2025-06-25T22:14:50.7486734Z" xmlns="
   http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
3   <NamespaceUris>
4     <Uri>https://example.com/capability/cap</Uri>
5     <Uri>http://opcfoundation.org/UA/default</Uri>
6   </NamespaceUris>
7   <Models>
8     <Model ModelUri="https://example.com/capability/cap" Version="2019-04-24"
       PublicationDate="2025-06-25T22:14:50.7486734Z" ModelVersion="2019-04-24.0.0"
       ">
9     <RequiredModel ModelUri="http://opcfoundation.org/UA/default/" />
10   </Model>
11 </Models>
12 <Aliases>
13   <Alias Alias="Boolean">i=1</Alias>
14   <Alias Alias="SByte">i=2</Alias>
15   <Alias Alias="Byte">i=3</Alias>
16   <Alias Alias="Int16">i=4</Alias>
17   <Alias Alias="UInt16">i=5</Alias>
18   <Alias Alias="Int32">i=6</Alias>
19   <Alias Alias="UInt32">i=7</Alias>
20   <Alias Alias="Int64">i=8</Alias>
21   <Alias Alias="UInt64">i=9</Alias>
22   <Alias Alias="Float">i=10</Alias>
23   <Alias Alias="Double">i=11</Alias>

```

```

24     <Alias Alias="DateTime">i=13</Alias>
25     <Alias Alias="String">i=12</Alias>
26     <Alias Alias="ByteString">i=15</Alias>
27     <Alias Alias="Guid">i=14</Alias>
28     <Alias Alias="XmlElement">i=16</Alias>
29     <Alias Alias="NodeId">i=17</Alias>
30     <Alias Alias="ExpandedNodeId">i=18</Alias>
31     <Alias Alias="QualifiedName">i=20</Alias>
32     <Alias Alias="LocalizedText">i=21</Alias>
33     <Alias Alias="StatusCode">i=19</Alias>
34     <Alias Alias="Structure">i=22</Alias>
35     <Alias Alias="Number">i=26</Alias>
36     <Alias Alias="Integer">i=27</Alias>
37     <Alias Alias="UInteger">i=28</Alias>
38     <Alias Alias="HasComponent">i=47</Alias>
39     <Alias Alias="HasProperty">i=46</Alias>
40     <Alias Alias="Organizes">i=35</Alias>
41     <Alias Alias="HasEventSource">i=36</Alias>
42     <Alias Alias="HasNotifier">i=48</Alias>
43     <Alias Alias="HasSubtype">i=45</Alias>
44     <Alias Alias="HasTypeDefinition">i=40</Alias>
45     <Alias Alias="HasModellingRule">i=37</Alias>
46     <Alias Alias="HasEncoding">i=38</Alias>
47     <Alias Alias="HasDescription">i=39</Alias>
48     <Alias Alias="HasCause">i=53</Alias>
49     <Alias Alias="ToState">i=52</Alias>
50     <Alias Alias="FromState">i=51</Alias>
51     <Alias Alias="HasEffect">i=54</Alias>
52     <Alias Alias="HasTrueSubState">i=9004</Alias>
53     <Alias Alias="HasFalseSubState">i=9005</Alias>
54     <Alias Alias="HasDictionaryEntry">i=17597</Alias>
55     <Alias Alias="HasCondition">i=9006</Alias>
56     <Alias Alias="HasGuard">i=15112</Alias>
57     <Alias Alias="HasAddIn">i=17604</Alias>
58     <Alias Alias="HasInterface">i=17603</Alias>
59     <Alias Alias="GeneratesEvent">i=41</Alias>
60     <Alias Alias="AlwaysGeneratesEvent">i=3065</Alias>
61     <Alias Alias="HasOrderedComponent">i=49</Alias>
62     <Alias Alias="HasAlarmSuppressionGroup">i=16361</Alias>
63     <Alias Alias="AlarmGroupMember">i=16362</Alias>
64     <Alias Alias="AlarmSuppressionGroupMember">i=32059</Alias>
65 </Aliases>
66 <UAObjectType NodeId="ns=1;i=1" BrowseName="1:SwitchType">
67     <DisplayName>SwitchType</DisplayName>
68     <References>
69         <Reference ReferenceType="HasProperty">ns=1;i=2</Reference>
70         <Reference ReferenceType="HasSubtype" IsForward="false">i=58</Reference>
71     </References>
72 </UAObjectType>

```

```

73   <UAVariable NodeId="ns=1;i=2" BrowseName="1:Value" ParentNodeId="ns=1;i=1"
      DataType="Boolean">
74     <DisplayName>Value</DisplayName>
75     <Description>The state of the switch; false for off and true for on.</
      Description>
76     <References>
77       <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
78       <Reference ReferenceType="HasModellingRule">i=78</Reference>
79       <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=1</Reference>
80     </References>
81   </UAVariable>
82   <UAObject NodeId="ns=1;i=3" BrowseName="1:Switch">
83     <DisplayName>Switch</DisplayName>
84     <References>
85       <Reference ReferenceType="HasProperty">ns=1;i=4</Reference>
86       <Reference ReferenceType="HasTypeDefinition">ns=1;i=1</Reference>
87     </References>
88   </UAObject>
89   <UAVariable NodeId="ns=1;i=4" BrowseName="1:Value" ParentNodeId="ns=1;i=3"
      DataType="Boolean">
90     <DisplayName>Value</DisplayName>
91     <Description>The state of the switch; false for off and true for on.</
      Description>
92     <References>
93       <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
94       <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=3</Reference>
95     </References>
96   </UAVariable>
97   <UAMethod NodeId="ns=1;i=5" BrowseName="1:Off">
98     <DisplayName>Off</DisplayName>
99     <References />
100  </UAMethod>
101  <UAMethod NodeId="ns=1;i=6" BrowseName="1:On">
102    <DisplayName>On</DisplayName>
103    <References />
104  </UAMethod>
105  <UAMethod NodeId="ns=1;i=7" BrowseName="1:Toggle">
106    <DisplayName>Toggle</DisplayName>
107    <References />
108  </UAMethod>
109 </UANodeSet>

```

Listing A.8: OPC UA NodeSet2 XML file from updated OPC UA Model Design file

```

1 {
2   "info": {
3     "title": "OPC UA types",
4     "version": "0.0.1",

```

```
5     "copyright": "",
6     "license": ""
7 },
8 "namespace": {
9     "types": "https://example.com/types/"
10 },
11 "defaultNamespace": "types",
12 "sdfData": {
13     "Boolean": {
14         "type": "boolean",
15         "nullable": false
16     },
17     "SByte": {
18         "type": "integer",
19         "nullable": false,
20         "minimum": -128,
21         "maximum": 127
22     },
23     "Byte": {
24         "type": "integer",
25         "nullable": false,
26         "minimum": 0,
27         "maximum": 255
28     },
29     "Int16": {
30         "type": "integer",
31         "nullable": false,
32         "minimum": -32768,
33         "maximum": 32767
34     },
35     "UInt16": {
36         "type": "integer",
37         "nullable": false,
38         "minimum": 0,
39         "maximum": 65535
40     },
41     "Int32": {
42         "type": "integer",
43         "nullable": false,
44         "minimum": -2147483648,
45         "maximum": 2147483647
46     },
47     "UInt32": {
48         "type": "integer",
49         "nullable": false,
50         "minimum": 0,
51         "maximum": 4294967295
52     },
53     "Int64": {
```

```

54         "type": "integer",
55         "nullable": false,
56         "minimum": -9223372036854775808,
57         "maximum": 9223372036854775807
58     },
59     "UInt64": {
60         "type": "integer",
61         "nullable": false,
62         "minimum": 0,
63         "maximum": 18446744073709551615
64     },
65     "Float": {
66         "type": "number",
67         "nullable": false
68     },
69     "Double": {
70         "type": "number",
71         "nullable": false
72     },
73     "String": {
74         "type": "string",
75         "nullable": false
76     },
77     "DateTime": {
78         "type": "string",
79         "sdfType": "unix-time",
80         "nullable": false
81     },
82     "Guid": {
83         "type": "string",
84         "nullable": false
85     },
86     "ByteString": {
87         "type": "string",
88         "sdfType": "byte-string",
89         "nullable": false
90     },
91     "NodeId": {
92         "type": "object",
93         "nullable": false,
94         "properties": {
95             "nsIndex": {
96                 "type": "integer"
97             },
98             "idType": {
99                 "sdfChoice": {
100                     "Numeric": {
101                         "const": 0
102                     },

```

```
103         "String": {
104             "const": 1
105         },
106         "Guid": {
107             "const": 2
108         },
109         "Opaque": {
110             "const": 3
111         }
112     },
113     "identifier": {
114         "type": "string"
115     }
116 }
117 },
118 "ExpandedNodeId": {
119     "type": "string",
120     "nullable": false
121 },
122 "StatusCode": {
123     "type": "integer",
124     "nullable": false
125 },
126 "QualifiedName": {
127     "type": "string",
128     "nullable": false
129 },
130 "LocalizedText": {
131     "type": "object",
132     "nullable": false,
133     "properties": {
134         "locale": {
135             "type": "string"
136         },
137         "text": {
138             "type": "string"
139         }
140     }
141 }
142 }
143 }
144 }
```

Listing A.9: OPC UA types supported by our converter

Appendix B

OPC UA information model to IETF SDF

```
1 {
2   "sdfObject": {
3     "environmentSensor": {
4       "description": "Sensor module for monitoring environmental conditions and
5         setting thresholds.",
6       "sdfProperty": {
7         "temperature": {
8           "description": "Current ambient temperature.",
9           "type": "number"
10        },
11        "humidity": {
12          "description": "Current ambient humidity level.",
13          "type": "number"
14        }
15      },
16      "sdfAction": {
17        "setThresholds": {
18          "description": "Set temperature and humidity alert thresholds.",
19          "sdfInputData": {
20            "tempThreshold": {
21              "type": "number"
22            },
23            "humidityThreshold": {
24              "type": "number"
25            }
26          },
27          "sdfOutputData": {
28            "status": {
29              "type": "string",
30              "description": "Acknowledgment of threshold setting."
31            }
32          }
33        }
34      }
35    }
36  }
```

```
31         }  
32     }  
33 }  
34 }  
35 }  
36 }
```

Listing B.1: IETF SDF modeled environment sensor

```
1 {
2   "info": {
3     "title": "EnvironmentSensor",
4     "description": "EnvironmentSensorType",
5     "version": "2025-06-26",
6     "copyright": "Copyright 2025 Fraunhofer AICOS. All rights reserved.",
7     "license": ""
8   },
9   "namespace": {
10     "defaultNs": "https://example.com/capability/cap",
11     "types": "https://example.com/types/"
12   },
13   "defaultNamespace": "defaultNs",
14   "sdfObject": {
15     "EnvironmentSensor": {
16       "sdfProperty": {
17         "Temperature": {
18           "sdfRef": "types:/sdfData/Float"
19         },
20         "Humidity": {
21           "sdfRef": "types:/sdfData/Float"
22         }
23       },
24       "sdfAction": {
25         "SetThresholds": {
26           "sdfInputData": {
27             "tempThreshold": {
28               "sdfRef": "types:/sdfData/Float"
29             },
30             "humidityThreshold": {
31               "sdfRef": "types:/sdfData/Float"
32             }
33           },
34           "sdfOutputData": {
35             "status": {
36               "sdfRef": "types:/sdfData/String"
37             }
38           }
39         }
40       }
41     }
42   }
43 }
```

Listing B.2: OPC UA information model to IETF SDF resulting file