Efficient web access to Open Platform Communications Unified Architecture semantics

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

Vorgelegt von

Rainer Schiekofer

aus Mainburg

Hauptberichter: **Univ.-Prof. Dr.-Ing. Dr. h. c. Michael Weyrich**
Mitberichter: **Univ.-Prof. Dr.-Ing. Dr. h. c. mult. Alexander Verl**

Tag der mündlichen Prüfung: **26.07.2023**

Institut für Automatisierungstechnik und Softwaresysteme
der Universität Stuttgart

2023

CONTENTS

GLOSSARY

API Application Programming Interface.

HATEOAS Hypermedia As The Engine Of Application State.

HMI Human-Machine Interface.

HTTP Hypertext Transfer Protocol.

IIoT Industrial-Internet-of-Things.

IoT Internet-of-Things.

JSON JavaScript Object Notation.

JSON-LD JavaScript Object Notation for Linked Data.

OPC UA Open Platform Communications Unified Architecture.

OWL Web Ontology Language.

RDFS Resource Description Framework Schema.

REST Representational State Transfer.

SHACL Shapes Constraint Language.

SPARQL SPARQL Protocol and RDF Query Language.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

XML Extensible Markup Language.

ABSTRACT

Ten years ago, industry and academia have set an ambitious vision to bring down the lot numbers of mass production to size one. This would enable highly configurable and individualized products. The vision has eventually become known as the fourth industrial revolution. For a long time, mass production of highly customized products was associated with high costs and resource inefficiency. A solution for these issues would open up entirely new possibilities. So far, setting up the production lines for mass production has been a very expensive and time-consuming task. As a result, only bigger companies could afford to bring up new products to the market through mass production. In contrast, small companies and highly innovative startups can get easily pushed out of the market by competitors who are able to leverage mass production to produce a similar product cheaper.

To realize the vision of the fourth industrial revolution, four design principles have been identified: interconnection, information transparency, decentralized decisions, and technical assistance. Within this thesis, the technology stack from the World Wide Web is investigated as a possible technology building block for the interconnection and information transparency challenge of future Industry 4.0 applications. To ease the migration path of the industry domain this thesis combines the web technology stack with the most promising Industry 4.0 technology of the industry domain, which is OPC UA. For the integration of OPC UA into the World Wide Web ecosystem, three main challenges have to be solved: (1) Providing access to OPC UA data through web standards; (2) Transformation of OPC UA semantics to Semantic Web standards; (3) Provide an efficient interface to access the data.

First, this thesis provides a mapping from OPC UA to REST, including a HTTP mapping, a RESTful resource representation, a concept for stateless service calls (including the contribution to the OPC UA standardization), as well as a full mapping of all OPC UA service sets. Second, a full mapping from OPC UA semantics to Semantic Web standards is provided. This thesis defines a concept to automatically transform the semantics of any OPC UA data model into an OWL ontology also including type hierarchies as well as modeling constraints. Third, this thesis presents how SPARQL can be used to query OPC UA graphs natively and also will present rules to transform standardized OPC UA queries into SPARQL queries. Furthermore, some issues of the standardized OPC UA query language are highlighted also including solutions to bypass them through native SPARQL queries.

KURZFASSUNG

Vor zehn Jahren haben sich Industrie und Wissenschaft die ehrgeizige Vision gesetzt, die Losgrößen der Massenproduktion auf die Größe eins zu senken. Dies würde hochgradig konfigurierbare und individualisierte Produkte ermöglichen. Diese Vision ist schließlich als vierte industrielle Revolution bekannt geworden. Lange Zeit war die Massenproduktion hoch individualisierter Produkte mit hohen Kosten und Ressourcenineffizienz verbunden. Eine Lösung für diese Probleme würde völlig neue Möglichkeiten eröffnen. Bislang war der Aufbau von Produktionslinien für die Massenproduktion eine sehr teure und zeitaufwändige Aufgabe. Folglich konnten es sich nur größere Unternehmen leisten, neue Produkte durch Massenproduktion auf den Markt zu bringen. Kleine Unternehmen und hoch innovative Start-ups können dagegen leicht von Konkurrenten mit Hilfe von Massenproduktion aus dem Markt gedrängt werden, indem ein ähnliches Produkt billiger hergestellt werden kann.

Zur Verwirklichung der Vision der vierten industriellen Revolution wurden vier Gestaltungsprinzipien identifiziert: Vernetzung, Informationstransparenz, dezentralisierte Entscheidungen und technische Unterstützung. Innerhalb dieser Arbeit wird die Technologie-Plattform rund um das World-Wide-Web, als möglicher Baustein um die Gestaltungsprinzipien Vernetzung und Informationstransparenz zu adressieren, genauer betrachtet. Zur Erleichterung des Migrationspfads der Industriedomäne wird hierbei die vielversprechendste Industrie 4.0 Technologie der Industriedomäne, OPC-UA, mit der Technologie des World-Wide-Web kombiniert.

Für die Integration von OPC-UA in das World-Wide-Web Ökosystem müssen drei Herausforderungen gelöst werden: (1) Bereitstellung des Zugriffs auf OPC-UA Daten durch Webstandards; (2) Transformation der OPC-UA Semantik in Semantic Web Standards; (3) Einen effizienten Zugriff auf die Daten zur Verfügung stellen.

Die erste Herausforderung wird in dieser Arbeit durch eine vollständige Abbildung von OPC-UA auf das Architekturkonzept REST gelöst. Dies beinhaltet eine Abbildung auf HTTP, die Einführung von RESTful Ressourcen-Repräsentationen, ein Konzept um zustandslose Service-Aufrufe auszuführen (inklusive den zugehörigen OPC-UA Standardisierungsbeiträgen) sowie eine vollständige Abbildung aller OPC-UA Service-Sets. Die zweite Herausforderung wird in dieser Arbeit durch eine vollständige Abbildung der OPC-UA Semantik auf Semantik Web Standards adressiert. Im Rahmen der Arbeit wird hierbei ein Konzept zur automatischen Transformation der Semantik von beliebigen OPC-UA Datenmodellen zu OWL Ontologien, inklusive Typ-Hierarchien und Modellierungsvorschriften, definiert und evaluiert. Die dritte Herausforderung wird durch ein Konzept gelöst, das es erlaubt OPC-UA Graphen mithilfe der SPARQL Technologien abzufragen. Zusätzlich werden Abbildungsvorschriften vorgestellt wie standardisierte OPC-UA abfragen auf SPARQL abgebildet werden können. Des weiteren werden einige Schwachstellen der standardisierten OPC-UA Abfragesprache vorgestellt und aufgezeigt wie diese Schwachstellen durch die direkte Verwendung von SPARQL behoben werden können.

1 INTRODUCTION

The term "Industrie 4.0" is well-known in Germany for nearly ten years now. The idea behind this term is the introduction of the fourth industrial revolution, where the first industrial revolution describes the transition from hand production to steam and water power-based production, the second industrial revolution is entered through inventions like production lines and the usage of electricity for manufacturing, and the third industrial revolution is founded on digitalization and microelectronics like programmable logic controllers. In contrast to the third industrial revolution, the fourth industrial revolution aims to provide mass production for highly configurable and individualized products based on four design principles: interconnection, information transparency, decentralized decisions, and technical assistance [67]. During the last years various academic papers, technical articles, standardization bodies, and conferences focused on this topic [151, 152, 142, 140, 133, 22, 107, 21, 34, 96, 101]. Furthermore, the terms Internet of Things (IoT) and Industrial Internet of Things (IIoT) can be considered closely related to Industry 4.0 and also are discussed heavily in academia and industry [165, 166, 45, 131, 104, 170, 163]. Finally, the goal of all these activities is to make use cases like dynamic reconfiguration of automation devices [70, 134, 167, 94], plug and produce [118, 122, 42], and analytics [89, 16, 106, 84, 64] feasible.

While the industry domain prefers well-understood and stable technology to provide high stability and reliability to their customers, the customers of the consumer goods domain are often very happy to be one of the first users of a new technology that might still contain some flaws. Based on this observation, the consumer goods domain could be considered as a good inspiration for possible future directions of the slower-moving industry domain. For challenges like interconnection, this domain heavily uses web technology like representational state transfer (REST), which can be considered the technological enabler for the World Wide Web. Furthermore, the web technology stack also provides concepts to solve the information transparency challenges through Semantic Web technology. Last but not least, web search engines like Google or Bing demonstrate how web information can be accessed in a very efficient way. In summary, the web technology stack provides solutions for some of the most important Industry 4.0 challenges and

therefore might be an interesting migration path for industry-specific technology on the way forward to the fourth industrial revolution.

The following Section 1.1 gives further insights into the challenges. Section 1.2 presents the goal of this thesis, followed by Section 1.3, which provides the contributions of this thesis to standardization and academia. Finally, Section 1.4 gives an overview of the further structure of this thesis.

1.1 Challenges for efficient web access to OPC UA semantics

At the moment our society stands on the edge of a new era. As the connectivity and processing power of industrial embedded devices increase more and more, a lot of new applications become feasible. This phenomenon has different names in different domains, for example, SmartGrid, SmartHome, Industrial-Internet-of-Things (IIoT), and of course the nowadays term for the automation domain, Industry 4.0. It is possible that these new properties of industrial embedded devices will change our lives in a similar way as the introduction of smartphones. The success of these new applications will depend on the interoperability of the transport layer, to enable different devices to communicate with each other, the interoperability of the semantic layer, to enable different devices to understand the meaning of the communication, and to provide an efficient way to access this information. These challenges can be considered enablers for typical Industry 4.0 scenarios.

(C1) Interoperability on the transport layer

One of the challenges in addressing interoperability on the transport layer is the huge number of different protocols. Nearly every domain has developed its own protocol to solve similar problems. In the end, the question arises if really a dedicated protocol for each domain is necessary, or if a single protocol is sufficient to address most of the common use cases. With that in mind, the starting point of this thesis is to identify the most promising Internet-of-Things protocol of the automation domain, which seems to be OPC Unified Architecture [127, 95]. OPC UA [77] does not only aim to solve the interoperability on the transport layer, instead, also the interoperability on the semantic layer shall be addressed by OPC UA, through the introduction of so-called *Companion Specifications*. However, to reach the level of a real Industrial-Internet-of-Things protocol, OPC UA must be able to reach out to other domains too.

A communication technology, which everybody already knows and also is present in nearly every domain, is the REST architecture [46]. REST is derived from the classic web, which is already connected to each domain in one way or another. The basic idea of this work is to extend OPC UA with REST capabilities to finally reach the status of an Internet-of-Things protocol and provide a solution for the interoperability of the transport layer challenge.

(C2) Interoperability on the semantic layer

In the area of factory automation OPC UA is the new standard that is promised to lift field device communication from low-level signal exchange schemes onto a semantic level, contributing to the realization of flexible manufacturing scenarios within the Industry 4.0 vision. OPC UA is a machine-to-machine communication protocol for industrial automation developed by the OPC Foundation. However, despite all the improvements that OPC UA brings over conventional device communication, it still exhibits certain problems when it comes to capturing the semantics of m2m communication structures: much of the semantics of the OPC UA basic constructs are defined in specification documents in an implicit way, only accessible to the human implementor. Moreover, the relatively new OPC UA specifications also lack implementation in available tools, which now just start to emerge.

On the other hand, Semantic Web [24] technology is state-of-the-art for representing and processing explicit semantics for data models in information systems in general, and specifically for the web. The standardized ontology languages RDF(S) and OWL provide a representation framework for formulating semantically rich knowledge graphs. Being supported by an active research community for some years now, it also offers an established set of tools that support these standards. Hence, it appears to be natural to investigate the use of the already matured Semantic Web technology stack for the relatively new OPC UA standard for capturing semantics more formally.

(C3) Efficient querying of information

If the research challenges C1 and C2 are addressed properly, it can be assumed that sooner or later the automation domain will be faced with huge standardized OPC UA information models with detailed descriptions of the underlying physical devices. This introduces big opportunities for a lot of use cases like analytics and human-machine interfaces (HMI), which can be programmed against standardized information models, enabling the deployment on each machine independent of the manufacturer without additional engineering effort. However, one important part to use such information models is still missing. Without some kind of query functionality it will soon be impossible to access the data points in an efficient way on the aggregating layers like edge and cloud and bind them to apps (e.g., a predictive maintenance app for an engine which of course needs some field values, like temperature/power/...). It is worth noting that OPC UA offers a query language for searching OPC UA information models, but up to now there is no publicly available implementation, as far as I know. Of course, it is not a practical solution to search the graph node by node for each application (on the cloud level ten-thousands of OPC UA *Nodes* have to be searched by hundreds of apps in parallel). Another problem is that the OPC UA-specific query language is so complex that some industry researchers even introduced an internal domain-specific language for constructing OPC UA *Queries* [59]. Rather than inventing a new query language from scratch or directly implementing OPC UA *Query*, an already existing query language should be used. Based on C2 and the proposed solution to use Semantic Web technology, SPARQL seems to

be a promising candidate for an efficient OPC UA access API. Furthermore, SPARQL also offers an HTTP API which allows to easily integrate it into the proposed C1 solution of this thesis.

1.2 Goal of this work

The goal of this thesis is a good integration into the selected technology ecosystems. For the research Challenge 1 (C1) this means that the mapping also takes the main client for RESTful applications into account (a web browser) and, therefore, also focuses on the support of the REST paradigm hypermedia as the engine of application state (HATEOAS) as it is defined by Fielding [46] and HTTP as the underlying communication technology. For the REST paradigm HATEOAS, this implies, that this thesis provides a self-descriptive message format with embedded hypermedia control elements. For the HTTP mapping, this implies, that the different semantics and functions of the HTTP verbs are used properly. If necessary this also means that existing OPC UA services could be modified or new services could be introduced to provide a more RESTful user experience like the integration of the web redirection concept. Finally, the thesis should provide a consistent concept for all the different OPC UA services. Only if all services can be executed through a REST API a full integration into the REST ecosystem can be ensured.

The ecosystem integration of Challenge 2 (C2) can be addressed if the semantics of OPC UA are transformed to the matching semantic concepts of OWL. To achieve this goal, it is necessary to identify the different parts of OPC UA which are semantic meaningful. Additionally, the resulting ontology should provide a good integration into the corresponding ecosystem. This can be achieved, for example, through tools like Protégé which can be used to display OPC UA ontologies (including subtype hierarchies) and ensure modeling constraints based on reasoning. Furthermore, the ontology also has to be in such a form that query functionality can be provided through ecosystem tools like SPARQL. Finally, the scope of this thesis is on the mapping direction from OPC UA to OWL. The inverse mapping direction from OWL to OPC UA is not addressed within this thesis and is out of scope. This also means that the concepts of this thesis might not be sufficient to translate existing OWL ontologies back to OPC UA information models.

Finally, the ecosystem integration of an efficient OPC UA query API, Challenge 3 (C3) can be quantified through the successful execution of the example queries of OPC UA Part 4 Annex B, which can be seen as some kind of baseline for efficient access to huge OPC UA data models. Furthermore, a mapping shall be provided which is able to translate the OPC UA specific query language into SPARQL queries. Based on the fact, that there is no existing public available prototype for the OPC UA query language, shortcomings of the OPC UA query language should be identified and if possible alternative concepts to address the underlying problem shall be identified. Furthermore,

the concept should also cover scalability requirements and concepts for data aggregation to also address huge aggregated information models.

1.3 Contributions

This thesis contributes to the research areas of (Semantic) Web, OPC UA, and the Industrial Internet of Things in general.

First, the thesis proposes a method for **web access to OPC UA data** through the introduction of the representational state transfer (REST) paradigm into OPC UA. Within Section 3.1 the evaluation metrics are derived and applied to the corresponding related work in this area. Based on this evaluation the open research challenges are identified. The first sections of Chapter 4 highlight the solution proposal to address the open research challenges. Section 4.2 focuses on the contributions to standardization while the other sections provide insights into the not standardized parts of the solution proposal. Details of the prototypical implementation are presented in Section 4.7. Finally, Section 4.8 shows the evaluation results of the proposed solution concept and gives a final statement of how well the open research challenges could be addressed through this thesis.

Second, a concept is introduced to automatically transform the **semantics of OPC UA information models** to a formal OWL ontology. Also in this case Section 3.2 derives the evaluation metrics and applies these metrics against related work in this area of research. Based on the evaluation results the open research challenges are identified and the corresponding solution concepts are presented within Chapter 5. Insights into the demonstrator of the transformation tool are given in Section 5.12. Finally, the evaluation of the solution proposal is presented in Section 5.13 also including a statement of how well the open research challenges are addressed.

Third, based on the OWL transformation a method is introduced how the **OPC UA query** language can be translated into SPARQL and also how native SPARQL queries can be formulated. Section 3.3 derives the corresponding evaluation metrics and evaluates the related work according to these metrics. The result of this evaluation is used to formulate the open research challenges. In Chapter 6 the first sections focus on the solution proposal for the open research challenge including insights into the prototypical implementation of Section 6.1. Finally, Section 6.6 provides the evaluation of the given solution proposal and concludes with a final statement on of how well the open research challenges can be addressed by this thesis.

1.4 Overview

The remainder of this thesis is structured as follows (see also Figure 1.1):

OPC UA web access the **analysis** for this research challenge is provided through Section 2.1, Section 2.4 and Section 3.1. The first section highlights certain aspects of the OPC UA technology like the different communication paradigms (Client/Server and PubSub) and the *Subscription* model. In Section 2.4, a brief insight into the requirements of Representational State Transfer (REST) according to Fielding [46] is given. Section 3.1 derives the evaluation metrics, analyzes the related work, and formulates the research challenge. The **synthesis** of the research challenge is provided within Chapter 4 and Chapter 7. Chapter 4 discusses the solution concept, the demonstrator, and finally concludes with an evaluation of the presented solution. Besides the presentation of the overall architecture, the contributions to the OPC UA standardization, the mapping to HTTP, as well as selected service mappings are presented in greater detail. Finally, Chapter 7 provides a summary, conclusion, as well as an outlook for OPC UA web access.

OPC UA semantics presents the **analysis** for the research challenge through Section 2.1, Section 2.2, and Section 3.2. Within Section 2.1, the different modeling elements (nodes and edges) of the information model of OPC UA are explained and a basic modeling example is presented. The Web Ontology Language (OWL) is presented in Section 2.2, starting with the different modeling elements and finishing with different concepts to make statements above these modeling elements. The evaluation metrics, the evaluation of related work, as well as the formulation of the open research challenge is summarized in Section 3.2. Chapter 5 and Chapter 7 provide the **synthesis** for this research challenge. Chapter 5 contains a formal mapping between OPC UA information models and the OWL language. This shows in particular how any OPC UA-based model can be represented in OWL as a basis for automated transformation. The chapter concludes with a prototypical implementation and an evaluation of the presented solution. Finally, Chapter 7 provides a summary, conclusion, as well as an outlook for OPC UA semantics.

OPC UA query provides the **analysis** for this research challenge in Section 2.1, Section 2.3, and Section 3.3. Within Section 2.1, the OPC UA query API is introduced also including the example information model of OPC UA Part 4 and the corresponding query examples. In the following, Section 2.3 focuses on the SPARQL technology. First, the basic concepts are introduced like the different areas of a SPARQL query. Second, graph patterns and expressions that are used within this thesis are explained in further detail. Section 3.3 analyzes the related work, derives the evaluation metrics, and formulates the research challenge. The **synthesis** of the research challenge is provided in Chapter 6 and Chapter 7. The native SPARQL language is compared with OPC UA in Chapter 6. In addition, a mapping from OPC UA query to SPARQL and the prototypical setup and architecture is presented. The chapter closes with an evaluation of all example queries of OPC UA Part 4 Annex B (complex examples) and an assessment of the presented solution against the

research metrics of Section 3.3. Finally, Chapter 7 provides a summary, conclusion, as well as an outlook for OPC UA query.

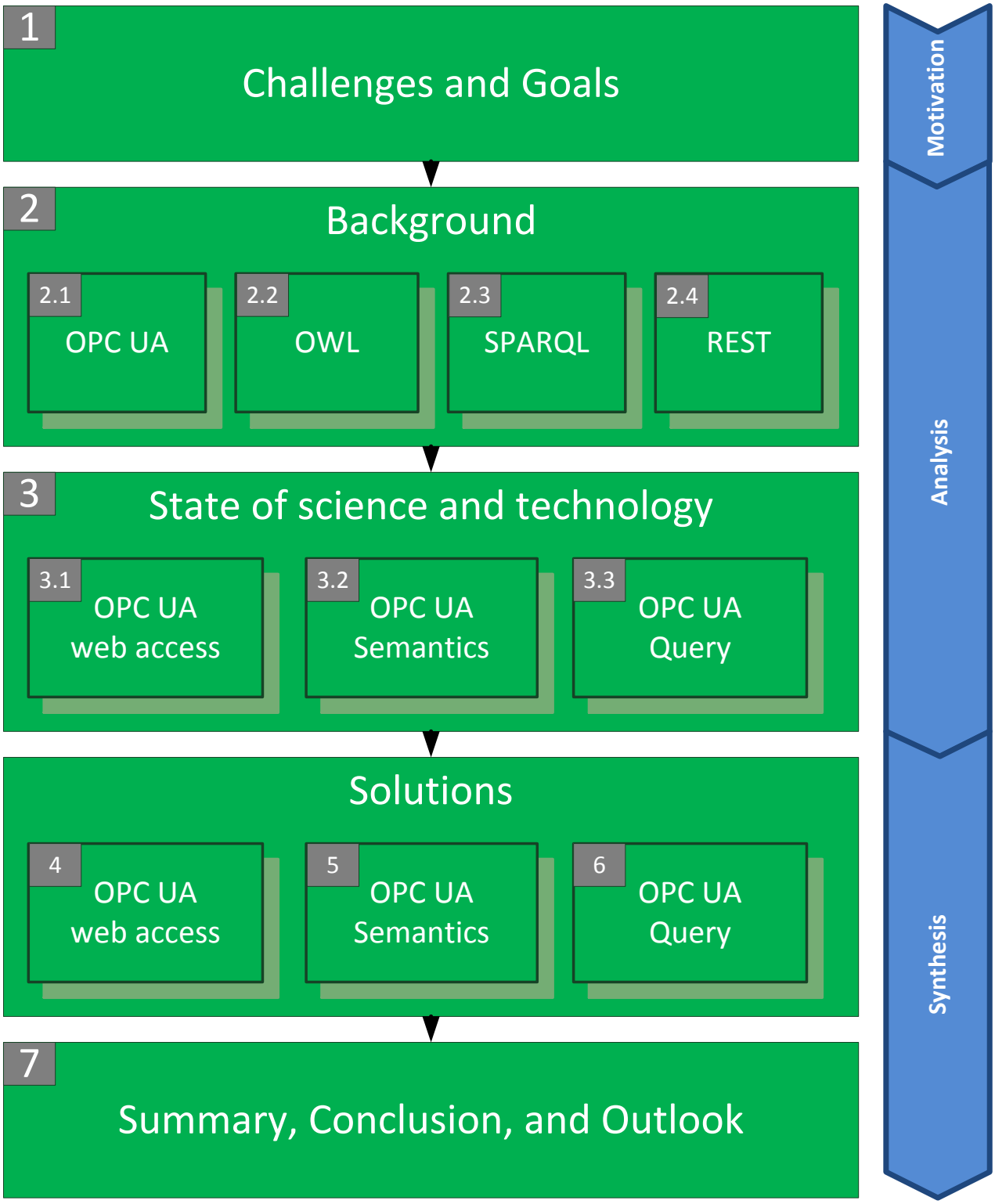


Figure 1.1 – Structure of this thesis.

2 BACKGROUND

In this chapter, the basic concepts of OPC UA relevant to this thesis are presented and explained (Section 2.1). Furthermore, overviews of OWL (Section 2.2), SPARQL (Section 2.3), and REST (Section 2.4) are given to provide enough background to understand the mappings to OPC UA.

In the area of automation, OPC Unified Architecture (OPC UA) [77, 97] is one of the most important standards for device communication and promises to lift low-level signal exchange schemes onto a semantic level, contributing to the realization of flexible manufacturing scenarios. To finally reach this goal the OPC Foundation was very busy building the foundation for future Industry 4.0 scenarios in the last few years. Examples of recent activities are: a cloud interface based on the PubSub pattern (finished)[111]; the introduction of domain-specific semantics which mainly are developed by the VDMA (ongoing)[161]; real-time capabilities for OPC UA (ongoing)[113]; the so-called Field Level Communication (FLC) which aims to unify the different field bus stakeholders (ongoing)[110]; the support of dictionaries like ECLASS (ongoing)[112].

2.1 OPC Unified Architecture

Finally, all these activities resulted in the current OPC UA architecture of Figure 2.1. The main architecture of OPC UA consists of two pillars (OPC UA client/server and OPC UA PubSub), which are unified through a common data layer based on OPC UA information models. Each of these pillars introduces different transport technologies. For example, in the case of client/server UA TCP, HTTPS, and WebSocket communication can be chosen and in the PubSub case UDP, MQTT, and AMQP. While UA TCP is mostly used by small embedded devices in the lower layers of a factory network, transport protocols like HTTPS and WebSocket communication are more common in the upper layers like edge and cloud. PubSub implementations mainly focus on the UDP and MQTT variant. Furthermore, OPC UA provides an own security layer to encrypt messages and thus allows end-to-end encryption (*Secure Channel* and *Secure Message* of Figure 2.1). The next higher layer is the serialization layer. OPC UA offers different forms of serialization like a highly optimized proprietary binary encoding (*UA Binary* and *UADP*) but also supports well-known serialization

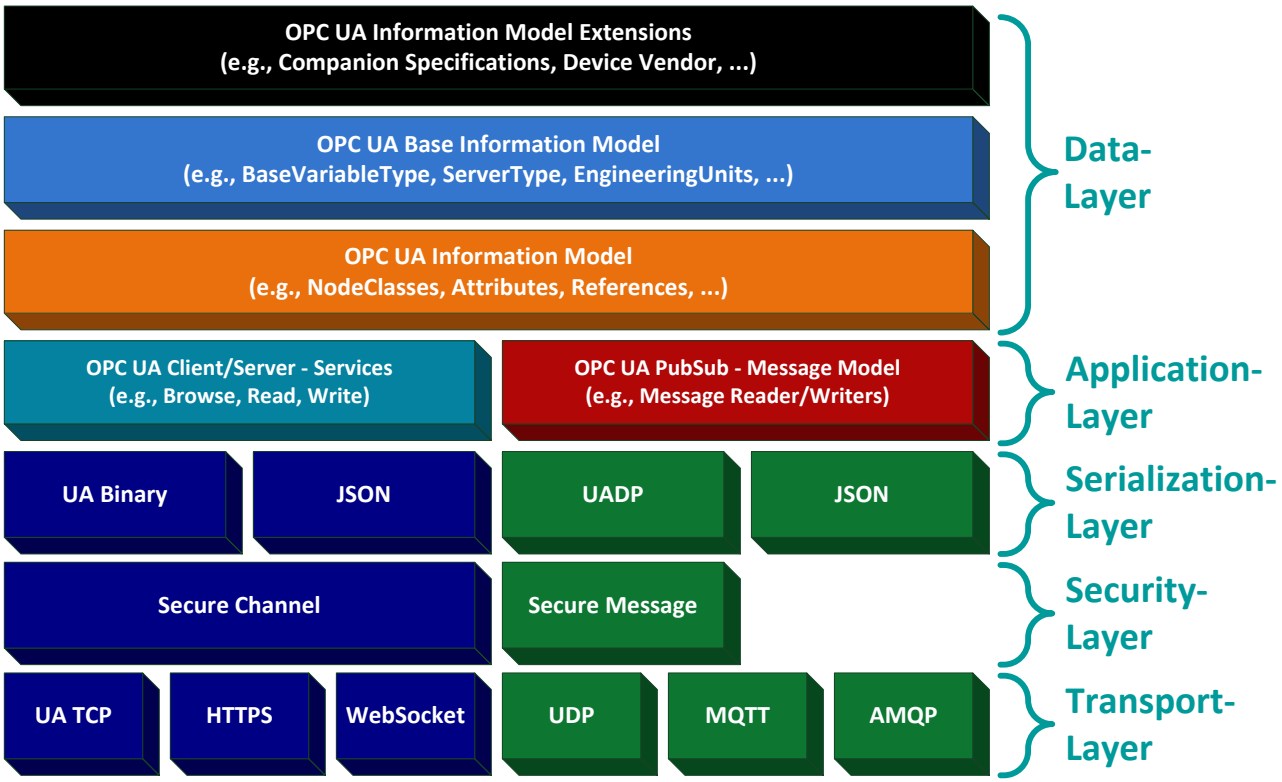


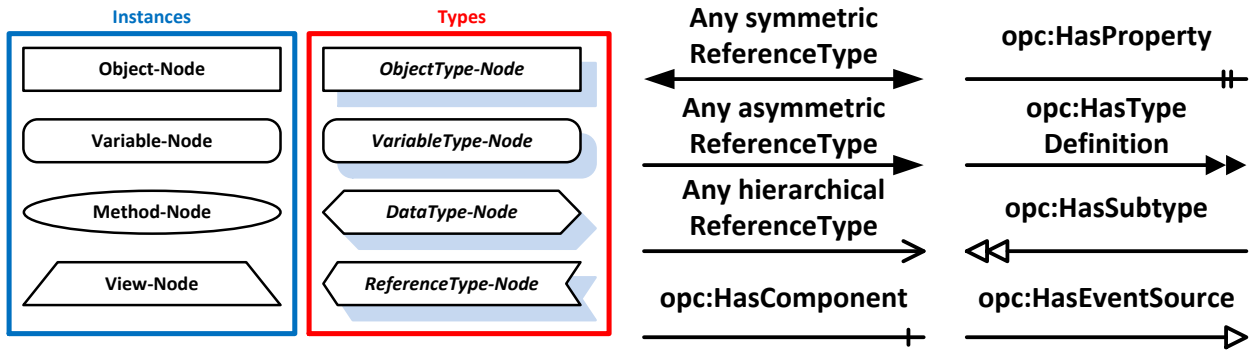
Figure 2.1 – OPC UA basic architecture overview (simplified).

formats like JSON. The top layer of the pillars is the application-layer. For OPC UA client/server the application-layer is structured around a service-oriented design pattern consisting of several services to introspect (e.g., *Read* service) and manipulate (e.g., *Write* service) the data-layer (see also OPC UA Part 4). In contrast, OPC UA PubSub is built around the publish-subscribe pattern and focuses on the definition of message streams (see also OPC UA Part 14). Nevertheless, regardless of the application-layer, every information exchange of OPC UA is grounded in the extensible graph-based OPC UA information model (data-layer of Figure 2.1).

Section 2.1.1 covers the data-layer in greater detail, while Section 2.1.2 provides details about OPC UA client/server and OPC UA PubSub. Section 2.1.3 highlights some aspects around the OPC UA *Query* service and Section 2.1.4 provides insights in OPC UA *Subscriptions*.

2.1.1 Data Layer

The main concept of OPC UA information models is a graph architecture. The nodes of the graph are also called *Nodes* and the edges between the nodes are named *References* in OPC UA. Each *Node* in OPC UA can be uniquely identified through a *NodeId*. Furthermore, OPC UA introduces eight different *NodeClasses*, which can be categorized into a type and instance category (see also



(a) Graphical notation of *Nodes*.

(b) Graphical notation *ReferenceTypes*

Figure 2.2 – Graphical notation of OPC UA.

Figure 2.2a). *View-*, *Variable-*, *Object-*, and *Method-Nodes* are part of the instance category (left side of Figure 2.2a). *DataType-*, *VariableType-*, *ObjectType-*, and *ReferenceType-Nodes* are part of the type category (right side of Figure 2.2a). These two categories can be compared with object-oriented programming languages, where types can be considered classes and the other category is used to instantiate these classes. For example, a smart sensor typically offers several measurement values, which could be exposed through a new *ObjectType-Node*. This *ObjectType-Node* would expose the common structure for all smart sensors of a particular category and also offers the semantics behind the values (e.g., a temperature value in degree Celsius). A sensor instance would then be modeled through an *Object-Node*, which would refer to the newly introduced *ObjectType-Node* as its type. Furthermore, OPC UA also introduces different types to describe the semantics of the edges (see also Figure 2.2b). Also, edges are described through a *NodeClass* named *ReferenceType*. In the following, a basic description for each *NodeClass* is given:

Objects are mainly used to introduce structures. For example, *Objects* can be used to expose machines, components of the machine, software artifacts, and much more. The main difference between *Variables* and *Objects* is, that *Objects* missing the possibility to expose actual values (e.g., the actual temperature value of a sensor). However, a smart sensor could also be exposed as an object with several attached *DataVariables* and *Properties*. Furthermore, if an object is used instead of a *Variable* it is also possible to attach *Method-Nodes*, which could offer functions like calibration. Eventually, it depends on the use case what kind of *NodeClass* should be used.

Variables in OPC UA fulfill several aspects. The most important part is to offer process information like the actual temperature value. However, besides process information *Variables* also can be used to provide meta-data like the engineering unit of a specific sensor value. OPC UA introduces two main concepts around *Variables*: (1) *DataVariables* are used to expose *Variables* with substructures. For example, a smart sensor, which offers the value filtered and unfiltered; (2) *Properties* are used to describe the referencing *Node*. For example, the smart sensor of the previous example also has to provide the engineering unit of the value, which could be described through a

Property. The main difference between both concepts is, that *Properties* neither can be subtyped, nor can be used to provide substructures, which also implies that a *Property* cannot have its own *Properties*.

Methods in OPC UA have the exactly same semantics as their counterparts in object-oriented programming languages. They have input- and output-arguments and can be executed through the *Call* service of OPC UA. Nevertheless, because *Methods* always run to completion within OPC UA several design guidelines exists around this concept. *Methods* in OPC UA should be lightweight, which means it should never take several hours to execute a *Method*. If long-running processes have to be modeled the OPC UA state-machine concept should be used instead (see also OPC UA Part 10). In addition, a method always has an owning *Object*, similar to the corresponding class instance in object-oriented programming languages.

Views are used in OPC UA to provide different aspects of the same information model. For example, a machine may introduce an operator and a maintenance *View*. The former *View* is used by the production personnel during operation, while the latter one is used by service-engineers to carry out maintenance tasks. Each *View* provides only the essential information for the different users and thus allows a much more efficient information extraction. *Views* can be used in two different ways: (1) The *View-Node* acts as an entry point into a sub-graph; (2) The *View* hides certain *References* on selected *Nodes*.

ObjectTypes are the *Type-Nodes* for *Object-Nodes*. Each *Object-Node* specifies exactly one *ObjectType-Node* as its *Type*. Also in the case of *ObjectTypes* OPC UA defines several different standard *ObjectTypes* like the *BaseObjectType*. However, it is also allowed to define user-specific new *ObjectTypes*.

VariableTypes are the corresponding *Type-Nodes* for *Variable-Nodes*. Similar to *Object-Nodes*, also *Variable-Nodes* specify exactly one *VariableType-Node* as its *Type*. OPC UA defines several standard *VariableTypes* like the *PropertyType* and the *BaseDataVariableType*. All *Types* offer the possibility to define abstract *Types*. Abstract *Types* cannot be instantiated but further subtyped with concrete *Types*.

DataTypes: As previously mentioned, *Variable-Nodes*, as well as *VariableType-Nodes*, are able to expose a value of a sensor. For example, a temperature value could be represented as an integer but also as a floating point value. So, each *Variable* also specifies the corresponding *DataType* to further specify the format of the value. OPC UA introduces several different *DataTypes* like String and Double but also allows to define, for example, user-specific *Structures*, which consist of several simple *DataTypes*. Furthermore, OPC UA also defines *Enumerations* in a similar way than object-oriented programming languages. These *DataTypes* can also be further subtyped to refine constraints and semantics. Finally, it is also possible to subtype simple *DataTypes* to add more concrete semantics like a special encoding for pictures.

ReferenceTypes are used to expose the semantics and constraints of the graph edges. Similar to all other *Type-Nodes ReferenceType-Nodes* are organized in a *Type* hierarchy, including semantics and restriction inheritance. To simplify modeling OPC UA defines several standard *ReferenceTypes* and the corresponding graphical notation (see also Figure 2.2b). *ReferenceTypes* can be categorized into two main groups, hierarchical- and non-hierarchical-*ReferenceTypes*. The former ones can be used to expose tree-like structures in graphical tools, while the latter ones are used to expose information, which cannot be organized in a hierarchically fashion.

Each *NodeClass* defines its own set of *Attributes*. While some *Attributes* are common across several different *NodeClasses* (e.g., *DisplayName*, *BrowseName*, etc.), other *Attributes* are exclusive for a single *NodeClass* (e.g., the symmetric-*Attribute* of the *ReferenceType*). Annex A.3.1 further details the *Attributes* of each *NodeClass*. Finally, Figure 2.3 shows an example information model in the graphical notation, which is used throughout this thesis. The **opc:BName** *Variable-Node* is attached to an *Object-Node* through a *HasProperty-Reference* and specifies the *PropertyType* as its *TypeDefinitionNode*. Within this thesis, *BrowseNames* are used in the graphical notation, including a namespace-prefix (e.g., "opc:" and "rs:"). The samples shown in this thesis often hide information that is unimportant in the concrete context. So, the **rs:SampleObject** also references the *BaseObjectType* of the OPC UA *Namespace* with a *HasTypeDefinition-Reference* but this is left out in Figure 2.3 for brevity.

However, to finally achieve interoperability on the semantic layer it is not enough to only provide the syntax of a modeling language, instead, also the vocabulary must be provided. While an abstract cross-domain vocabulary is provided by the OPC Foundation itself, domain-specific vocabulary are added through a standardization process for OPC UA information models. The resulting artifact of this process is a *Companion Specification*. In previous years most of the *Companion Specification* were mappings from other already existing standards to OPC UA like AutomationML, PLCopen, ISA-95, etc. [15, 121, 78]. Eventually, all these standards are rather

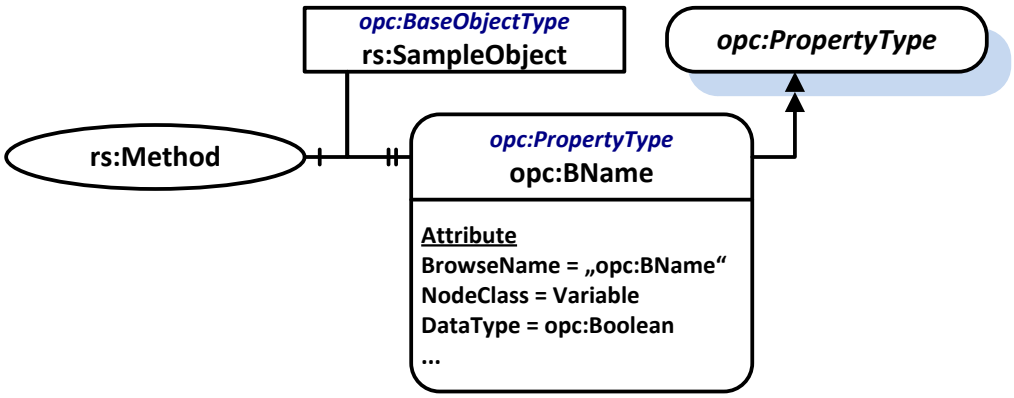


Figure 2.3 – Example OPC UA graphical notation used throughout this thesis.

abstract and define mainly generic semantics. However, in the last few years also *Companion Specification* started to emerge with detailed descriptions of the underlying devices (e.g., machine vision, robotics, powertrain, CNC machines, etc. [114]). These specifications are mainly driven by the VMDA [160]. The VDMA can be considered the largest industry association in Europe and represents more than 3200 companies within the manufacturing domain.

2.1.2 Application Layer

Since version 1.04 of OPC UA two main access patterns exist to interact with an OPC UA server. The classic way, based on the client/server pattern (Section 2.1.2.1) and the new cloud interface, which is based on a publish-subscribe pattern (Section 2.1.2.2).

2.1.2.1 OPC UA Client/Server

The OPC UA client/server architecture was the only available communication pattern until version 1.04 of OPC UA. Based on the client/server pattern, OPC UA introduces different services to define the interaction between client and server. These services are logically grouped to *Service Sets*. Each

Service Set	**Description**
Discovery	Offers services to find servers and the corresponding endpoints.
SecureChannel	Is used to establish secure connections between server and clients.
Session	Contains all services which are related to sessions (e.g., authentication).
NodeManagement	Includes services to alter the graph-based information model.
View	Offers services to explore the information model.
Query	Introduces a OPC UA query language and the necessary services to execute queries against OPC UA information model.
Attribute	Can be used to read and write *Attributes*.
Method	Exposes the *Call* service which is used to invoke OPC UA *Methods*.
MonitoredItem	Contains services for creation and modification of *MonitoredItems*, which can be used in combination with the *Subscription Service Set* to monitor, for example, sensor values for changes.
Subscription	Is used in combination with the *MonitoredItems* to create notifications for events and value changes.

Table 2.1 – OPC UA *Service Sets* [77].

Service Set comprises all services belonging to the same functionality (see also Table 2.1). Before most of the services can be used the client has to execute several steps: (1) The client opens a secure channel with the *OpenSecureChannel* service of the *SecureChannel Service Set*; (2) After the secure channel is established the *CreateSession* service (*Session Service Set*) is used to create a new session; (3) The next step is to activate the newly created session with the *ActivateSession* service, which is also part of the *Session Service Set*; (4) If every service execution completed successfully the client can now use the *Read* service from the *Attribute Service Set* to fetch the *NamespaceArray* from the server; (5) Based on the content of the *NamespaceArray*, the client is now able to generate *Node* addresses and use other services to interact with the graph.

2.1.2.2 OPC UA PubSub

The basic architecture of OPC UA PubSub is shown in Figure 2.4. The middle part of the picture exposes the three main elements of the OPC UA PubSub communication pattern: *MetaData*, *Data Messages*, and security key handling. A *Data Message* contains the data which is published by the Publisher. Data Messages can be transmitted via broker-less or broker-based communication using different transport protocols (e.g., MQTT or AMQP). The broker-less architecture is most famous in environments like the factory shop floor, where tight time constraints have to be fulfilled. In future scenarios, OPC UA PubSub UDP in combination with TSN might be able to address hard real-time use cases. In the broker-based scenario, all *Publishers* and *Subscribers* are connected to a broker. However, from an architectural point of view, the broker can also be realized based on a distributed system, which allows scaling with the number of *Subscribers* and *Publishers*. The content of *Data Messages* (syntax and semantics) is described within the *MetaData*. The exchange

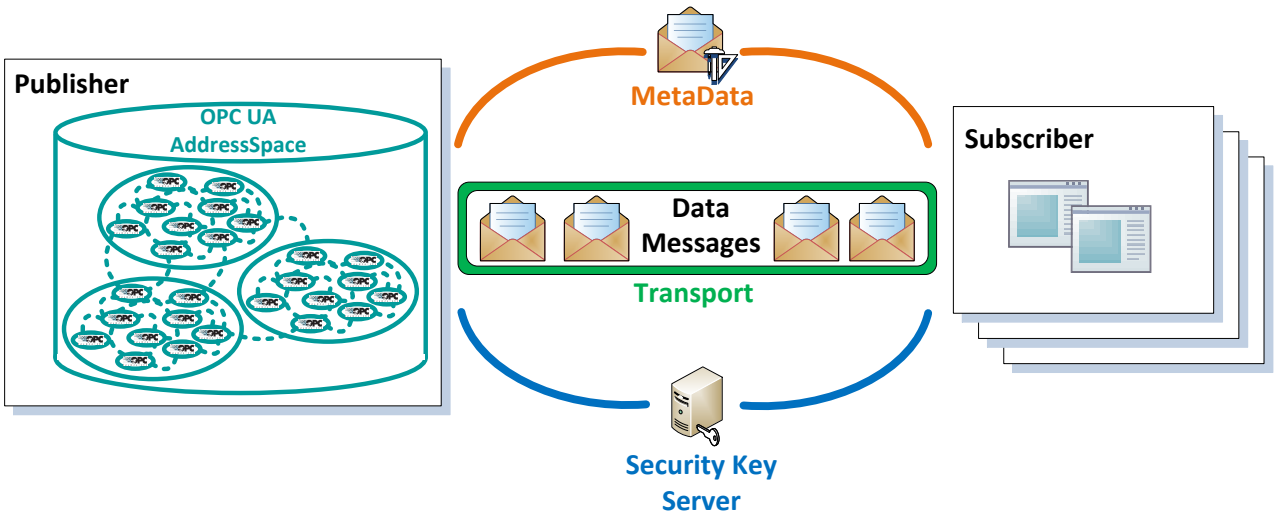


Figure 2.4 – OPC UA PubSub architecture overview (simplified).

of the synchronous security keys is managed by the *Security Key Server*. *Publishers*, as well as *Subscribers*, can request the keys for a given security group after successful authentication on the *Security Key Server*. Normally, the keys have a limited lifetime and have to be refreshed regularly, which allows to remove previously participating *Publishers* and *Subscribers*.

2.1.3 OPC UA Query

At the time of writing of this thesis, the OPC UA *Query Service Set* is probably the only OPC UA *Service Set* that has no publicly available implementation up to now. Even the fact that a whole annex was introduced in OPC UA Part 4 (see also Annex A.3.2 for the example type and instance model of OPC UA Part 4), which added a lot of examples for this service, provided not enough motivation for prototypical implementations. OPC UA *Queries* can be disassembled into two main parts. 1) A filter part, which is used to select what kind of *Instances* should be returned. OPC UA *Query* allows, for example, to filter on *Views* or on *Types*. However, also more sophisticated filter arguments can be formulated like a greater than relation, or even a complete graph pattern that has to be fulfilled. 2) The data which should be returned. Also in this case OPC UA *Query* allows defining relations across several intermediate *Nodes*. This architecture allows formalizing a wide range of queries. For example, it is possible to filter for the maintenance date of certain machines and after that return the exact position of these machines. Based on these results an efficient maintenance schedule could be generated. For the filter part, OPC UA defines several different *FilterOperator* and *FilterOperand* parameters (see also OPC UA Part 4). Most of *FilterOperators* can be easily understood only through the name like the "equals" *FilterOperand* and are not further discussed in this thesis. However, the *RelatedTo FilterOperator*, which is used to model graph-patterns, has very high complexity and is introduced in Annex A.3.2. In the case of *FilterOperands*, the *AttributeOperand* is identified as the operand with the highest complexity and is also included in Annex A.3.2. In the following, the focus is shifted to the *QueryFirst* service of OPC UA Part 4 (see also Table 2.2).

The **view** parameter is used to select a *View*. A *View* in OPC UA typically contains only parts of the *AddressSpace*. For example, a maintenance-view might only include *Nodes* and *References* which are relevant for a service-engineer and hide all the other *Nodes* and *References*. The **nodeTypes** array contains elements of the *NodeTypeDescription* structure (marked with indents). The **typeDefinitionNode** selects the *Instances* for this service. Only *Instances* of this *Type* or subtypes (if the **includeSubtypes** parameter is true) are considered as valid results. The **dataToReturn** array (*QueryDataDescription*, marked with indents) is used to select the data which shall be returned and consists of three sub-parameters: **relativePath**, **attributeId**, and **indexRange**. The **relativePath** is used to define a path from the filtered *Instances* to a target *Node* or target *Reference* across several intermediary *Nodes*. Moreover, the **relativePath** has slightly different behavior in OPC UA *Query*

Request		Response	
Name	**Type**	**Name**	**Type**
requestHeader	RequestHeader	responseHeader	ResponseHeader
view	ViewDescription	queryDataSets[]	QueryDataSet
nodeTypes[]	NodeTypeDescription	nodeId	ExpandedNodeId
typeDefNode	ExpandedNodeId	instanceTypeDefNode	ExpandedNodeId
includeSubtypes	Boolean	values[]	BaseDataType
dataToReturn[]	QueryDataDescription	continuationPoint	ContinuationPoint
relativePath	RelativePath	parsingResults[]	ParsingResults
attributeId	IntegerId	statusCode	StatusCode
indexRange	NumericRange	dataStatusCodes[]	StatusCode
filter	ContentFilter	dataDiagnosticInfos[]	DiagnosticInfo
maxDataSets	Counter	diagnosticInfos[]	DiagnosticInfo
maxReferences	Counter	filterResult	ContentFilterResult

Table 2.2 – *QueryFirst Service* Parameters [77].

than it has in the *TranslateBrowsePathsToNodeIds* service. To be able to address all use cases of OPC UA *Query* it is necessary to specify sometimes the *Type* of the *Node* instead of the *BrowseName*. This can be done by setting the *NamespaceIndex* to zero and the string part of the *QualifiedName* to the XML representation of the *NodeId*. The **attributeId** and **indexRange** are applied to the target *Node* of this path if a target *Node* exists. The **filter** parameter is one of the most complex parameters within the complete OPC UA specification. A **filter** consists of several so-called *filterOperators* (e.g., *equals*, *greaterThan*, *and*, *or*, *relatedTo*, ...) and so-called *filterOperands*, which are the input parameter for the *filterOperators*. It is also possible to combine different *filterOperators* to build very expressive filters. Additionally, OPC UA defines conversion rules, which introduce an additional layer of complexity because of some unexpected implicit conversions (e.g., implicit casts from *String* to *Byte*). The parameters **maxDataSetsToReturn** and **maxReferencesToReturn** are used to limit the maximum number of results.

The response structure of a *QueryFirst* request starts with a **queryDataSets** array. Each array entry consists of a **nodeId**, which is the *NodeId* (a unique identifier in OPC UA) of an *InstanceNode* of the requested *typeDefinitionNode* with all restrictions (e.g., **view** and **filter**) applied. The **instanceTypeDefinitionNode** is the corresponding *TypeDefinition* of the *Instance*. This parameter is important because also subtypes of the *TypeDefinition* can be returned and it is also allowed to define an array of **nodeTypes** in the request, which typically define different **typeDefinitionNodes**. Finally, the **values** array contains the requested results defined by **dataToReturn**. The **continuationPoint** parameter is used if not all results can be returned in a single response. The **parsingResults** parameter contains the list of parsing results for the *QueryFirst* service, while the

diagnosticInfos parameter contains diagnostic information for the requested *NodeTypeDescription*. Finally, the **filterResult** parameter contains information about **filter** errors.

In addition to the *QueryFirst* service, the OPC Foundation specified the *QueryNext* service to retrieve further results if not all results could be returned in a single response (see also Table 2.3).

The **releaseContinuationPoint** parameter can be used to request the next results for the given *ContinuationPoint* (if set to "false"), or to delete all further results (if set to "true"). A client shall always use this service to free resources if a *ContinuationPoint* is returned by a *QueryFirst*, or *QueryNext* call. The **continuationPoint** parameter is used to enter the *ContinuationPoint* of a previous *QueryFirst*, or *QueryNext* call.

The response structure of a *QueryNext* request starts, identical to *QueryFirst*, with a **queryDataSets** array. This array has the same content structure as the *QueryFirst* array. The **revisedContinuationPoint** parameter is used if not all results can be returned in a single response. If the **releaseContinuationPoint** parameter is set to "true" this value is always "null".

2.1.4 OPC UA Subscriptions

OPC UA *Subscriptions* consist of several *MonitoredItems*. A *MonitoredItem* can be configured to monitor *Attributes* or *Events* (see also Figure 2.5). A *MonitoredItem* has several parameters: The **ItemToMonitor** parameter specifies the *Node* which shall be monitored, while the **MonitoringMode** introduces three different states (disabled, sampling, and reporting). The *RequestedParameters* parameter specifies, for example, the **QueueSize**, **SamplingInterval**, and **Filter**. OPC UA defines three different **Filter** parameters for *MonitoredItems*: *DataChangeFilter*, *EventFilter*, and *AggregateFilter*. This allows configuring the *MonitoredItem* very fine-grained. *Subscriptions* can also be further parameterized. If the *Subscription* and the corresponding *MonitoredItems* are generated and configured, the client starts to call the *Publish* service. With this service, new notifications can be requested and old ones can be acknowledged. Besides the basic parameterization shown in Figure 2.5, OPC UA also defines a *Triggering Model*. In a nutshell, this concept allows defining several *MonitoredItems*, which sample values but do not report these values to the underlying *Subscription* (*ItemsToReport*). Furthermore, another *MonitoredItem* named *TriggeringItem* is configured and assigned to the *ItemsToReport*. If now the *TriggeringItem* creates a notification the *ItemsToReport*

Request		**Response**	
Name	**Type**	**Name**	**Type**
requestHeader	RequestHeader	responseHeader	ResponseHeader
releaseContinuationPoint	Boolean	queryDataSets[]	QueryDataSet
continuationPoint	ContinuationPoint	revisedContinuationPoint	ContinuationPoint

Table 2.3 – *QueryNext Service* Parameters [77].

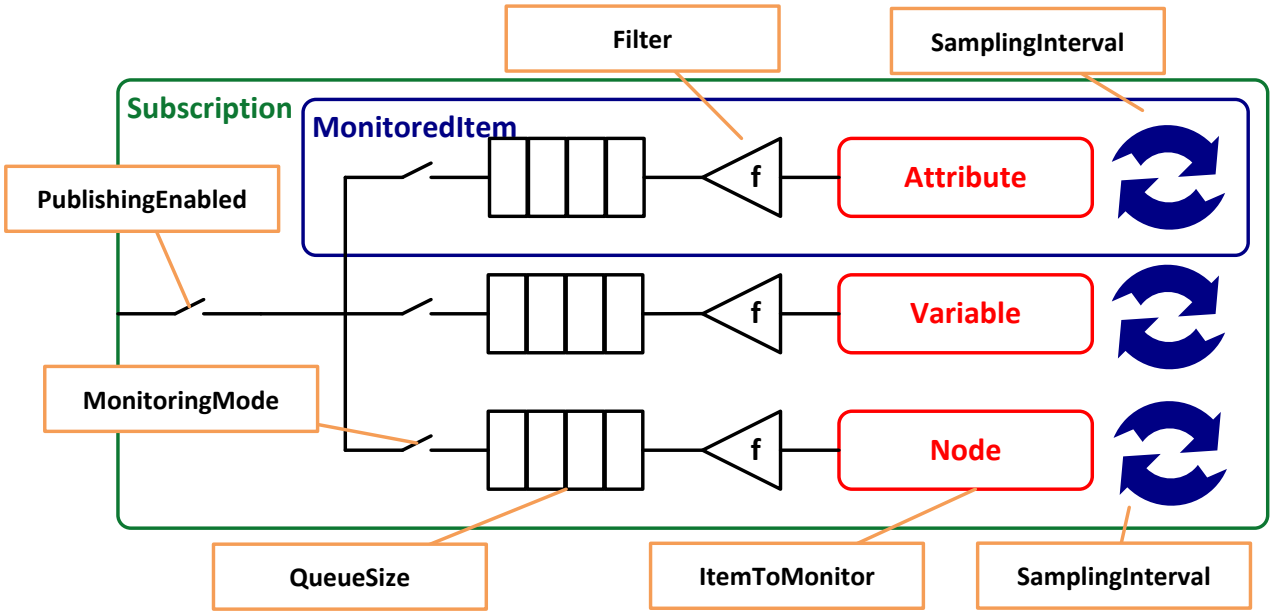


Figure 2.5 – OPC UA Subscriptions basic overview.

also reports their values to the underlying *Subscription*. This allows to efficiently collect data if certain conditions are true.

In the following, the differences between both concepts are further explained. While in the case of OPC UA client/server *Subscriptions* are client-specific and cannot be shared between different clients, OPC UA PubSub allows different clients to use the same *Subscription*. In contrast, only OPC UA client/server *Subscriptions* define the *Triggering Model* concept, which is not available for OPC UA PubSub *Subscribers*.

2.2 Web Ontology Language (OWL)

Within this section, the basics of OWL [116] is presented (Section 2.2.1), followed by an overview of further concepts, which can be used to express constraints and semantics of the main building blocks (Section 2.2.2). Based on the formal definition of OWL, OWL Reasoners [68, 44] can automatically perform reasoning tasks such as checking consistency and inferring implicit relationships.

2.2.1 Basic concepts

OWL Individuals denote objects in OWL. For example, a concrete person like Anna or John could be described as an individual. OWL defines two groups of individuals: (1) Named individuals, which can be identified through an IRI [41]; (2) Anonymous individuals, which can only be identified based on a local node ID as an identifier rather than a global IRI and because of that cannot be

referenced from an outside ontology. Each individual can be described through annotation property assertions, class assertions, object property assertions, and data property assertions. Furthermore, OWL allows specifying if individuals are identical or different.

OWL Classes denote classes of objects. The classes Person and Engineer are examples of possible OWL classes. OWL already introduces two classes owl:Thing, which represents the set of all individuals, while owl:Nothing is the empty set. Besides these two classes, further classes can be introduced and organized in subsumption hierarchies. Finally, each class can be described through annotation property assertions and further axioms like the subclass axiom. Similar to individuals (owl:samesAs and owl:differentFrom) also for OWL classes concepts for equivalence and disjointness exist.

OWL Object Properties relate objects to objects (e.g., relating a child to their parent with hasChild). OWL introduces two object properties owl:topObjectProperty, which connects all possible pairs of individuals, while owl:bottomObjectProperty does not connect any individual at all. Similar to classes also object properties can be organized in a subsumption hierarchy. Each object property can be described with annotation properties and characteristics like functional, inverse functional, transitive, symmetric, asymmetric, reflexive, and irreflexive. Furthermore, inverse object properties can be specified as well as domain and range restrictions. Finally, equal to classes also object properties have a concept for equivalence and disjointness.

OWL Data Properties assign data values to objects (e.g. relating a height to a person). Also in this case two data properties are introduced by OWL. The owl:topDataProperty connects all possible individuals with all literals, while the owl:bottomDataProperty does not connect any individual at all. Similar to classes data properties can be organized in subsumption hierarchies. Besides the usage of annotation properties for the description of data properties, domains, and ranges can be defined. Furthermore, each data property also can be defined as functional and identical to classes, a concept for equivalence and disjointness exists.

OWL Annotation Properties can be used to record ontology meta-information, such as the author and creation date. OWL as well as RDFS defines several annotation properties like rdfs:label and owl:versionInfo. Also, annotation properties can be organized in subsumption hierarchies and also offer the possibility to be described through other annotation properties. Furthermore, range, as well as domain restrictions, can be defined.

OWL Datatypes denote data values (e.g., the name John is of the datatype xsd:string). OWL already defines several datatypes like owl:rational or owl:real and also makes use of several XML schema datatypes like xsd:float. In addition, datatypes can be further described with annotation properties and also allow to define datatype definitions.

2.2.2 Expressions, Axioms, Assertions, and Data Ranges

The most important concepts of OWL used within this thesis can be categorized in OWL expressions, axioms, assertions, and data ranges. In this section, the used concepts of each category are explained in greater detail.

OWL Expressions: OWL provides constructs for building complex expressions on classes and properties, such as: intersection (Mother and Female), union (Male or Female), existential restrictions (a class of Persons who have at least one child), universal restrictions (a class of Persons who have only male children), minimum cardinality restrictions (a class of Persons who have at least 3 children), maximum cardinality restrictions (a class of Persons who have at most 3 children), exactly cardinality restrictions (a class of Persons who have exactly 3 children), literal value restrictions (a class of Persons who have children with the name Anna). Notice, that most of these restrictions can also be applied for OWL object properties and OWL data properties. For the sake of simplicity, the examples are provided only for the OWL class concept.

OWL Axioms: OWL defines several axioms, for example, class subclass axioms (Female subClassOf Person), disjointness axioms (Female disjointWith Male), equivalence axioms (Person equivalent (Male or Female)), property domain axioms (if a person relates to a child with a hasChild property, this person is also of class Parent), property range axioms (if a child relates to a person with a hasMother relation, this person is also of class Female), inverse object property axioms (the childOf relation is the inverse object property to hasChild), functional axioms (a child can have only one mother), irreflexive object property axioms (A child cannot be its own child), symmetric object property axioms (if a man is married to a woman also the woman is married to the man), transitive object property axioms (parents are related to their children and also to the children of their children). Notice that, also in this case the restrictions and examples are only provided for OWL classes and sometimes for OWL object properties if they do not apply to OWL classes but also can be found on, for example, OWL data properties.

OWL Assertions: Individual axioms make statements about individuals like positive property assertions (Anna has John as a child), negative property assertions (Jack is not a child of Anna), class assertions (Anna is of class Female), individual equality assertions (Anna uses the pseudonym Anna123 on Facebook), individual inequality assertions (Jack and John are different persons).

Data Ranges: The ranges of Datatypes can be restricted by concepts like DataType Restrictions (the age of a teenager has to be below 20), enumeration restriction of literals (e.g., days of the weekends can be named Saturday or Sunday).

2.3 SPARQL Protocol and RDF Query Language (SPARQL)

The SPARQL Protocol and RDF Query Language (SPARQL)[147] is designed to query and manipulate RDF-based data sources and can be considered as the successor of other RDF query languages like RQL, SeRQL, TRIPLE, RDQL, N3, and Versa [65]. Within this section, the basic concept behind SPARQL queries is introduced first (Section 2.3.1), while the second section focuses on some SPARQL graph patterns and expressions used throughout this thesis (Section 2.3.2).

2.3.1 Basic concepts

In the following section, the basic structure of a SPARQL query is explained, starting with the introduction of prefixes for namespaces through Lines 1-2 of Listing 2.1. Also, SPARQL queries make use of a triple like syntax based on RDF as shown in Line 6, where the subject is "ex:Hervey", the predicate is "foaf:knows", and the object is expressed through "?friend". As mentioned previously, subjects are addressed through IRIs, which could be very long and shortened through the SPARQL PREFIX concept. For example, the full IRI of the subject in Line 6 is "http://example.org/Hervey". Line 4 gives an example of a possible query form. SPARQL defines several different query forms like SELECT, ASK, CONSTRUCT, and DESCRIBE. The SELECT form is used to return the results of variables and their bindings directly, while the ASK form returns a boolean value indicating if the defined query pattern matches or not. In the case of Line 6, the SPARQL query shall return all bindings of the **?friend** variable (SPARQL variables are identified through a preceding question mark). Furthermore, the DISTINCT keyword is used to remove duplicates from the result list. Lines 5-7 specify the query pattern enclosed in the WHERE section. Within the WHERE section, SPARQL offers the possibility to formulate very expressive query patterns. Finally, Line 8 shows a further example of how the results can be modified. In this case, the LIMIT modifier with the value ten states that at most ten results shall be returned for the given query. This ensures, that the client is not overwhelmed if the query returns a large number of results. In combination with

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 PREFIX ex:   <http://example.org/>
3
4 SELECT DISTINCT ?friend
5 WHERE {
6   ex:Hervey foaf:knows ?friend.
7 }
8 LIMIT 10
```

Listing 2.1 – Example SPARQL query.

the ORDER BY modifier it is also possible to formulate statements like: Find the ten largest cities in Europe.

2.3.2 Graph patterns and expressions

Within this thesis, several SPARQL graph patterns and expressions are used. To understand the concepts used throughout this thesis the most important SPARQL concepts are now explained in greater detail:

Graph patterns: SPARQL provides constructs to formulate complex graph-patterns such as: SequencePath (the grandchild of a parent, see also Line 1 of Listing 2.2), AlternativePath (the parents of a child, see also Line 2), ZeroOrMorePath (offspring of a person including the person itself, see also Line 3), OneOrMorePath (offspring of a person without the person itself, see also Line 4), OPTIONAL graph patterns (return also the married partner of a given parent if such a connection exists, see also Line 5), the "a" keyword (a shortcut for the IRI "`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`", see also Line 6), UNION (allows to evaluate different graph patterns in one query, see also Line 7).

Expressions can be used to manipulate graph-patterns with SPARQL binary and unary operators such as: equals (evaluates to "true" if variable A and variable B are equal), greater (evaluates to "true" if variable A is greater than variable B), less (evaluates to "true" if variable A is less than variable B), greater equal (evaluates to "true" if variable A is greater or equal compared with variable B), less equal (evaluates to "true" if variable A is less or equal compared with variable B), logical and (evaluates to "true" if variable A, as well as variable B, can be evaluated to "true"), logical or (evaluates to "true" if variable A or variable B can be evaluated to "true"), not (evaluates to "true" if the variable evaluates to "false" and vice versa). Furthermore, SPARQL defines REGEX expressions (evaluates to "true" if variable A matches the specified REGEX pattern), FILTER expressions (if evaluated to "false" the given graph pattern is removed from the result set), COALECSE expressions (returns the first expression without error), EXISTS expressions (returns "true" if the specified graph pattern exists), BOUND expressions (returns true if the specified variable is bound to a

```
1 ?parent ex:hasChild/ex:hasChild ?grandChild.
2 ?child ex:hasMother | ex:hasFather ?parent.
3 ?person ex:hasChild* ?offspring.
4 ?person ex:hasChild+ ?offspring.
5 OPTIONAL {?parent ex:married ?partner}.
6 ?person a ex:PersonType.
7 {graph pattern} UNION {graph pattern}.
```

Listing 2.2 – Example SPARQL graph patterns.

concrete value), CAST expressions (casts a value to a specified datatype), DATATYPE expressions (returns the datatype IRI of the specified variable). Finally, SPARQL also defines aggregate functions like COUNT (counts the number of times a given expression is bound) and also allows to assign variables through a special BIND statement.

2.4 Representational State Transfer (REST)

Representational State Transfer (REST) was introduced by Fielding in [46]. In his work Fielding derives the REST architectural style from the very successful web. Finally, the following rules were identified for a RESTful architecture:

Client-Server: Andrews et al. [8] defined a server as a process that handles repeatedly requests from clients. From that point of view, a client triggers actions on a server and so, can be considered the active part, while the server is the passive part and reacts based on the client requests.

Cache: A cache is a component that is placed between client and server and is able to serve previously cached responses from the server for identical requests. This reduces the server load and message latency.

Uniform interface: Fielding et al. [46] stated out, that the uniform interface is the central feature that distinguishes REST from other network-based styles. The four defined constraints for the REST interface are: the identification of resources; manipulation of resources based on representations; self-descriptive requests and responses; hypermedia as the engine of application state (HATEOAS).

Layered system: Garlan et al. [54] defined layered systems as hierarchically organized systems where each layer provides a service to the layer above and uses services from the layer below. If the inner layers are hidden from all except the adjacent outer layer, the coupling between systems can be significantly reduced.

Statelessness: Statelessness in the context of client-server interaction means, that no session state is allowed. To be more concrete, a client cannot take advantage of previously stored information at the server, like, for example, some language settings, instead, all necessary information to process the request must be included in each message.

Code-on-demand: Fugeta et al. [53] introduced the code-on-demand style to be able to extend the functionality of the client application, without deploying a new application on the client. However, this is only an optional requirement for the REST architectural style, because not each client may support the necessary runtime environment.

Notice that, HTTP is not mandatory for REST but, of course, can be used as building block for a REST architecture. However, the REST design pattern can be applied also to other technologies without the usage of HTTP. Nevertheless, HTTP or to be more concrete the World Wide Web

architecture is one very successful example of this paradigm. Based on that, the combination of REST and HTTP is very promising and also would allow using the resulting architecture for web mashups.

3 STATE OF SCIENCE AND TECHNOLOGY

In this section, an overview of existing products, techniques, and approaches that may be used, modified, and/or extended to reach the goal of this thesis is given. Whenever suitable, the State-of-the-Art is presented in other chapters of this thesis.

3.1 Web access to OPC UA information models

As already explained in Section 2.4, REST has five mandatory rules and one optional rule. In the following, a closer look is taken if all these rules already can be fulfilled by OPC UA, or if some changes have to be introduced into the OPC UA standard.

Client-Server: As stated out in Part 1 of the OPC UA specification, the basic architecture of OPC UA is based on the client-server pattern.

Cache: Support of caching in the classic web is typically done by special response headers, which allow intermediary servers to determine if the response could be served again for an identical request. However, the typical OPC UA use case is to serve data from field devices. This data often changes on a millisecond base. In OPC UA the client is able to specify a so-called "maxAge" parameter for the *Read* service. A server now is able to serve a cached sensor value to the client, as long as the timestamp of the latest cached value is in the requested client range.

Uniform interface: Each OPC UA *Node* can be seen as a resource. *Node* representations can be constructed in such a way that the manipulation of resources could be done based on the representation. Self-descriptive requests can be achieved by introducing a way to express the kind of message (e.g., based on HTTP headers like the content-type header). The HATEOAS requirement can be achieved through the mapping of OPC UA *References* to the hypermedia concept of links.

Layered system: The typical application domain of OPC UA consists of several layers, for example, ShopFloor-layer, MES-layer, and ERP-layer. Because of that, the layered system architecture is also part of the OPC UA architecture.

Statelessness: The last mandatory requirement of REST cannot be fulfilled with OPC UA version 1.03 and earlier because a client always has to establish a session to access the information model of an OPC UA server. This is even true for services like *Read* or *Browse*. A session in OPC UA is used to store some information, for example, the authorization information and the requested locales. However, it is not enough to just identify all information, which is stored at the client/server during the session set up, it is also necessary to check what kind of additional concepts in OPC UA depend on sessions.

Code-on-demand: Typically an OPC UA server offers interfaces to applications through the OPC UA information model. However, it is also possible to, for example, exchange code snippets as part of the *Value-Attribute* of a *Variable-Node*. Eventually, every architecture that allows exchanging of self-defined content in combination with meaningful meta-data can be used to exchange code snippets. For example, the *Value-Attribute* could contain valid HTML code also including JavaScript code and thus this optional requirement for a RESTful architecture can also be fulfilled by OPC UA.

In the following, different research approaches will be analyzed and evaluated.

RESTful Industrial Communication [63]: The authors introduced a RESTful OPC UA architecture. The architecture covers **Client-Server**, **Cache**, and the **Layered system** through REST patterns very well. Based on the standard *Read* service of OPC UA also the **Code-on-demand** requirement can be fulfilled well but might lack the correct MIME types for an automatic interpretation by a web browser. However, because the authors have not shown any solution for the dynamic *Namespace*- and *ServerArray* problem, it is likely that the **Statelessness** concept is not addressed. In addition, also one important feature of an **Uniform interface**, HATEOAS, is not part of the concept. Finally, Figure 3.1 shows the supported services of the approach.

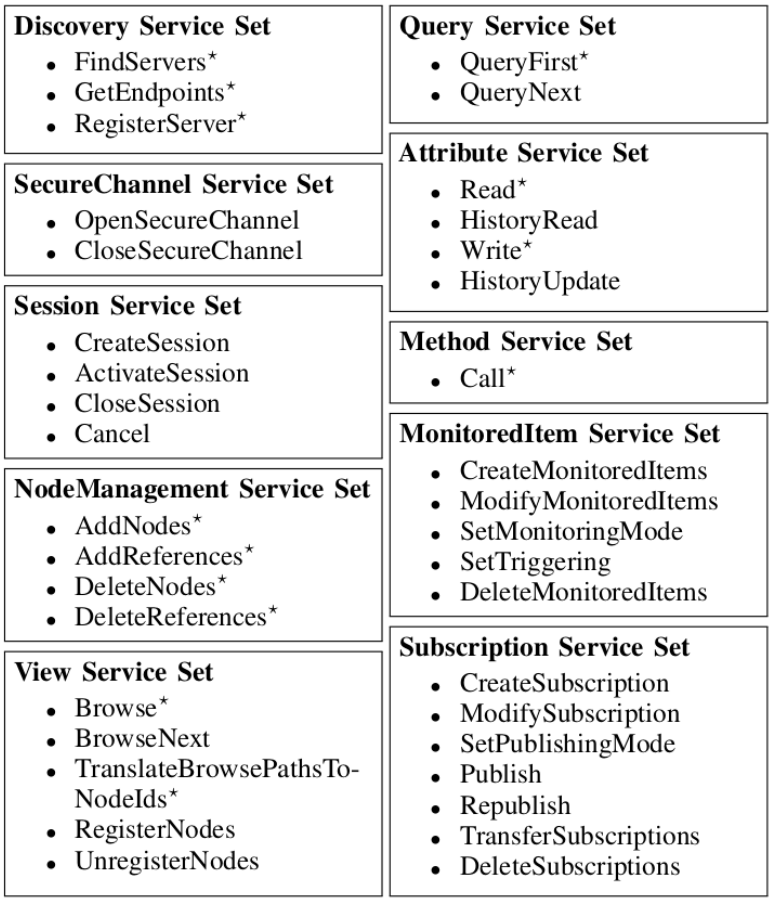


Figure 3.1 – Services are defined in the OPC UA standard. Services that are inherently stateless are marked with a star [63].

OPC UA over CoAP [164]: This draft is also working on an OPC UA REST interface. However, the approach seems to build on the findings of [63], which are already discussed above. Because of that, the same statements should apply.

Protocol interoperability [39]: Also, this proposal is inspired by [63] and therefore the same statements should apply. However, this concept only maps 7 services from OPC UA to a REST API.

HyperUA [123]: HyperUA offers a very nice web-based interface (see also Figure 3.2). The problem of OPC UA, that each service that uses the *Namespace-* and *ServerArray* needs an active session, is solved by still creating sessions and encoding all necessary information into the URLs. While this approach also includes the HATEOAS paradigm and therefore offers a RESTful feeling, finally, each client still has to create a session and therefore the service cannot fully leverage all benefits of REST and also does not support **Statelessness**. Besides that, HyperUA even introduces a concept for RESTful *Subscriptions*. Nevertheless, in this case, the *Subscriptions* are exposed through a proprietary REST interface, which makes it impossible for standard OPC UA session-based as well as session-less clients to access the *Subscriptions*. Eventually, HyperUA covers the **Client-Server**, **Cache**, **Uniform interface**, and **Layered systems** requirements very well. The **Code-on-demand** requirement could be addressed in a similar way than already discussed above.

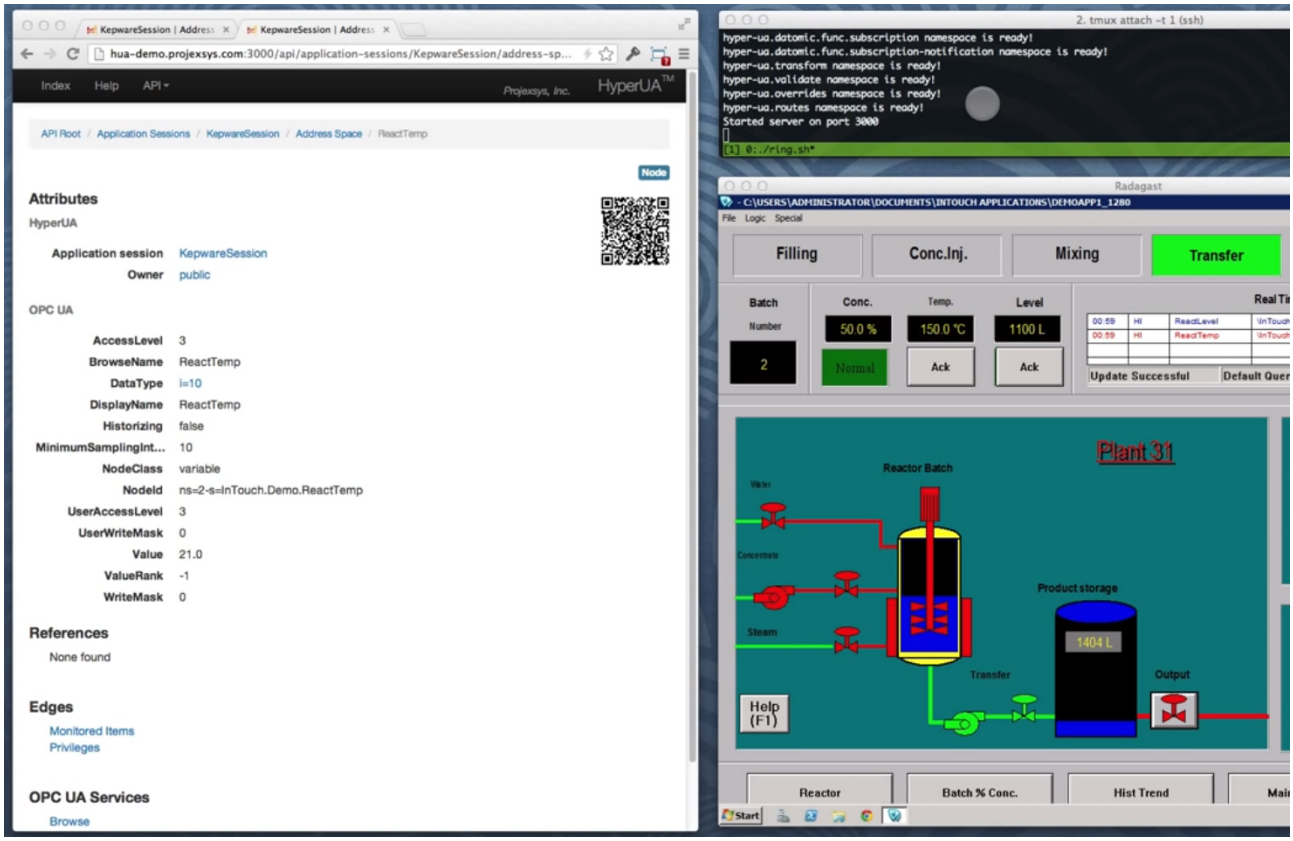


Figure 3.2 – HyperUA Example Server [123].

dataFEED OPC Suite [146]: The dataFEED OPC Suite introduces a so-called REST client API. After taking a closer look at the documentation and at the evaluation version of the software framework, the REST API is identified as some kind of data push API. Basically one can define so-called "Actions", which can be invoked by user-defined conditions and after that send a message to a user-defined REST endpoint. In the end, it can be concluded that the dataFEED OPC Suite REST API has other goals than this thesis.

KEPServerEX [85]: Kepware published an IoT-Gateway plugin for their OPC UA software framework KEPServerEX. This plugin also offers a REST interface to access OPC UA data. The basic concept behind this REST API is three predefined URLs, which allow someone to execute some kind of read, write, and browse service. However, these services only have the name in common with the corresponding OPC UA services and therefore are completely disjoint with the goals of this thesis. For example, the browse service returns all "tags", which are configured for the given REST server interface and not the *References* of OPC UA *Nodes*.

OPC UA PubSub [77]: Since version 1.04 of OPC UA also supports the publish-subscribe communication paradigm (see also OPC UA Part 14), which also can be considered as a form of Group-Subscriptions. Based on the fundamental different communication paradigm the **Client-Server**, **Uniform interface**, **Code-on-demand**, and **Layered system** concept of REST cannot be

covered. In contrast, the **Statelessness** requirement, as well as the **Cache** requirement are covered very well.

Industrial Middleware [61]: This paper discusses a linked data architecture for OPC UA, also leveraging an OPC UA REST API. The concept also uses the HATEOAS concept for interconnecting different *Nodes* and therefore the **Uniform interface** requirement is covered very well. Besides that, the authors seem to use a similar concept as the authors of [63], for introducing REST to OPC UA, and therefore the same statements should apply.

Table 3.1 shows the fulfillment of the above-introduced requirements for the discussed research approaches.

Table 3.2 shows how much of the different OPC UA services are covered by the different research approaches.

In conclusion, Table 3.1 and Table 3.2 show the evaluation results of the different research approaches mentioned above. Based on this evaluation several results can be derived. First, statelessness is not addressed correctly in the actual research and still is an open research challenge. Second, most of the current research struggles with HATEOAS principles (Uniform interface). Third, a lot of OPC UA services cannot be accessed with REST APIs. Finally, it can be derived that a complete REST interface for OPC UA still is an unsolved and open research challenge.

Requirements \ Research approaches	RESTful Ind. Com. [63]	OPC UA over CoAP [164]	Protocol interop. [39]	HyperUA [123]	dataFEED OPC Suite [146]	KEPServerEX [85]	OPC UA PubSub [77]	Industrial Middleware [61]
Client-Server	++	++	++	++	NA	NA	- -	++
Cache	++	++	++	++	NA	NA	++	++
Uniform interface	-	-	-	++	NA	NA	- -	++
Layered system	++	++	++	++	NA	NA	- -	++
Statelessness	- -	- -	- -	- -	NA	NA	++	- -
Code-on-demand	+	+	+	+	NA	NA	- -	+
Sum (18):	12	12	12	**14**	0	0	6	**14**

Table 3.1 – Requirements and evaluation for OPC UA web access.

.

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

Research approaches / OPC UA Services	RESTful Ind. [63]	OPC UA o. CoAP [164]	Protocol inter. [39]	HyperUA [123]	dataFEED OPC [146]	KEPServerEX [85]	OPC UA PubSub [77]	Ind. Middleware [61]
Discovery (3)	X	X		X	NA	NA		X
SecureChannel (2)				X	NA	NA		X
Session (4)					NA	NA		X
CreateSession				X	NA	NA		X
ActivateSession				X	NA	NA		X
CloseSession				X	NA	NA		X
Cancel					NA	NA		X
NodeManagement (4)	X	X	X		NA	NA		X
View (5)					NA	NA		
Browse	X	X	X	X	NA	NA		X
BrowseNext				X	NA	NA		
TranslateBrowsePaths	X	X			NA	NA		
RegisterNode					NA	NA		
UnregisterNode					NA	NA		
Query (2)					NA	NA		
QueryFirst	X	X			NA	NA		
QueryNext					NA	NA		
Attribute (4)					NA	NA		
Read	X	X	X	X	NA	NA		X
HistoryRead					NA	NA		X
Write	X	X	X		NA	NA		X
HistoryUpdate					NA	NA		
Method (1)	X	X			NA	NA		X
MonitoredItem (4)					NA	NA		
CreateMonitoredItems				X	NA	NA		
SetMonitoringMode				X	NA	NA		
SetTriggering					NA	NA		
DeleteMonitoredItems				X	NA	NA		
Subscription (5)					NA	NA		
CreateSubscription				X	NA	NA		
ModifySubscription				X	NA	NA		
Publish					NA	NA		
TransferSubscriptions					NA	NA		
DeleteSubscriptions				X	NA	NA		
Sum (34):	13	13	7	17	0	0	0	**18**

Table 3.2 – Coverage evaluation of OPC UA services for web access.
Legend: X = mapped (1), " " = not mapped (0), NA = Not Applicable (0)

3.2 Semantics in OPC UA information models

OPC UA offers the capability to store very rich and standardized semantics in form of OPC UA information models. However, for the use cases analytics, validation, and query of OPC UA data models a mapping to the OWL/RDF(S) ecosystem would introduce several benefits, as it would allow applying relevant existing tools from this ecosystem, rather than having to re-invent them in the OPC UA world. The mapping in Chapter 5 aims to be the first such comprehensive mapping from OPC UA to OWL.

For an understanding of the mapping it is necessary to understand that OPC UA information models are built in a modular way (see Figure 3.3), where the OPC UA meta-model (Meta-Layer) provides the basic building blocks for information models, continuing with the OPC UA core information model (Base-Layer), which is provided by the OPC Foundation itself, followed by OPC UA companions (Companion-Layer), OEM-specific schema extensions (Extension-Layer) of OPC UA companions, and finally OPC UA instance models (Instance-Layer) that describe configuration and data items of individual devices based on schemas of the Base-Layer, Companion-Layer, and Extension-Layer. The mapping transforms OPC UA information models by translation of the modules (levels Base-Layer - Instance-Layer described above) into RDF/OWL ontologies that

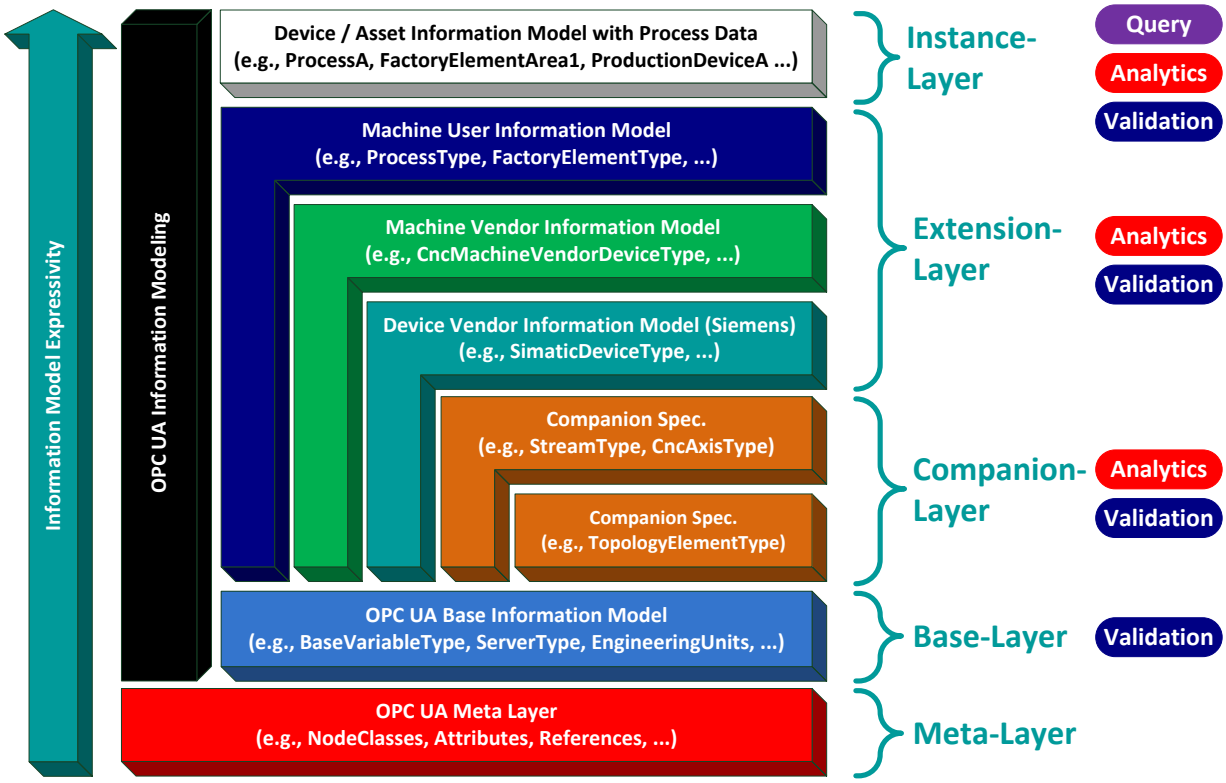


Figure 3.3 – Overview of OPC UA Information Modeling [136].

import each other in the same way the respective OPC UA modules do. Once this transformation is performed, the resulting RDF/OWL ontologies can be used for the purpose of validation, querying, and analytics. Figure 3.3 also depicts the typical OPC UA information model layers for each use case.

One essential requirement to map OPC UA to OWL is that OPC UA constraints and semantics (implicit and explicit) are transformed correctly to a formal ontology language like OWL. Only if the concepts match on both sides, all benefits of Semantic Web technology can be unleashed and used to solve the identified use cases. In the following, a short analysis of OPC UA semantics is presented, revealing some of the issues which are preventing a trivial mapping to OWL.

Figure 3.4 shows an example information model of OPC UA based on [162]. The left side of the picture contains a small fracture of a typical OPC UA *TypeModel*, which is provided by a *Companion Specification*. Several *Type-Nodes* are defined, for example, a "CncChannelType", a "CncAxisType" and also a "DataItemType". These *Types* are defined with special *NodeClasses* in OPC UA (e.g., *ObjectType-Node* and *VariableType-Node*). Each *Instance-Node* (Figure 3.4 right side) *References* its *TypeDefinition* with a special *ReferenceType* named "HasTypeDefinition". While *Type-Nodes* (e.g., *VariableType*, *ObjectType*, etc.) can be subtyped and therefore inherit semantics and constraints from the supertype, the same is not true for *Instance-Nodes* (e.g., *Variable*, *Object*).

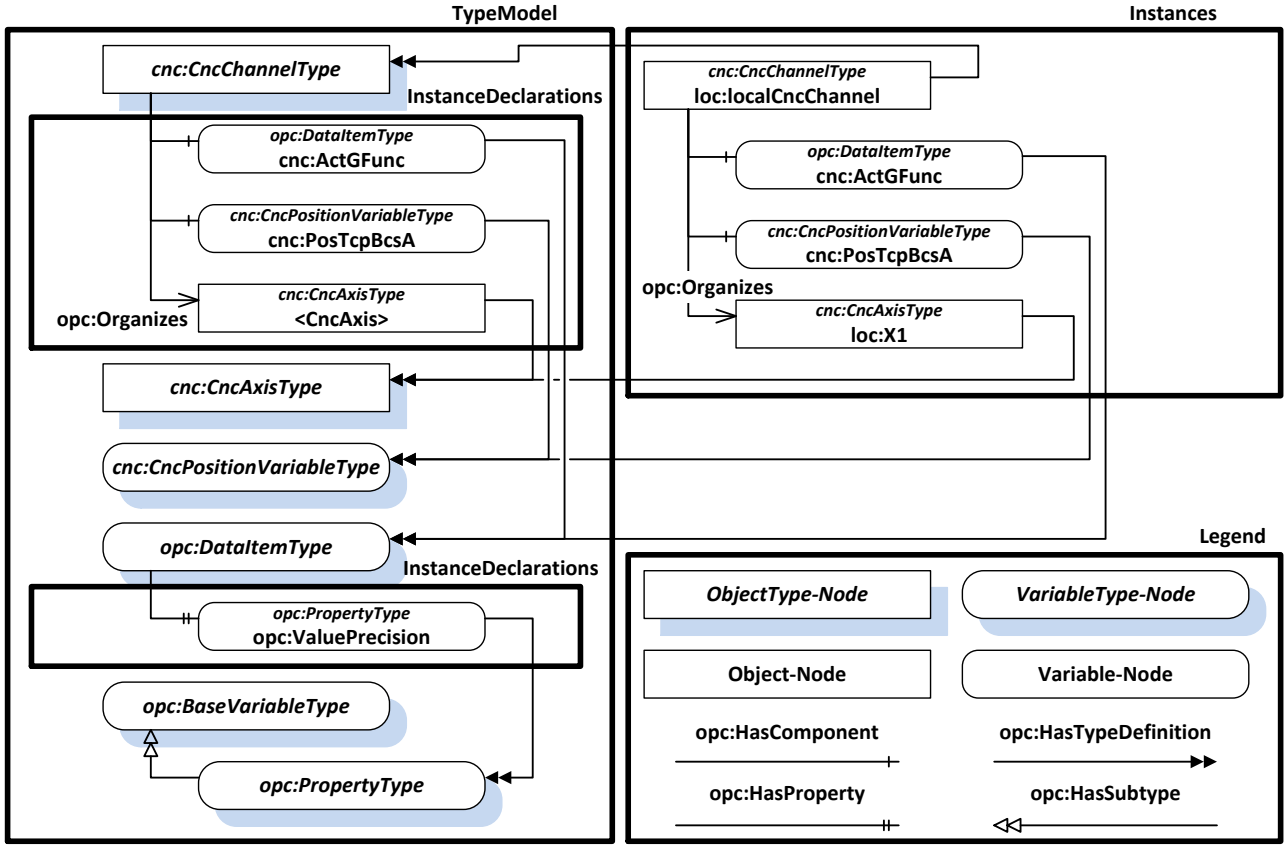


Figure 3.4 – An example OPC UA information model for CNC machines [136].

However, surprisingly also *Instance-Nodes* can be part of an OPC UA *TypeModel*. These special *Instance-Nodes* are also called *InstanceDeclarations* within OPC UA. An *InstanceDeclaration* is a *Node* that is defined in the context of a *Type-Node* and is used to model the sub-structure of a *Type*. The sub-structure of the CncChannelType is depicted in Table 3.3. Each *InstanceDeclaration* is defined by several characteristics. (1) The *ReferenceType* interconnects the defining *Type-Node* with the *InstanceDeclaration*. It is also allowed to use a subtype of the concrete *ReferenceType*. (2) The expected *NodeClass* and *BrowseName* (a *BrowseName* is a string with a *NamespaceURI* assigned to it, for example, "http://opcfoundation.org/UA/CNC/" or in short "cnc:", as *NamespaceURI* and "CncChannelType" as string part), which must be identical on each *Instance-Node*. (3) The *DataType* (if applicable) and *TypeDefinition* (if applicable). Also in this case subtypes are allowed. Finally, the corresponding *ModellingRule* is assigned. OPC UA defines several *ModellingRules* and also allows to define new *ModellingRules*, if necessary. The *Mandatory-ModellingRule*, for example, enforces, that each *Instance-Node* of the CncChannelType must *Reference* a *Node* similar to the corresponding *InstanceDeclaration*. Similar in this case means, the same *NodeClass* and *BrowseName* combined with the same *DataType* and *TypeDefinition* or a subtype (if applicable), *Referenced* by the defined *ReferenceType* or a subtype of it. Similarly, there are *Optional-ModellingRules*, stating that *InstanceDeclarations* are not compulsory on the instance level. The *Placeholder-ModellingRule* is used if the *BrowseName* of the *Instance-Node* is not defined within the *Companion Specification* and can be freely chosen for *Instance-Nodes* (only in combination with *Variables* or *Objects*).

A typical *Companion Specification* describes the semantics of these *InstanceDeclarations* in textual form and in machine-readable form (OPC UA *NodeSet*). In the case of the ActGFunc *InstanceDeclaration* from CncChannelType this is done like [162]: **ActGFunc:** "Array of active G functions; there can be several G functions active at a time (modal and non-modal G functions).";

As shown above, the semantics of such *Companion Specifications* is very rich and can be used for use cases like monitoring applications, which are able to find the necessary data points automatically based on standardized semantics. However, as also depicted in Figure 3.4, OPC UA has some

Table 3.3 – CncChannelType definition (see also [162]).

Attribute	**Value**			
BrowseName	CncChannelType			
IsAbstract	False			
References	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the CncComponentType				
HasComponent	ActGFunc	String[]	DataItemType	Mandatory
HasComponent	PosTcpBcsA	CncPosDT	CncPosVarT	Mandatory
...	...	...	...	...
Organizes	<CncAxis>		CncAxisType	OptionalPlaceholder
NOTE: This row represents no Node in the AddressSpace. It is a placeholder pointing out that instances of the ObjectType will have those Objects.				

special modeling practices. For example, just consider the fact that the semantics of the ActGFunc *Instance-Node* (right side of Figure 3.4) is defined by the *InstanceDeclaration* (left side of Figure 3.4), but the *Instance-Node* specifies the DataItemType as its *Type* and not the *InstanceDeclaration*. While the connections between *InstanceDeclarations* and *Instance-Nodes* are pretty obvious for each OPC UA expert, a typical Semantic Web expert would probably not have guessed these implicit connections. Note that, there is no other direct *Reference* between an *InstanceDeclaration* and an *Instance-Node*, the connection is implicitly made through the identical *BrowseName*. Of course, OPC UA defines further rules to ensure that this concept always can be applied. For example, it is forbidden to define two identical *BrowseNames* for *InstanceDeclarations* in the context of the same *Type*. In contrast, a Semantic Web expert would have probably expected a new *VariableType*, which is a subtype of DataItemType and is *Referenced* by the *HasTypeDefinition-ReferenceType* of an *Instance-Node*. Exactly such modeling artifacts have prevented a trivial mapping from OPC UA to a formal language like OWL till now because, for each OPC UA design pattern, the corresponding concept and transformation rule must be identified.

Besides the concepts which are mentioned above, several additional main concepts to express semantics in OPC UA can be identified:

ReferenceTypes are used to interconnect *Nodes* with each other. There are several different *ReferenceTypes* already defined for OPC UA, but it is also allowed to define new *ReferenceTypes*. As all other *Type-Nodes ReferenceTypes* are organized in a subtype hierarchy with the inheritance of semantics and constraints.

DataTypes in OPC UA are used to define semantics and constraints for the *Value-Attribute* of *Variable-* and *VariableType-Nodes*. Like *ReferenceTypes*, *DataTypes* are organized in a subtype hierarchy, which can be used to refine semantics and to add additional constraints. For example, the *ByteString-DataType* can be used as *DataType* for representing images. However, to add further semantics the OPC Foundation subtyped this *DataType* with an *Image-DataType*. Of course, there are plenty of different formats for images and also constraints that correspond to these formats (e.g., meta information, encoding, etc.). Because of that, the OPC Foundation subtyped the *Image-DataType* again and added further *DataTypes* like *ImageBMP-DataType* and *ImageJPG-DataType*. Besides the normal use case of *DataTypes* described above, there exists a special *DataType* in OPC UA, which is called *Structure-DataType*. This *DataType* can be used to combine several native *DataTypes* into one single *DataType*. For example, a range can be described by two values. The high-value for the upper limit of the range and the low-value for the lower limit of the range. In some applications, it is crucial that both values can be read in one single transaction context, which can only be reached in OPC UA by fetching a single *Value-Attribute* of a *Node*. However, in other applications, this is not important and therefore the OPC Foundation introduced the notion of *ComplexVariables*. *ComplexVariables* in OPC UA are based on *Structure-DataTypes*, which also define the semantics of the fields (e.g., high-value and low-value). Single values (e.g., high-value) can also

be exposed by additional *Variable-Nodes*, which only offer a single field of the *Structure-DataType*. The connection between structure fields and the corresponding *Variable-Nodes* is done again based on *BrowseNames*, similar to the *InstanceDeclaration* concept.

Attributes: There are several *Attributes* in OPC UA which can be considered semantic meaningful like the *Symmetric-Attribute* of *ReferenceType-Nodes*.

Requirements \ Research approaches	Industrial Middleware [61]	OWL Ontologies in OPC UA [31]	OPC UA methods semantics [82]	OPC UA Reasoning [17]	OPC UA and Semantic Web [98]	Ontology-Based OPC UA [150]	OPC UA NodeSet Ontologies [119]
Namespaces	- -	-	- -	+	++	++	++
Attributes	-	- -	-	-	-	-	+
DataType	-	- -	- -	-	- -	-	- -
ReferenceType	+	- -	- -	+	-	+	++
ObjectType	+	+	- -	+	+	+	-
VariableType	+	+	- -	+	+	+	-
InstanceDeclaration	- -	- -	- -	- -	- -	- -	- -
MethodInstanceDecl	- -	- -	++	- -	- -	- -	- -
Instances (Object, Variable, ...)	-	+	-	+	+	+	+
Sum (27):	9	7	5	12	11	**13**	12

Table 3.4 – Requirements and evaluation for OPC UA semantics.

.

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

Table 3.4 shows the requirements for a full OPC UA semantics mapping. The **Namespaces** requirement is used to identify if the layering of OPC UA information models is covered and if the transformation can be done automatically. The **Attributes** requirement represents if all semantic concepts of OPC UA *Attributes* (like the *Symmetric-Attribute*) are transformed correctly to

the corresponding semantic concepts within the Semantic Web. The **DataType**, **ReferenceType**, **ObjectType**, and **VariableType** requirements represent how well these OPC UA concepts and the corresponding semantics (like type hierarchy, inverse relations, constraints, etc.) are mapped to the Semantic Web. The concept behind the **InstanceDeclaration** and **MethodInstanceDeclaration** requirement is explained in great detail above and shows if the semantics are identified and captured correctly. Finally, the **Instances** requirement (*InstanceDeclarations* are not included) indicates if the semantics of *Instance-Nodes* is captured and transformed correctly to the Semantic Web.

Industrial Middleware [61]: The paper describes a special OWL ontology for OPC UA. The authors seem to define a new manual ontology for OPC UA and because of that the **Namespace** requirement seems to be not covered. The **Attribute** and **DataType** requirement is covered high-level but the paper does not provide details for concepts like domain, range, type hierarchy, mapping to XML data types, structures, arrays, enumerations, etc. The **ReferenceType**, **ObjectType** as well as the **VariableType** constraint seems to be covered well but no details on certain aspects like inverse, type hierarchy, domain, range, OPC UA *Properties*, and cardinality are given. Concepts for **InstanceDeclaration** and **MethodInstanceDeclaration** are not discussed at all in the paper. **Instances** are modeled by the authors. However, no concepts for the correct typing (e.g., an *Instance* is of type "opcua:Node") nor any constraints like mandatory and optional *Attributes* are present in the publication.

Integrating OWL Ontologies in AML and OPC UA [31]: Bunte et. al. present an interesting approach with a focus on the reversed mapping direction from OWL to OPC UA (see also Figure 3.5). Based on the published material it can be assumed that it is possible to translate OWL namespaces to OPC UA and vice versa. However, the introduced way seems to be not compatible to translate existing OPC UA *Namespaces* from *Companion Specifications* correctly to the namespace concept of OWL. **Attributes**, **DataTypes**, **ReferenceTypes**, as well as **InstanceDeclarations** and **MethodInstanceDelcarations** are not discussed at all by the authors. The **ObjectType**, **VariableType**, and **Instance** concept is used heavily to map all OWL features to an OPC UA information model. However, also in this case the mappings only should be able to correctly translate OPC UA information models to OWL if they are modeled exactly as proposed by the authors, which is not true for all OPC UA information models.

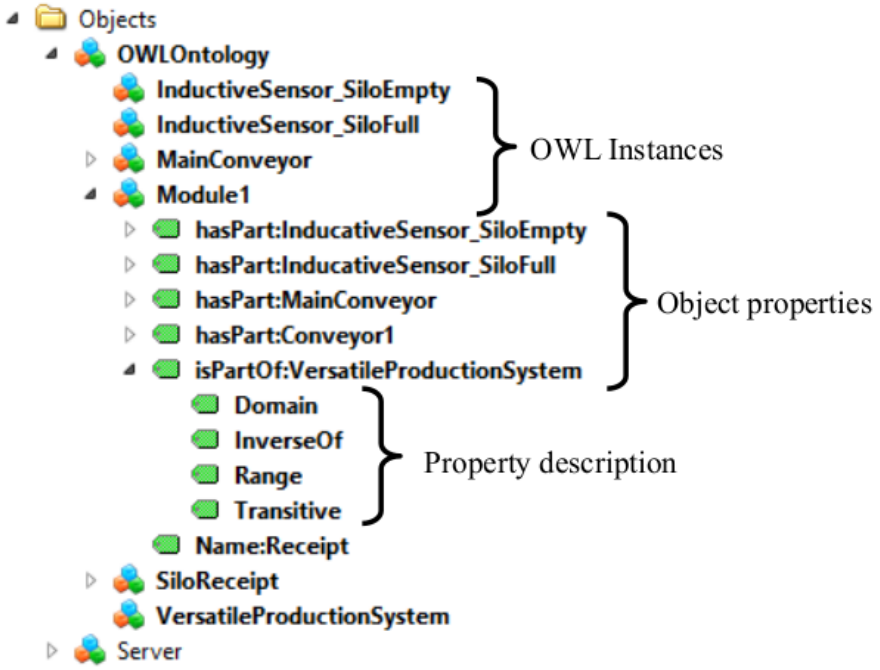
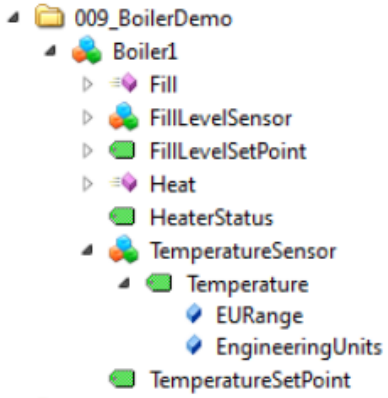


Figure 3.5 – Transformed OPC UA information model from an ontology [31].

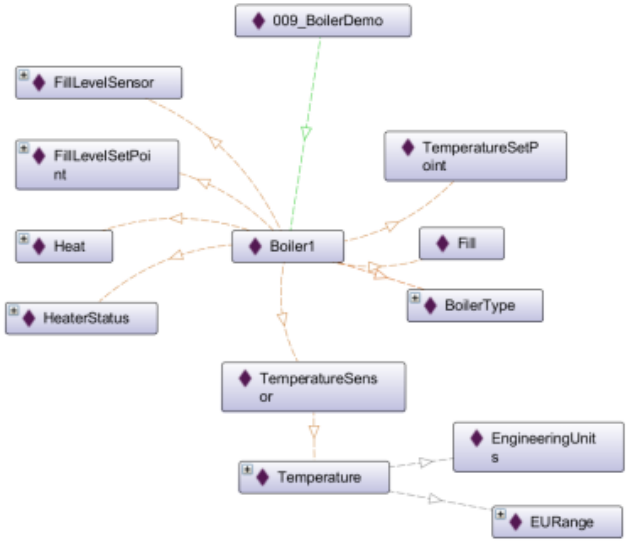
Semantics for OPC UA methods [82]: The authors developed a concept of how OWL-S[117] can be introduced to OPC UA *Methods*. Regarding the *MethodInstanceDeclaration* concept the proposed concept not only should be able to cover all features of the identified requirement, moreover, the authors also found a way to introduce formalized concepts for pre- and postconditions of methods. Nevertheless, because the publication focuses exclusively on OPC UA *Methods* other requirements that cannot be covered at all like **Namespaces**, **DataType**, **ReferenceType**, **ObjectType**, **VariableType**, **InstanceDeclaration**. The **Attributes** and **Instances** requirement is only covered partly.

OPC UA Reasoning [17]: Within this paper, the authors focus on an interesting transformation of the OPC UA standard *Namespace* "http://opcfoundation.org/UA/" to an OWL ontology to show some of the benefits of Semantic Web technology in several use cases like adaptable factories, energy management, and human-machine-interfaces (see also Figure 3.6). The proposed mapping of the standard OPC UA *Namespace* seems to be automatically generated. However, there seems to be no concept for automatic import of other *Namespaces* like *Companion Specifications*. Table 3.5 shows the published transformation rules for the proposed approach. The **Attributes** and **DataType** requirement is only covered partly and does not provide details for concepts like domain, range, type hierarchy, mapping to XML data types, structures, arrays, enumerations, etc. The **ReferenceType** is mapped well but without type hierarchy and constraints. **ObjectTypes** and **VariableTypes** are mapped to a type hierarchy also in OWL but it seems that no constraints are mapped like the mandatory and optional modeling rules as well as some *PropertyType* specific constraints. The semantics behind **InstanceDeclarations** and **MethodInstanceDeclarations** is

also not covered by the published content of the authors. Finally, **Instances** are typed correctly but also seem to be modeled without constraints (e.g., *Attribute* constraints for each *NodeClass*).



(a)



(b)

```
PREFIX opcua:            <http://opcfoundation.org/UA/>
PREFIX rdf:              <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:             <http://www.w3.org/2000/01/rdf-schema#>
PREFIX opcuauserns1:     <urn:eclipse:milo:hello-world>
ASK {
    ?boiler opcua:N40 ?boilertype .
    ?boilertype   rdfs:label "BoilerType" .
    ?y opcua:N47 ?temp_sensor .
    ?temp_sensor opcua:N40 ?temp_sensor_type .
    ?temp_sensor_type   rdfs:label "TemperatureSensorType"
  }
```

(c)

Figure 3.6 – Example of an OPC UA information model (a) transformed into OWL DL (b) and a SPARQL query asserting that all objects of type “BoilerType” have at least one temperature sensor [17].

OPC UA Concept	OWL Concept
ObjectType, EventType Class HasSubType	SubClassOf
HasProperty	RDF Property
HasTypeDefinition	RDF Type
HasComponent	RDF Member
Object, Method, Reference, View	Individual
Attribute “NodeId”	Extended RDF resource uri constructed with the namespace uri.
Attribute ”Datatype”	Datatype or Class
Attribute “Value”	DataHasValue
Attribute “Description”	Literal
...	...

Table 3.5 – Excerpt of the transformation rules used to map OPC UA concepts into OWL concepts [17].

OPC UA and the Semantic Web [98]: Majumder et. al. present a comparison between OPC UA and the Semantic Web represented through RDF, RDFS, and OWL. The authors compare several aspects of OPC UA with the Semantic Web like *Views* and *Methods* but in most cases do not provide any details on a possible mapping. However, as also stated by the authors of [98], the paper is only a first preliminary step in the journey for providing such a mapping. Table 3.6 shows the published mapping elements of the authors. **Namespaces** of OPC UA are also mapped to namespaces in OWL and *Companion Specifications* are also covered through the approach. **Attributes** and **ReferenceTypes** are correctly mapped to the corresponding concepts in OWL but the paper does not provide details for concepts like domain, range, type hierarchy, mapping to XML data types, structures, arrays, enumerations, etc. **ObjectTypes** and **VariableTypes** are mapped to classes and also the type hierarchy should be covered but also in this case no constraints (e.g., mandatory and optional *ModellingRules*) are addressed. The concept behind **DataTypes**, **InstanceDeclarations**, and **MethodInstanceDeclarations** is not discussed within the publication. Finally, **Instances** are mapped to OWL individuals without constraints.

OPC UA	Semantic Web
OPC UA	Semantic Web
ObjectTypes, VariableTypes	owl:class
Instance of a type	OWL Individual
HasTypeDefinition	rdf:type
ReferenceTypes	owl:objectproperty
Node attributes	owl:datatypeproperty
HasSubtype	owl:subClassOf
NodeID	URI
query service	SPARQL

Table 3.6 – Mapping Between OPC UA and Semantic Web modelling elements [98].

Ontology-Based OPC UA [150]: Steindl et. al. presents another interesting approach to convert OPC UA information models into OWL ontologies. Within the publication the following transformation rules are presented [150]:

- An rdfs:label is created for each resource from the OPC UA node's DisplayName.
- All resources corresponding to an OPC UA node of node class ObjectType, VariableType, or DataType are declared as owl:Class.
- All resources corresponding to an OPC UA node of node class ReferenceType are declared as owl:ObjectProperty.
- All resources corresponding to an OPC UA node being the source of a HasTypeDefinition reference are declared as individual of the corresponding class.
- All resources corresponding to an OPC UA node being the source of a HasSubtype reference are declared as superclass of the corresponding class.
- All object properties, for which the corresponding OPC UA ReferenceType's attribute Symmetric is set, are declared as symmetric property.
- For all object properties, for which the corresponding OPC UA ReferenceType's attribute InverseName is set, an inverse object property is created.
- All properties relating OPC UA nodes to their OPC UA attributes are declared as annotation properties.

Based on the published information it can be assumed that **Namespaces** are correctly mapped to OWL namespaces and that *Companion Specifications* can also be translated automatically. **Attributes** and **DataTypes** are mapped partly without details for concepts like domain, range, mapping to XML data types, structure constraints, arrays, enumeration constraints, etc. **ReferenceTypes**, **ObjectTypes**, and **VariableTypes** are mapped well but without constraints. Furthermore, the type hierarchy of *ReferenceTypes* is only modeled in the OWL class concept. The publication does not discuss **InstanceDeclarations** and **MethodInstanceDeclarations**. Eventually, **Instances** are mapped also correctly to OWL individuals without constraints.

OPC UA NodeSet Ontologies [119]: The authors of this paper present OWL ontologies of OPC UA information models for the integration of OPC UA semantics into semantic digital twins of manufacturing resources. Furthermore, the authors describe a concept to further enrich these ontologies with, for example, device and component skills, capability models, and geometry models. **Attributes** are mapped well including range and domain constraints. **DataTypes** are mapped partly but the paper does not provide details for concepts like type hierarchy, structures, arrays, enumerations, etc. **ReferenceTypes** are mapped very well including type hierarchies and characteristics like symmetric and inverseOf but no domain and range constraints. In contrast to the very expressive **ReferenceType** mapping, the **ObjectType** and **VariableType** mapping is quite simplistic as depicted in Figure 3.7. Neither types nor type hierarchies are covered but basic constraints (e.g., *Attribute* constraints for *NodeClasses*) are modeled. The publication does not present concepts for **InstanceDeclarations** and **MethodInstanceDeclarations**. Finally, **Instances** are mapped well and even cover some basic *Attribute* constraints but based on the missing type concept do not cover *ModellingRule* constraints of OPC UA types.

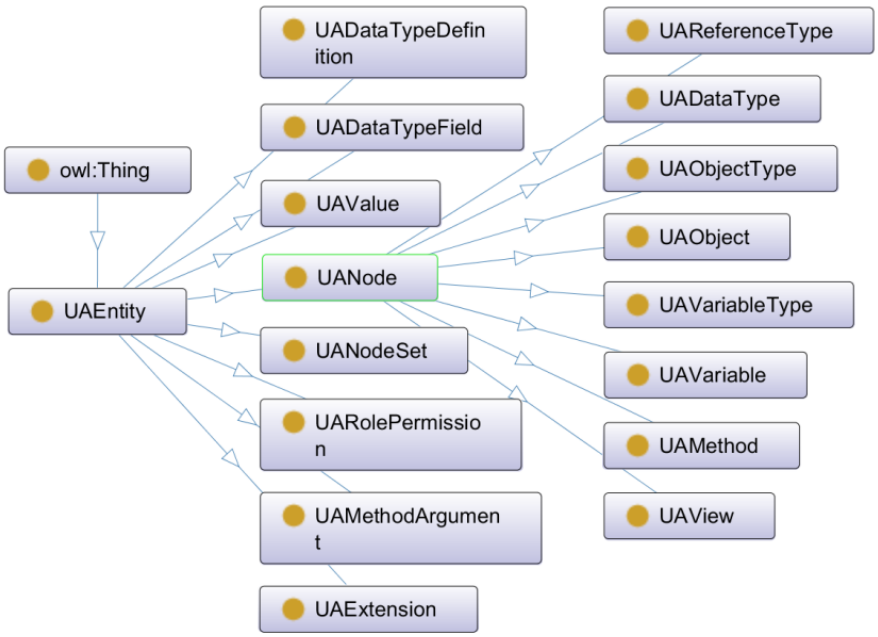


Figure 3.7 – Upper taxonomy of the OPC UA core ontology [119].

In conclusion, Table 3.4 shows the evaluation results of the different research approaches mentioned above. Based on this evaluation several results can be derived. First, the concept behind *InstanceDeclarations* is not covered at all in actual research. Second, the modeling of OPC UA constraints is only covered in parts by some researchers. Third, a complete translation of all OPC UA semantics into the Semantic Web is not presented in one single concept. Eventually, this leads to an open research challenge for the translation of OPC UA semantics to the Semantic Web.

3.3 Querying of OPC UA information models

Currently, a lot of activity around OPC UA information modeling can be observed. These activities typically center around the enrichment of the OPC UA vocabulary, with the goal to describe machines on a very detailed level. As more terms become available also the number of *Nodes* per information model is raising. Furthermore, OPC UA becomes more and more popular on the aggregating layers like edge and cloud. With this in mind, it can be assumed that sooner or later the automation domain is faced with huge (standardized) OPC UA information models with very detailed descriptions of the underlying physical devices. This introduces big opportunities for a lot of use cases like analytics and human-machine interfaces (HMI), which can be programmed against standardized information models, enabling the deployment on each machine independent of the manufacturer without additional engineering effort. However, one important part to use such OPC UA information models is still missing. Without some kind of query functionality it is soon impossible to find the necessary data points on the aggregating layers like edge and cloud to bind them to the apps (e.g., a predictive maintenance app for engines, which of course needs some values from the engine itself, like temperature/power/...). It is worth noting that OPC UA offers a query language for searching OPC UA information models, but up to now, there is no publicly available implementation, as far as I know. Of course, it is not a practical solution to search the graph node by node for each application (on cloud level ten-thousands of *Nodes* would have to be searched by hundreds of apps in parallel). Another problem is that the OPC UA-specific query language is so complex that some industry researchers even introduced an internal domain-specific language for constructing OPC UA *Queries* [59].

As motivated above, some form of query capability is crucial for the further success of OPC UA as Industrial Internet of Things protocol. Table 3.7 shows the main requirements for an useful OPC UA query API based on OPC UA *Query* (see Section 2.1.3). The **View** row is based on the view parameter of OPC UA *Query* in a similar way the **NodeTypeDescription** row is based on the nodeTypes[] parameter and the **Filter** row is based on the filter parameter. The **Limits** row refers to the concepts behind the maxDataSets and maxReferences parameter of OPC UA *Query*. Finally,

the **Part 4 Annex B** row represents how many of the example queries of OPC UA Part 4 Annex B Complex Examples can be correctly executed through the query API.

Industrial Middleware [61]: The paper mainly focuses on a Linked Data architecture and gives insights into the mapping details. The authors also mention that it is possible to query the so-called "LD cloud" with SPARQL. However, within the published content no concepts are presented for **Views**, **Filters**, **Limits**, and **Part 4 Annex B** requirements. The *NodeTypeDescription* requirements can be implicitly extracted out of the paper but also seems to not be covered completely (e.g., include subtypes feature is not mentioned in the publication).

OWL Ontologies in OPC UA [31]: Within the publication, a query feature is not mentioned at all. Nevertheless, based on the presented transformation rules into the Semantic Web it should be possible to cover certain aspects of the *NodeTypeDescription* requirement.

Semantics for OPC UA methods [82]: The authors developed a concept of how OWL-S[117] can be introduced to OPC UA *Methods*. Based on the fact that only one OPC UA concept is covered it should only be possible to cover parts of the *NodeTypeDescription* requirement.

OPC UA Reasoning [17]: This paper also discusses a concept to query OPC UA information models. Nevertheless, the focus of the presented query features seems to be quite different because no concepts are presented to address the **View**, **Filter**, **Limits**, and **Part 4 Annex B**

Requirements \ Research approaches	Industrial Middleware [61]	OWL Ontologies in OPC UA [31]	OPC UA methods semantics [82]	OPC UA Reasoning [17]	OPC UA and Semantic Web [98]	Ontology-Based OPC UA [150]	OPC UA NodeSet Ontologies [119]
View	NA	NA	NA	NA	-	NA	NA
NodeTypeDescription	+	+	NA	+	+	+	+
Filter	NA	NA	NA	NA	NA	NA	NA
Limits	NA	NA	NA	NA	NA	NA	NA
Part 4 Annex B	NA	NA	NA	NA	NA	NA	NA
Sum (15):	2	2	0	2	**3**	2	2

Table 3.7 – Requirements and evaluation for OPC UA *Query*.

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

requirements. In the end, the proposed query feature should be able to cover large fractions of **NodeTypeDescription** requirement but a full coverage analysis is not possible based on the published material.

OPC UA and the Semantic Web [98] also present first ideas on how SPARQL could be leveraged to address certain aspects of an OPC UA query API like the **View** requirement. Furthermore, the presented mapping between OPC UA and the Semantic Web should allow covering aspects of the **NodeTypeRestriction** requirement. In contrast, the **Filter**, **Limits**, and **Part 4 Annex B** requirements are not discussed in this publication.

Ontology-Based OPC UA Data Access [150]: Steindl et. al. present an interesting mapping from OPC UA to the Semantic Web and also show how the resulting ontology can be queried with SPARQL. Based on the published content the **NodeTypeDescription** parameter is covered well and only small things like the *ReferenceType* hierarchy in form of object properties is missing to cover all functions of the **NodeTypeDescription**. Furthermore, the authors present a very interesting concept to query for historical data, which could be leveraged to provide a generic concept for the *History Services* of OPC UA. The **View**, **Filter**, **Limit**, and **Part 4 Annex B** requirements are not covered within this publication.

OPC UA NodeSet Ontologies [119]: The paper shows an approach how OPC UA information models can be transformed into ontologies and also gives some insights into querying these ontologies. Based on the assessment of the published material it seems that the **NodeTypeDescription** requirements can be covered well but full coverage is not shown. The **View**, **Filter**, **Limit**, and **Part 4 Annex B** requirements are not covered within this publication.

In conclusion, Table 3.7 shows the evaluation results of the different research approaches mentioned above. Based on this evaluation several results can be derived. First, certain features of an OPC UA query language are not addressed at all like the **Filter**, **Limits**, and **Part 4 Annex B** requirements or covered only in parts like the **View** requirement. Second, features like the *NodeTypeDescription* can be covered well indirectly through the presented Semantic Web mappings. However, also this feature is not covered completely by any of the presented research approaches mainly due to the fact that a full-featured OPC UA query API is not the main goal for the presented approaches. Finally, it can be argued that a full featured OPC UA query API still is an unsolved and open research challenge. ´

4 WEB ACCESS TO OPC UA INFORMATION MODELS

While OPC UA is very well suited as interoperability technology in the manufacturing domain, up to now it can be considered only a niche technology in most other domains. In contrast, a technology that is already present in nearly every domain is the REST architecture [46]. REST is derived from the classic web, which is already connected to each domain in one way or another and can be used to provide larger interoperability on the transport layer across different domains for OPC UA. The basic idea of this work is to extend OPC UA with REST capabilities to finally reach the status of an Internet-of-Things protocol. A lot of research was already done on this topic [63, 164, 61, 39] but some challenges like the introduction of statelessness into OPC UA are still not addressed in the research community. Parts of this chapter are also published in [137, 138].

The remainder of this chapter is structured as follows:

Section 4.1 introduces the basic design decisions behind the web architecture. Furthermore, the section gives an overview of the different services of OPC UA and how these services are mapped.

Section 4.2 summarizes the standardization contributions of this thesis, starting with an analysis of OPC UA sessions and followed by details of the standardization proposal, which is now part of the OPC UA specification since version 1.04.

Section 4.3 focuses on the mapping of services to the OPC UA information model. First, the *Discovery Service Set* is discussed in greater detail. Second, a concept for RESTful batch requests is presented.

Section 4.4 exemplifies how OPC UA can be mapped fully to HTTP. This includes the usage of different HTTP verbs, additional HTTP headers, the usage of HTTP result codes, the URI design, and the definition of resource representations.

Section 4.5 provides insights into how OPC UA *Subscriptions* are mapped to REST in an efficient way.

Section 4.6 gives an overview of REST design patterns, which can be used to add new features to OPC UA like web browser support.

Section 4.7 introduces the prototypical implementation of the REST binding through examples.

Section 4.8 evaluates the findings and contributions of this chapter against the corresponding research challenge. Furthermore, the section outlines further ideas and thoughts about the web and OPC UA.

4.1 Overall Architecture

This section is structured into two main sections. In the beginning, the design decisions are presented in greater detail (Section 4.1.1) followed by an overview of the mapped services (Section 4.1.2). The mapped services consist of newly introduced services to enhance the REST experience of OPC UA, modified services to offer these services in RESTful environments, and unmodified services that can be directly translated into the proposed REST design pattern. The mapping can further be categorized into two main groups, HTTP based service implementations, and OPC UA information model based implementations.

4.1.1 Design Decisions

In the following section, some insights into the architecture design goals, and the reasons why they should be achieved are given.

Efficient SessionlessInvoke: As already discussed, OPC UA services which depend on *NodeIds* and so on the *Namespace*- and *ServerArray*, need some further investigation. One goal here is to introduce some new efficient service for session-less clients.

Uniform HTTP interface: For this approach, it is decided to use HTTP as the underlying protocol, mainly because of huge client support (web browser/web server). To ensure compliance with the REST rules, OPC UA has to be mapped to the uniform HTTP interface.

Batch support: Most OPC UA service sets already implement batch support. This feature significantly reduces the overhead for clients and servers, if more than one value should be fetched (the typical industry use case). Based on several expert discussions this feature is considered very important for the industry domain to ensure the required request volume and because of that also is one of the design goals of this thesis.

URI templates [49] are used to offer the client a recipe for URI construction. For example, the URI template "https://host/{userName}{?knows}" could be used to construct the following URI: "https://host/user?knows=sam", where the "userName" is "user" and the value for "knows" is "sam". Besides the usefulness of such a concept for RESTful services, there is a lot of discussions out there whether URI templates should be specified within some kind of document, or should only

be discoverable during runtime [109]. However, there are several reasons, why the harmonization of URI templates across OPC UA servers has more benefits than drawbacks. For example, most actual OPC UA applications know exactly the *NodeIds* which they want to fetch from the OPC UA server (e.g., the well-known *NamespaceArray*). These *NodeIds* are sometimes deeply nested in the information model. The classical hypermedia approach, with only a single well-known entry-point, would require that several "Browse"-Calls had to be made, which would yield a lot of overhead and can be considered one of the major drawbacks of the HATEOAS approach [57]. Another reason is, that a typical *ExpandedNodeId* OPC UA structure, which is the only way to link to server external *Nodes*, could not be used as a link relation to another server if a common URI template is not specified. Because of that, the benefits of harmonizing URI templates across servers outweigh the drawbacks, in the case of OPC UA, and are also applied in the same way by a lot of very popular web services like Amazon's S3 [5], Google's Gmail [60], Apple's News API [13], and the Twitter API [155].

Browser support: One of the greatest benefits of the classic web is, that it is not necessary to install a special application for each web-application any longer. Instead, only one application must be installed, also known as a web browser, and can be used for a lot of different applications like webmail, online banking, shopping, and a lot more. It is reasonable to assume that this is one of the main reasons, why nearly every device is shipped with a web browser or a web server on board. This is also the main reason, why the REST-binding should also be compliant with a simple web browser, similar to the Amazon S3 REST API [5]. In the end, this would enable the use case to check a single OPC UA value without installing additional applications on most devices. It would be also much easier to download the device documentation directly from the device's OPC UA server by simply tipping in a URL into a standard web browser.

Programming against the TypeDefinition is a concept in OPC UA, which allows someone to find *Nodes* based on their *BrowsePath*. This is, for example, the preferred way to identify *Instance-NodeIds*, based on their complex *TypeDefinitions*. For this use case, the *TranslateBrowsePathsToNodeIds* service is introduced by the OPC Foundation. In a nutshell, this service allows you to chain several *Browse* requests in one *TranslateBrowsePathsToNodeIds* request. If a common URI template is introduced for that service, it would even be possible to apply this concept, in combination with redirections, across several servers.

Resolution of ExpandedNodeIds: The last goal is to be able to resolve *ExpandedNodeIds* in a simple way. This approach also depends on URI templates, because otherwise, it would not be possible to construct a valid URL only based on the actual available OPC UA structures. Only if an easy concept is introduced to link to other OPC UA servers, OPC UA is able to grow into a highly distributed network of millions of connected devices like the web.

4.1.2 OPC UA Service Overview

Table 4.1 shows all services of OPC UA and defines the mapping of these services to the REST architecture pattern. The second column named IM marks if the given service is modeled with the OPC UA information model (see also Section 4.3 for an example). The representation type for each service is depicted in column three. Notice that, services with information model mapping depend on the representation of other services like the *Call-Service* or the *Read-Service*. Furthermore, services like *OpenSecureChannel* and *CloseSecureChannel* are directly mapped to the HTTPs binding of OPC UA, while services like *Cancel*, *RegisterNode*, and *UnregisterNode* are mapped to RESTful design patterns (see also Section 4.6.3 and Section 4.6.4). Finally, Table 4.1 contains also new services. New services are introduced to allow, for example, a common concept for pagination. OPC UA addresses pagination typically with designated *Next services (e.g., *BrowseNext*). However, one exception from this design pattern can be found in the *HistoryRead* service, which integrates the pagination directly into the *HistoryRead*-service. The benefit from introducing an uniform *Next interface is explained in more detail in Section A.1.4.3. Another new service is the *ModifyReferences* service. This service is introduced to make use of JSON-Patch [29] semantics for the modification of OPC UA *References* (see also Section A.1.4.5) and, therefore, acts as a further example on how accepted design patterns of the World Wide Web can be integrated into the OPC UA REST binding. The *ResolvePath* service is used to simplify the *BrowsePath* concept in a RESTful way and is explained in greater detail within Section 4.6.5. Eventually, this mapping also covers RESTful subscriptions. Of course, subscriptions can be considered stateful by definition. However, it is also possible to design RESTful subscriptions. An example of such a design is introduced in Section 4.5. As already mentioned above some of the services are discussed in the following sections, while other services are not discussed at all. The main goal behind this work is to transport design principles rather than doing direct standardization work. For this reason, the most challenging and interesting services are selected to act as examples for the abstract mapping.

4.2 Standardization

The focus of the upcoming sections is all about contributions to the OPC UA standardization. First, Section 4.2.1 details the session concept of OPC UA, which prevents the introduction of the REST design pattern into OPC UA. Second, based on the analysis results of the previous section a new OPC UA service is constructed and contributed to the OPC UA standardization to allow the execution of OPC UA services without active sessions. Third, the standardized HTTP(S) API of the OPC Foundation is explained in greater detail.

Service	IM	Mapping Type
Discovery	X	OPC UA information model
SecureChannel		
OpenSecureChannel		OPC UA HTTPs binding
CloseSecureChannel		OPC UA HTTPs binding
Session		
CreateSession		Mapped trough SessionlessInvoke (see Section 4.2.2)
ActivateSession		Mapped trough SessionlessInvoke (see Section 4.2.2)
CloseSession		Mapped trough SessionlessInvoke (see Section 4.2.2)
Cancel	X	Mapped to a TaskHandle concept (see Section 4.6.3)
NodeManagement		
AddNodes	X	OPC UA information model
AddReferences	X	OPC UA information model
DeleteReferences	X	OPC UA information model
(new) ModifyReferences		Direct HTTP Mapping (see Section A.1.4.5)
DeleteNode		Direct HTTP Mapping
View		
Browse		Direct HTTP Mapping (see Section A.1.4.2)
BrowseNext		Direct HTTP Mapping (see Section A.1.4.3)
TranslateBrowsePaths		Direct HTTP Mapping
RegisterNode	X	Direct HTTP Mapping (see Section 4.6.4)
UnregisterNode	X	Direct HTTP Mapping (see Section 4.6.4)
(new) ResolvePath	X	Direct HTTP Mapping (see Section 4.6.5)
Query		
QueryFirst		Direct HTTP Mapping
QueryNext		Direct HTTP Mapping (see Section A.1.4.3)
Attribute		
Read		Direct HTTP Mapping (see Section A.1.4.1)
(modified) HistoryRead		Direct HTTP Mapping
(new) HistoryReadNext		Direct HTTP Mapping (see Section A.1.4.3)
Write		Direct HTTP Mapping
HistoryUpdate		Direct HTTP Mapping
Method		
Call		Direct HTTP Mapping (see Section A.1.4.4)
MonitoredItem	X	OPC UA information model (see Section 4.5)
Subscription		
CreateSubscription	X	OPC UA information model (see Section 4.5)
ModifySubscription	X	OPC UA information model (see Section 4.5)
Publish	X	OPC UA information model (see Section 4.5)
TransferSubscriptions	X	OPC UA information model (see Section 4.5)
DeleteSubscriptions	X	OPC UA information model (see Section 4.5)
(new) CreateGroup	X	OPC UA information model (see Section 4.5)
(new) DeleteGroup	X	OPC UA information model (see Section 4.5)

Table 4.1 – *Service Set* Overview.

4.2.1 OPC UA sessions

As identified in the analysis chapter (Section 3.1), the OPC UA session concept is the final challenge that stands between OPC UA and the REST architecture. A session has two major responsibilities in OPC UA: First, storing some information on client and server. Second, guaranteeing the consistency between the client and server state. All information of the first category can be identified as the information, which is only transferred once during the session set up, for example, the locales. However, identifying all information of the second category is a little bit more challenging. For that, an analysis is necessary to identify what kind of information is only guaranteed to be stable within a session. However, before the second category can be addressed, another OPC UA concept must be introduced first.

The so-called *NamespaceArray* in OPC UA contains an array of URIs. In a nutshell, the *NamespaceArray* is some kind of lookup table, which is used to replace long URIs through small indices. Based on this concept, *NodeIds* and *ExpandedNodeIds* only contain the corresponding index value, also called *NamespaceIndex* in OPC UA, for the URI, instead of a long URI. For example, the *NodeId* with *NamespaceIndex* 2 (see also the top of Figure 4.1) refers to the third element of the *NamespaceArray* (e.g., "http://opcfoundation.org/UA/DI"). OPC UA Part 5 defines the *NamespaceArray* as dynamic, which means, that the content of this array can be changed during runtime (see also Figure 4.1). *NodeIds* and *ExpandedNodeIds* are used in most of the OPC UA services (e.g., *Read* / *Write* / *Browse* / ...) and because of that, the content of the *NamespaceArray* has to be cached during the start-up phase by each client. Stale client caches and the fact that changes of the *NamespaceArray* during runtime are explicitly allowed, lead to the problem of Figure 4.1. The example session-less client wants to display the temperature based on the red *Node* (previously addressed through the *NamespaceIndex* "2" and the numeric identifier "1" - see also top of Figure 4.1), but fetches the value of the blue *Node* (after the updated *NamespaceArray* addressed through the *NamespaceIndex* "2" and the numeric identifier "1" - see also bottom Figure 4.1) after the *NamespaceArray* was changed. Of course, such behavior cannot be tolerated for any application. So, Part 5 also restricts the way in which these arrays can be altered. For example, a server is not allowed to change these arrays in such a way during an active client session. If a client has an open session, the server can only add some new entries but is not allowed to alter existing entries. But of course, such changes can be done during a reconfiguration, or a reboot, if all sessions are terminated during the process. If the session of a session-based client is terminated, also the cached values for the *Namespace*- and the *ServerArray* are not valid any longer and have to be re-fetched if a new session is established. The problem here is of course, that a session-less client cannot be sure if such a change happens during two subsequent calls.

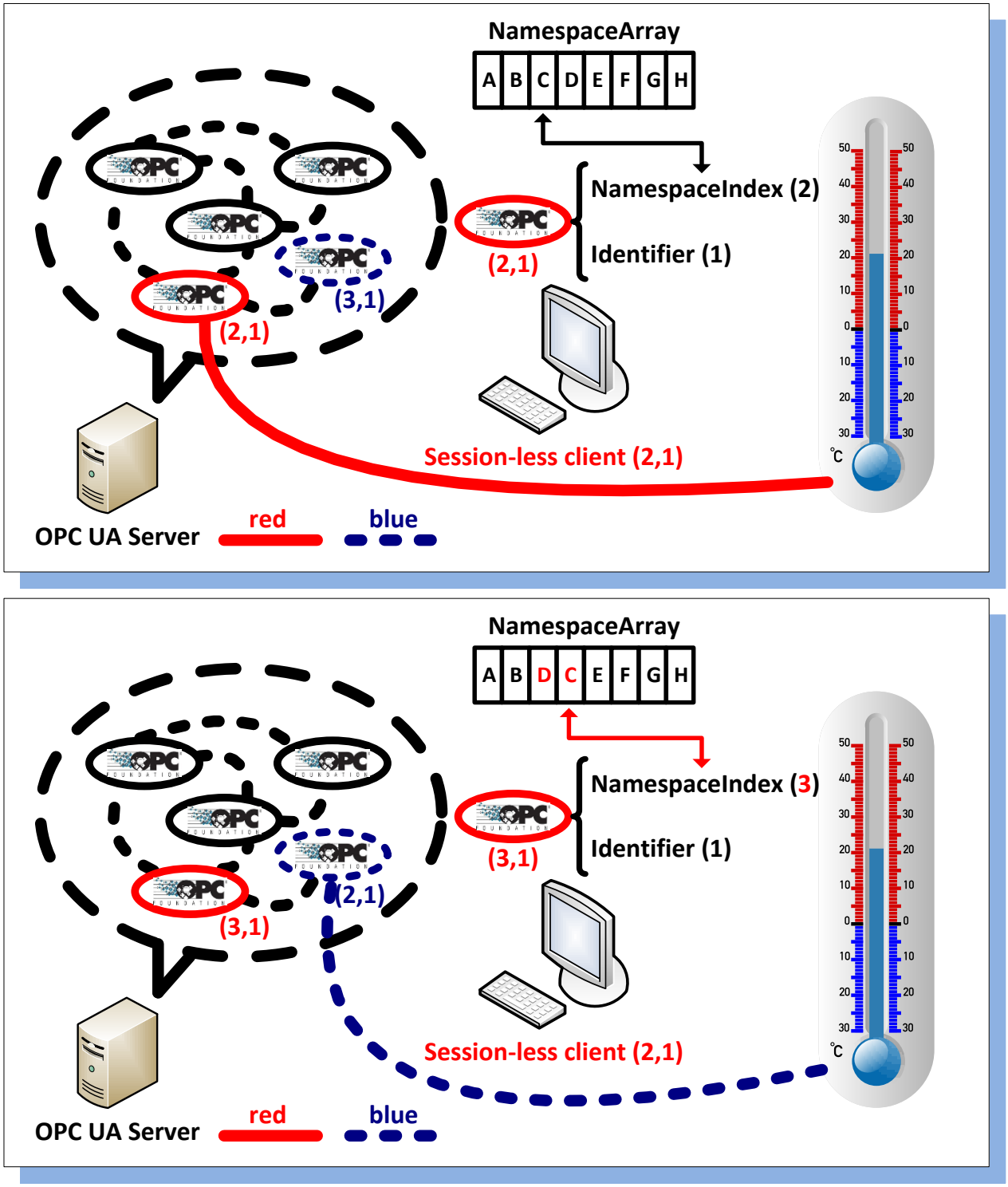


Figure 4.1 – Consequences of a changed *NamespaceArray* during runtime, without further preparations [137].

4.2.2 SessionlessInvoke

As previously stated, OPC UA cannot be considered stateless, even for services like *Read* or *Browse*. This is mainly because of the fact, that the *Namespace-* and *ServerArray* are only guaranteed to be stable within a session.

In Table 4.2 the service signature of the *SessionlessInvoke* service is shown. The (+) marks are the contributions of this thesis to the standardized service. A client is able to use *SessionlessInvoke* in two ways. One way is to specify the used *namespaceUris* and *serverUris* for each call and the other way is to set a so-called *urisVersion* in each call. The idea behind the first approach is straightforward and is not further discussed here. However, the second approach needs some further explanation. The basic idea behind the *urisVersion* is to versionize the *Namespace-* and *ServerArray*. Every time one of these arrays is changed, also the *urisVersion* must be altered. The *urisVersion* is introduced as new *Property* of the OPC UA *ServerObject*. Of course, a server has to ensure consistency between the *UrisVersion Property* and the *Namespace-* and *ServerArray*. This, for example, could be ensured by using the same semaphore. With that in mind, the concept is also straightforward. In the beginning, a client fetches the *NamespaceArray*, *ServerArray*, and the corresponding *UrisVersion*. The value of the *UrisVersion Property* has to be unique across restarts. This can be, for example, ensured through the usage of a time stamp in combination with time synchronization. After that, a client assigns the *NamespaceIndices* based on the cached arrays and sets the *urisVersion* field to the also cached *UrisVersion* value. An OPC UA server now only has to check if the *SessionlessInvoke urisVersion* field matches the local *UrisVersion Property* of the server. If this is not the case because, for example, the *NamespaceArray* was changed in the meantime, the server discards the request and informs the client about the stale cache values with a *BAD_VersionTimeInvalid StatusCode*. After that, a client has to refresh its cache and then is able to retry the request. To ensure cache consistency on the client-side, the client should first fetch the *UrisVersion Property* in a single request and after the response is received, try to fetch both arrays. This is necessary because in general there is no transaction context and also no sequential execution guarantee between different items in a batch request. However, it might be possible that

Name	**Type**	**Name**	**Type**
Request		**Response**	
(+) urisVersion	VersionTime	namespaceUris[]	String
namespaceUris[]	String	serverUris[]	String
serverUris[]	String	serviceId	UInt32
(+) localeIds[]	LocaleId	body	*
serviceId	UInt32		
body	*		

Table 4.2 – *SessionlessInvoke* service parameters [77].

some OPC UA servers support this feature within a single batch request, but this should not be assumed in general. Besides the above mentioned contributions, more contributions are made to correct some problems in combination with *SessionlessInvoke*, like the redefinition of the *Method* description: "Each *Method* is invoked within the context of an existing session" (OPC UA Part 3 V1.03).

4.2.3 Standardized HTTP(S) API of OPC UA

Section 2.1 introduced the possibility to use HTTP(S) as a transport layer for OPC UA. The initial main reason for this mapping is the now deprecated SOAP (Simple Object Access Protocol) binding of OPC UA. However, it is also possible to use OPC UA in combination with HTTP(S) without SOAP. Furthermore, in V1.04 of the OPC UA specification, the HTTP(S) section was extended for the newly added *SessionlessInvoke-Service*. Part 6 of OPC UA V1.04 now also allows invoking *Session-less Services* via HTTPS POST. In this case, the HTTP Authorization header shall have a bearer token as an access token provided by the *Authorization Service*. In addition, the content-type header shall be used to specify the encoding (e.g., "application/opcua+uabinary" or "application/opcua+uajson"). Finally, this allows to leverage HTTPS POST to invoke OPC UA services through the *SessionlessInvoke-Service*. However, this mapping cannot be considered a REST mapping according to Fielding [46] because, for example, every service call is mapped to HTTP POST and also the resource representation is not self-descriptive. Because of that, the following sections provide further insights into a REST mapping derived from the guidelines of Fielding.

4.3 Information Model Mapping

In Section 4.1.2 two main categories for the service mapping are identified. In the following, service mappings modeled with OPC UA information models are shown in an exemplary way through the *Discovery Service Set* (Section 4.3.1). In addition, Section 4.3.2 focuses on the question of how batch support can be introduced into a REST architecture for OPC UA.

4.3.1 Discovery Service Set

The *Discovery Service Set* acts as an example for all information model services of Table 4.1. Figure 4.2 depicts one part of the *Discovery Service Set* information model. The **NodeManagement**-*Object* of **RegisteredServersType**-*ObjectType* is the entry point in a list of complex *Variables* of type **RegisteredServerType**. Each registered server generates exactly one entry in the list with the *ServerId* as *BrowseName*. The structure of the *VariableType* **RegisteredServerType** is based on the *RegisteredServer* structure from OPC UA Part 4. In addition, the **NodeManagement**-*Object* also

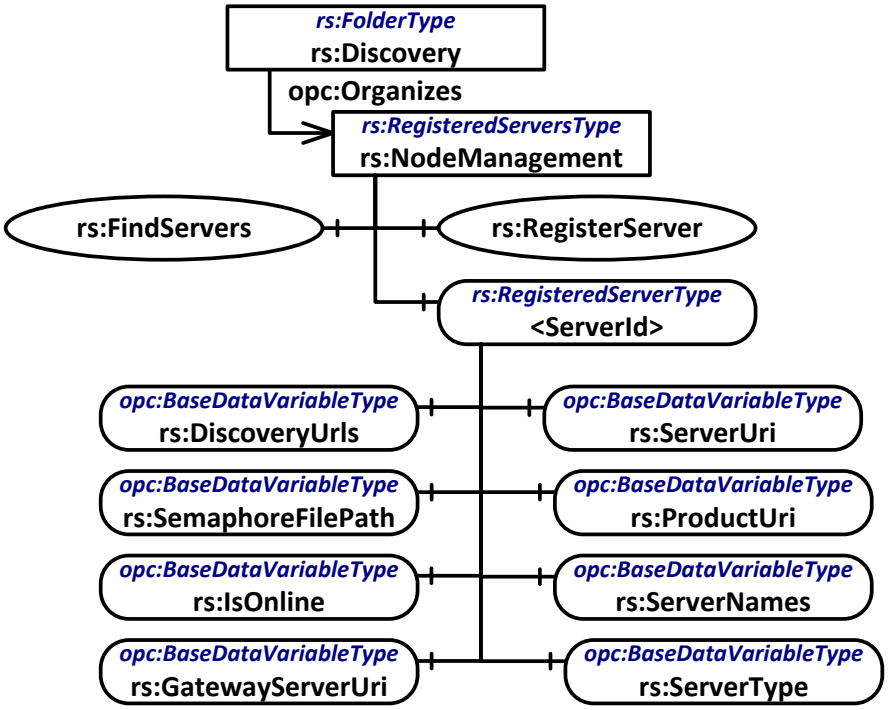


Figure 4.2 – *Discovery Service Set* information model.

```
1 Signature
2
3   FindServers (
4     [in]  String endpointUrl
5     /* LocaleIds are moved to header */
6     [in]  String[] profileUris
7     [out] EndpointDescription[] servers
8   );
```

Listing 4.1 – FindServers method signature.

contains two *Method-Nodes*. Both *Methods* represent the corresponding services of the *Discovery Service Set*. The signature of the **FindServers**-*Method* is shown in Listing 4.1. Line 5 of Listing 4.1 also shows that in some cases service parameters have to be modified. In this case, the parameter *localeIds* is already present in the header of the *SessionlessInvoke*-service, which is used as an envelope for all other services (see Section 4.2.2). Because of that, the *localeIds* parameter can be omitted in the *Method* signature. Besides calling the **RegisterServer**-*Method* also the *AddNodes*-service of OPC UA Part 4 can be used to achieve the same behavior. Notice that, OPC UA also defines a concept of client-side *Object* creation in OPC UA Part 3. OPC UA Part 3 defines special semantics for *ObjectType-Methods* with the *BrowseName* "Create". This special *Method* has to be attached to an *ObjectType* and creates an instance of the given *ObjectType* during execution. In contrast to the *AddNodes*-service, such a *Method* can also be used to set initial parameters for substructures of the *ObjectType-Instance*. However, if it is assumed that the **RegisterServerType** would be an *ObjectType* instead of a *VariableType* some usability problems in combination with a RESTful architecture would arise. In the example of Figure 4.2 the *Create-Method* would be attached to the **RegisteredServerType** instead of the **NodeManagement**-*Object*. While this solution offers standardized semantics based on the OPC UA specification it still might be harder for non OPC UA users to figure out the semantics on how new servers can be registered by just browsing the *AddressSpace*. This can be explained through the violation of one REST paradigm named self-descriptive representations. In this case, a client has to be aware of this OPC UA-specific concept and has to look up *ObjectTypes* for "Create"-*Methods*. In contrast, the create function (**RegisterServer**-*Method*) of Figure 4.2 is also part of the representation of the list object itself. In this case, the description of how to manipulate the list object (e.g., adding new servers) is part of the representation.

For endpoints, a similar structure is introduced with the **Endpoints**-*Object* of type **EndpointsType** as the entry point in a list of the complex *Variables* of the **EndpointType**-*VariableType*. Also in this case, the **Endpoints**-*Object* contains a *Method* with the *BrowseName* **GetEndpoints** to offer the corresponding service of OPC UA Part 4. In conclusion, the design pattern can be summarized in the following way:

1. Define *Structures* for complex service parameters (e.g., **RegisteredServerType**)

2. Find a representation of the service in the OPC UA information model (e.g., a list like **RegisteredServersType**)

3. Expose the service based on the representation (e.g., **FindServers**-*Method* in combination with structured *Objects* and/or *Variables*).

4.3.2 Batch support

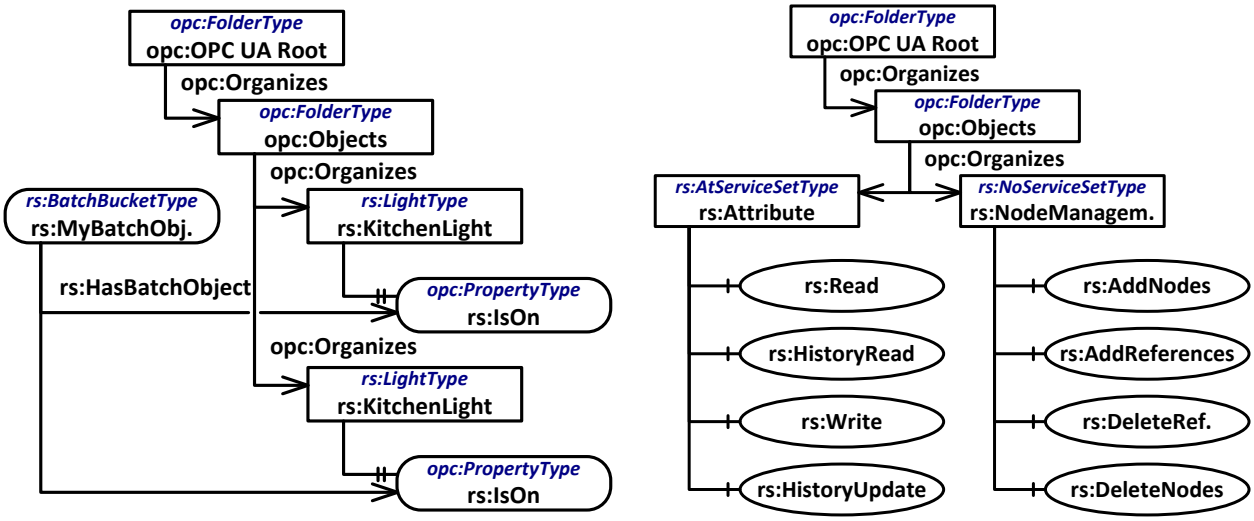
Most OPC UA services support batch requests. This feature allows specifying more than one item per service request. The main reason for this feature is to reduce the data on the wire and the necessary processing overhead in client and server applications. Classic REST APIs on the other hand are often optimized to access single resources. This is mainly because of the different use cases. For example, it often makes no sense to request all web-pages of a certain domain or all objects from a given Amazon S3 bucket. Another reason is, that big web services are hosted by a lot of servers, often with caches and load balancers in front of them. Because of that, it would even make more sense to break down batch requests into single requests and distribute them to different servers. In contrast, a typical OPC UA server runs on an embedded controller, which is often the only source for the data. Having identified the necessity of batch requests, several concepts are proposed, how batch requests can be mapped to this RESTful OPC UA API:

1. Definition of some kind of batch-*Node*, which can be configured through the client by adding special *References* (see Figure 4.3a).
2. Definition of a *Method*, similar to the classic OPC UA *Read* service (see Figure 4.3b).
3. Definition of a special batch URL for receiving and processing concatenate REST requests, similar as depicted in Listing 4.2.

The first concept is depicted in Figure 4.3a. In this case, a special *VariableType* named **BatchBucketType** is introduced. With the *AddReference*-service a client can add **HasBatchObject**-*ReferenceType References*. A **HasBatchObject**-*ReferenceType* is a subtype of the *HasOrderedComponent-ReferenceType* of OPC UA Part 3. The *HasOrderedComponent-ReferenceType* has the special characteristic that for each *Browse*-service request the *References* are returned in the same order.

```
[
  {
    "href": "/i=84/BrowseName",
    "method": "get"
  },
  {
    "href": "/i=2255/Value",
    "query": {"timestampsToReturn":2},
    "method": "get"
  }
]
```

Listing 4.2 – OPC UA RESTful batch request (inspired by [128]).



(a) Example of an BatchBucket-*Instance*.

(b) *Method*-based batch services.

Figure 4.3 – OPC UA information model concepts to expose batch requests.

Furthermore, the target *Node* of a **HasBatchObject**-*ReferenceType* must be of the *NodeClass Variable*. The *Value-Attribute* of the **BatchBucketType**-*Instance* returns an array of *DataValues* with the size of the number of **HasBatchObject**-*References*. Each entry of the array includes the *Value-Attribute* of the corresponding target *Node* of a **HasBatchObject**-*References*. The order is based on the return order of the *Browse*-service. After the creation of such a *Variable* a client is able to execute batch reads and batch writes on the defined group by simply reading or writing the *Value-Attribute* of the **BatchBucketType**-*Instance*.

The second concept is displayed in Figure 4.3b with an example information model of the batch versions for the *Attribute Service Set* and the *Node Management Service Set*. This case is similar to the concept of how the *Discovery Service Set* is mapped into the information model (see also Section 4.3.1).

The third concept is exemplified through a concept which is inspired by [128] (see also Listing 4.2). In this case, a client can serialize several REST requests including the URI, the HTTP-verb, headers, query arguments, etc. in a single JSON message and sends them with a POST request to the server. Of course, it would also be possible to further optimize this concept with shared values (e.g., the definition of one header-value for a group of objects).

While the first two approaches look more natural to an OPC UA client, the third one may look more familiar to typical web clients. However, the second approach should be the approach with the lowest implementation effort. In the end, all three approaches can be used to introduce batch capabilities in a RESTful OPC UA API. For the prototype, the second approach is chosen because this is, besides the lowest implementation effort, also a chance to show some *Method NodeClass* related concepts of the design.

4.4 HTTP mapping

Within this section, several aspects of the REST-based HTTP mapping are discussed. In the beginning, Section 4.4.1 gives a brief overview of how different HTTP verbs are mapped to OPC UA services and what kind of resource representation is exchanged during the service execution. Followed by the overview, Section 4.4.2 provides the mapping of OPC UA parameters to HTTP header and query arguments. Section 4.4.3 outlines the design rationale behind the mapping of result codes, while Section 4.4.4 opens a discussion about URI generation schemes. Finally, Section 4.4.5 introduces one of the most important resource representations of the REST mapping.

4.4.1 Mapping to HTTP verbs

Within Annex A.1.2 Table A.4 shows details of the basic approach of how some of the abstract services from OPC UA can be mapped to HTTP. Of course, it is not enough to only specify how OPC UA services should be mapped to HTTP, instead, one of the main topics of each REST-binding is the description of the resource representations.

4.4.2 Header and Query Mapping

Within Annex A.1.3 Table A.5 provides details of the mapping between common OPC UA parameter (the first column) and HTTP-Headers (the third column).

4.4.3 ResultCodes

In general, OPC UA also defines result codes for the different service requests similar to HTTP return status codes. Also, these codes can be translated into each other. For example, the OPC UA result code "Good" can be mapped to the HTTP status code "OK". The mapping could even be improved further through the introduction of so-called sub status codes. An example of such a code is the 403.2 for read access forbidden and 403.3 for write access forbidden as defined by Microsoft's IIS 7 [72]. However, most standard web clients would not be aware of OPC UA sub status codes and OPC UA optimized web clients should use the *StatusCode* directly through the corresponding header of Table A.5. Because of that, only a mapping between the well-known more abstract status codes seems beneficial.

4.4.4 URI design

The basic design pattern of REST does not talk a lot about the design of URIs. In contrast, most so-called "REST architectures" wrongly exclusively focus on the URI design and how certain functions

and/or structures can be encoded in URI templates. As also explained previously, the idea behind REST is much more than just the correct usage of HTTP and URIs. However, in practice it is often beneficial to also think about URI patterns. The main reason for that is, that such a pattern also forces the developers to consider the underlying resource structure, which in most cases leads to a more RESTful design in the end. For this thesis, an URI generation schema is designed, which is exemplified in Annex A.1.1 and is used to generate the URIs throughout this thesis. Notice that, most of the REST designs discussed in this chapter do not depend on any URI generation schema but could be easily combined with them. Furthermore, the definition of URI patterns also leads to a closer coupling of client and server, which is typically the main reason why URI patterns should not be provided/standardized in a RESTful design.

4.4.5 Resource Representation

In addition to the MIME-Types of Table A.4, Table 4.3 specifies the mapping for default OPC UA-*DataTypes*. The generation schema can easily be guessed for the missing *DataTypes*.

Furthermore, also a new *DataType-Property* “DefaultMimeType” (see also Table 4.4) is introduced. This *Property* is used in combination with the *Read* service. If a web client uses the *Read* service to request the *Value-Attribute* of a *Node* and the DefaultMimeType-*Property* is present, the server shall set the *Value-Attribute* of the DefaultMimeType-*Property* as Content-Type and serialize the payload accordingly. In most cases, a web browser is able to directly display PDFs or image formats like BMP. However, not each supported MIME type is also published through the Accept header. For this case, a server shall always return the "DefaultMimeType" if not explicitly an OPC UA-specific format is requested by setting the Accept header to, for example, application/opcua+json.

Besides the standard OPC UA *DataTypes* and some special features of the *Read* service in combination with the *Value-Attribute* the most important resource representation is returned by

DataType	**MIME-Type**
Boolean	application/opcua.Boolean+json application/opcua.Boolean+binary application/opcua.Boolean+xml
SByte	application/opcua.SByte+json application/opcua.SByte+binary application/opcua.SByte+xml
...	...
DiagnosticInfo	application/opcua.DiagnosticInfo+json application/opcua.DiagnosticInfo+binary application/opcua.DiagnosticInfo+xml

Table 4.3 – OPC UA *DataType* to MIME-Type mapping.

```
{
  "attributes": {
    "NodeIdValue": {
      "NodeId": {
        "namespace": 1,[...]
        "href": "<host>/1/s=1:Boolean"
      },[...]
    },[...]
    "Value": {
      "href": "<host>/1/s=1:Boolean/Value",[...]
    }
  },
  "referencesStatusCode": 0,
  "references": {
    "(!i=0:47)s=1:StaticVariablesFolder": {
      "nodeId": {[...]
        "href": "<host>/1/s=1:StaticVariablesFolder"
      },[...]},[...]
  },
  "forms": {
    "DefaultRead": {[...]
      "href": "<host>/1/s=1:Boolean/{attributeName}
               {?opcuaAuthenticationToken}",
      "method": "GET",
      "uriTemplate": true,
      "jsonschema": {
        "type": "object",
        "properties": {
          "attributeName": {
            "type": "string",
            "description": "The attribute name",
            "enum": ["NodeId",...]
          },[...]
        },
        "required": [ "attributeName" ]
      }
    },[...]
  }
}
```

Listing 4.3 – Example of application/opcua.NodeRepresentation+json (simplified).

Name	Use	DataType	Description
Standard Properties			
DefaultMimeType	O	String	The default MIME type of the DataType. In the case of *DataType ImagePNG* the *Value-Attribute* of this *Property* would be "image/png" according to IANA [75].

Table 4.4 – Additional MIME type *Property* for *DataTypes*.

the **Browse** service (see also Listing 4.3). The basic structure of this representation consists of three parts: **attributes**, **references**, and **forms**. In the **attributes** section all *Attributes* of a given *Node* are summarized, while the **reference** section offers all *References* of the *Node*. However, the **form** section might not be expected by an OPC UA user. One of the preconditions of a RESTful API is, that a client should be able to explore all service functions without additional documentation [47]. If this basic concept behind REST is taken into consideration, it is not enough to only display somehow that there are additional resources, for example, the **StaticVariableFolder** resource, instead, also all necessary information to access the resource must be provided to the client. This is done by introducing a new field in the OPC UA *NodeId* structure with the name **href**. The value of the **href** field is a valid URL to the target *Node* of the *ReferenceDescription* structure. The **href** field "<host>/2/s=1:Boolean" references the *NodeId* with the string identifier **Boolean** defined in *NamespaceIndex* **1**. In addition, the *SessionlessInvoke urisVersion* **2** is specified in the URI (between <host> and the *NodeId*). Notice that, this additional field does not force OPC UA architects to add additional information to already existing information models, instead, it can be easily generated automatically out of existing information. But expressing all possibilities with links would add a large amount of data to each representation. For that reason, the **form** section is introduced. In this section, it is possible to express the standard functions of the service based on URI templates [49]. In the **forms** section of Listing 4.3 an example for a possible *Read* URI template is given. For performance reasons, this section can also be suppressed through special filter settings.

A typical graphical OPC UA Client, like UAExpert [158], often offers some kind of template if the OPC UA *Call* service is invoked by the user. This template is based on well-known OPC UA properties, which are part of each *Method-Node*. However, a typical web client is not aware of these OPC UA-specific conventions. Because of that, the **form** section also provides some additional information if OPC UA *Method-Nodes* are involved. The most important part of such a representation is depicted in Listing 4.4. The representation provides all necessary information, including the target URL, the HTTP method, and the expected Content-Type. The section **requestSchema** and **responseSchema** provide an URL to the corresponding schema description for the request/response Content-Type. This is done to reduce the amount of data for clients which already know how to formulate a valid request for this *Method-Node*. Nevertheless, a generic client still is able to request a schema

```
{
  "href": "<host>/1/s=1:AttributeServiceSet",
  "method": "POST",
  [...],
  "accepts": ["application/opcua.CallRequest+json"],
  "requestSchema": {
    "href": "<host>/1/s=1:Read/RequestSchema"
  },
  "responseSchema": {
    "href": "<host>/1/s=1:Read/ResponseSchema"
  }
}
```

Listing 4.4 – Representation of *Methods* in the form section (simplified).

description by following the given URL. An example schema description can be found in Annex A.1.5. This schema description is based on JSON schema [168] version 1.4, which is well-known on the web and also offers client-side validation. Through HTTP content negotiation it is also possible to serve other schema descriptions.

4.5 Group-Subscriptions

As previously mentioned, OPC UA *Subscriptions* belong to exactly one client. If this exclusive relation is broken it becomes possible to share the same *Subscription* across several different clients and thus safe resources. In the following, Section 4.5.1 gives an overview of the main architecture behind group *Subscriptions*. Section 4.5.2 introduces the corresponding OPC UA information model and finally, the evaluation is provided in Section 4.5.3. Parts of this section are also published in [138].

4.5.1 Architecture

Figure 4.4 gives an overview of the architecture. On the left side, several clients are depicted, which want to be informed about updates in the OPC UA server. The right part of Figure 4.4 depicts the application. For the core architecture, two goals are identified: (1) Introduction of Group-Subscriptions in such a way that most parts of the existing OPC UA implementations do not have to be altered. (2) Group-Subscriptions should be usable with session-less clients as well as with session-based clients. Eventually, the decision is made to model Group-Subscriptions within the OPC UA information model. The OPC UA information model itself consists of methods, which can be used to recreate the *Services* of the *MonitoringItem Service Set* and the *Subscription*

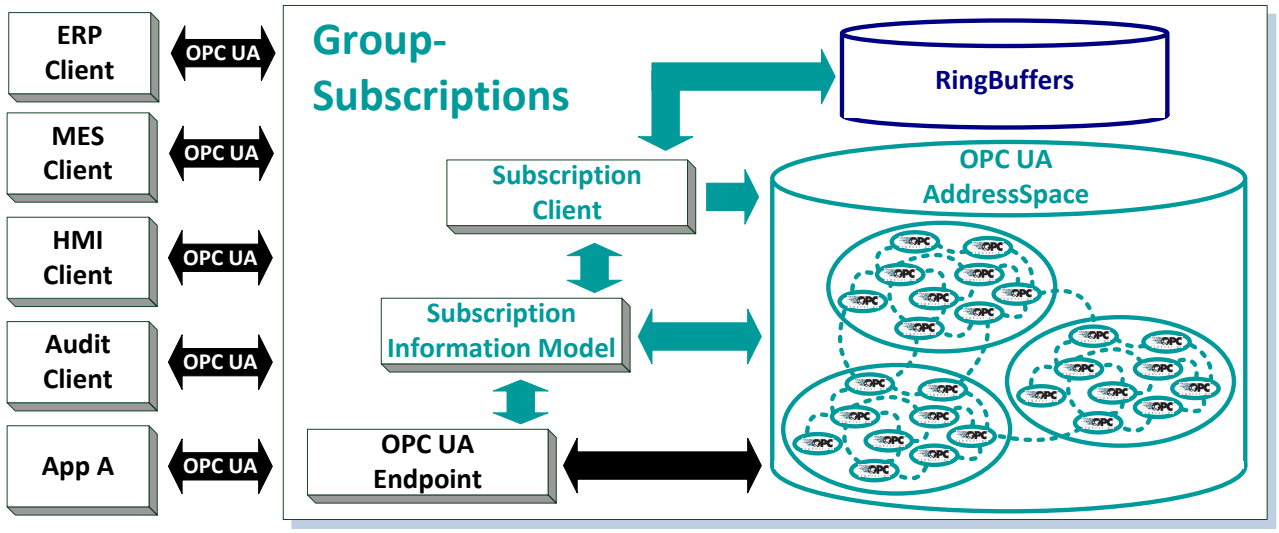


Figure 4.4 – Group-Subscriptions architecture [138].

Service Set, and further *Nodes*, which contain more information about the Group-Subscription configuration. Based on this information model an internal client can be controlled, which creates session-based *Subscriptions* through standard SDK calls. Nevertheless, normally *Subscriptions* and some of the corresponding *Services* are designed for a single client only. For example, the *Publish-Service* allows a client to acknowledge received sequence numbers. If a sequence number is acknowledged the server is allowed to delete the message from the internal buffers. Of course, if more than one client is using the same *Subscription* such behavior can lead to data loss for some of the clients. Because of that, ring buffers are introduced. The size of the ring buffers can be chosen during the creation of the Group-Subscription.

4.5.2 Information Model

Figure 4.5 shows parts of the developed information model. A lot of the elements should be familiar to OPC UA experts. However, some of them are altered or newly introduced. For example, the **Publish**-*Method* contains two input parameters: **publishSeqNr** and **keepAliveTime**. The first parameter is used to request a certain sequence number. This sequence number is automatically generated by the subscription client (see Figure 4.4) and incremented for each *Publish* response with new values. The *keepAliveTime* is used for long polling and defines the maximum time the call should be blocked. The second parameter is only important if the client requests a sequence number which is larger than the latest available sequence number. In this case, the method call is blocked for the given time amount. If during this time the requested sequence number becomes available the call immediately returns with the new results, otherwise, the client is informed that no results are available at the moment. A client can use this architecture to also request more than

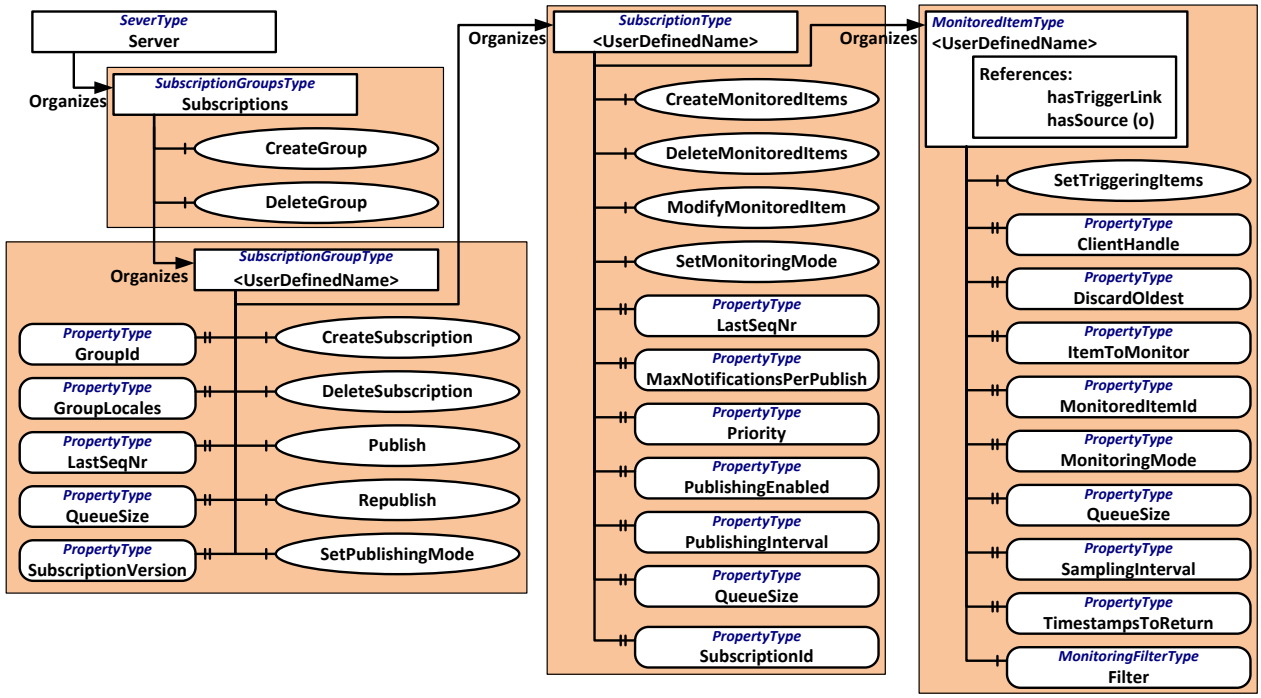


Figure 4.5 – Group-Subscriptions information model [138].

one future sequence number, which is equivalent to calling the standard *Publish-Service* several times. Of course, not all sequence numbers can be requested because of limited memory resources. The number of available sequence numbers can be configured through the queue size. However, clients which are joining late typically have no information about the actual sequence number. Because of that, a *Property* with information about the latest sequence number (**LastSeqNr** in Figure 4.5) is introduced. The initial values can be fetched with the standard *Read* service of OPC UA. The **SubscriptionVersion**-*Property* of Figure 4.5 indicates if the Subscription-Group is changed. The actual value of the **SubscriptionVersion**-*Property* is also returned in the **Publish** and **Republish** *Methods* and can be used by the client to check if the Group-Subscription is altered. Figure 4.6 depicts the newly introduced *ReferenceType*.

The **HasNotificationSource** is defined in the following way: The HasNotificationSource *ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HierarchicalReferences*. The semantics of this *ReferenceType* is to bind a *MonitoredItemType* to the *Object* or *Variable* which shall be monitored. The *SourceNode* of this *ReferenceType* shall be a *MonitoredItemType Object*. The *TargetNode* of this *ReferenceType* shall be the *Object* or the *Variable* which shall be monitored by the *SourceNode*. Each *MonitoredItemType* shall be the *SourceNode* of exactly one **HasNotification-Source** *Reference*.

The **HasTriggerLink** is defined in the following way: The **HasTriggerLink** *ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*. The

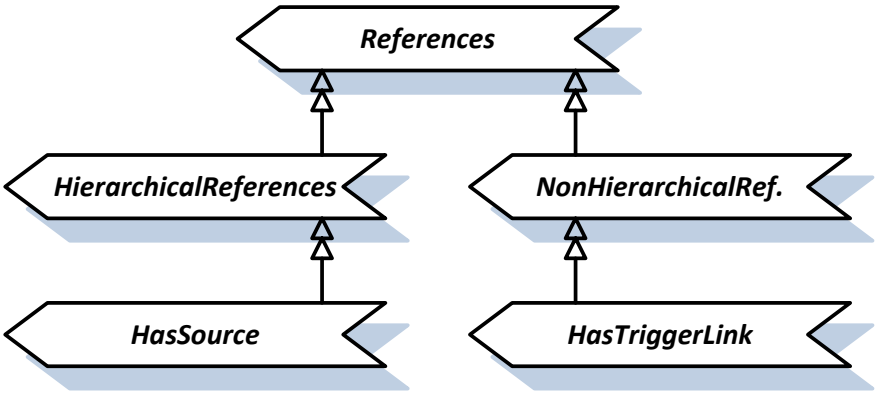


Figure 4.6 – *ReferenceTypes* of the Group-Subscription information model.

semantics of this *ReferenceType* is to bind a *TriggeringItem* to the corresponding *ItemToReport* (see also OPC UA Part 4 triggering model for further explanation). The *SourceNode* of this *ReferenceType* shall be the *TriggeringItem*. The *TargetNode* of this *ReferenceType* shall be the corresponding *ItemToReport*.

4.5.3 Evaluation

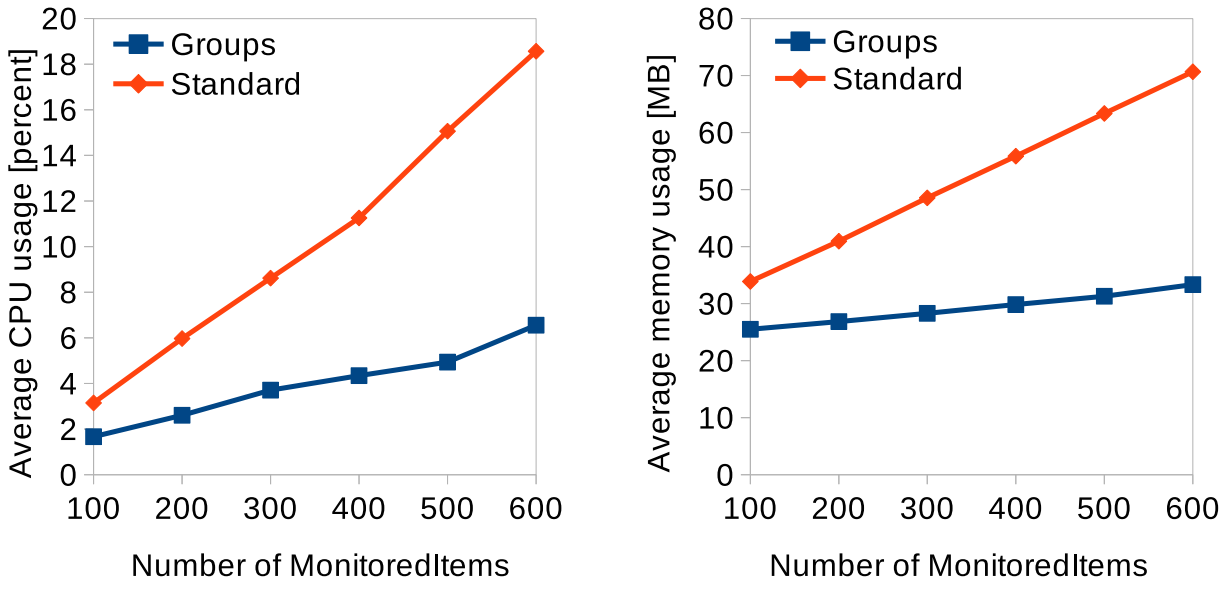
The concept of Group Subscriptions is implemented with the C++ SDK from Unified Automation (version 1.5.6) [157] to show the validity and the benefits of this architecture. Most of the features like the information model and the usage of an internal client could be integrated very easily into the SDK. However, for some obstacles, it is required to also change code deeper in the SDK. For example, the original SDK does not allow to execute the *Call Service* on the same OPC UA *Method* in parallel by more than one client. Instead, the next method call can only be executed if the previous one is finished. This is not a requirement of the OPC UA specification but an important implementation detail for the approach described above. To allow scenarios where clients fetch previous results while other clients are blocked for new results the SDK is extended with concurrency concepts for the *Call* service.

In the following, the memory consumption and CPU usage of Group-Subscriptions in comparison to the respective characteristics of standard OPC UA *Subscriptions* are analyzed. In all cases, the prototype runs on a machine with 4 logical cores, 2.6 GHz, and 8 GB RAM. Additionally, the sampling interval for each *MonitoredItem* is set to 500 ms, the publishing interval is set to 1000 ms and all queue sizes are set to twenty. Each experiment runs for two minutes and data points represent the average of three runs. If Group-Subscriptions are used, each client makes use of the same *Subscription*, which can be accessed through the information model of the OPC UA server.

In the case of standard OPC UA *Subscriptions*, each client creates its own *Subscription* and the corresponding *MonitoredItems*.

In the first experiment, the average CPU load and memory consumption for different numbers of *MonitoredItems* per *Subscription* is evaluated. In each experiment 100 clients connect to the OPC UA server and establish the corresponding *Subscription*. Figure 4.7 represents the results for the CPU load and memory consumption, respectively. As expected, the benefits of Group-Subscriptions continue to grow with the number of *MonitoredItems*. This is true for the CPU load as well as for memory usage.

In the second experiment, the average CPU load and memory consumption for different numbers of *Clients* per *Subscription* with a constant number of 300 *MonitoredItems* is evaluated. Also, for these experiments, 100 clients connect to the OPC UA server in each test run to minimize the measurement distortion through a different number of client *Sessions*. However, in this experiment, only 10 to 60 clients actively used *Subscriptions*, while the other clients were just on idle during the experiments. Figure 4.8 represents the results for the CPU load and memory consumption, respectively. For the average CPU usage as well as for the average memory consumption the benefits of Group-Subscriptions increase with the number of clients. However, as also depicted in Figure 4.8b for very few clients the standard *Subscription* mechanism offers less memory consumption than the Group-Subscription prototype. This is mainly due to the fact, that in the experiments for the standard *Subscription* mechanism the clients never were late and acknowledged instantly all sequence numbers leading to at most one publish item in the buffers. In contrast, in the



(a) CPU usage

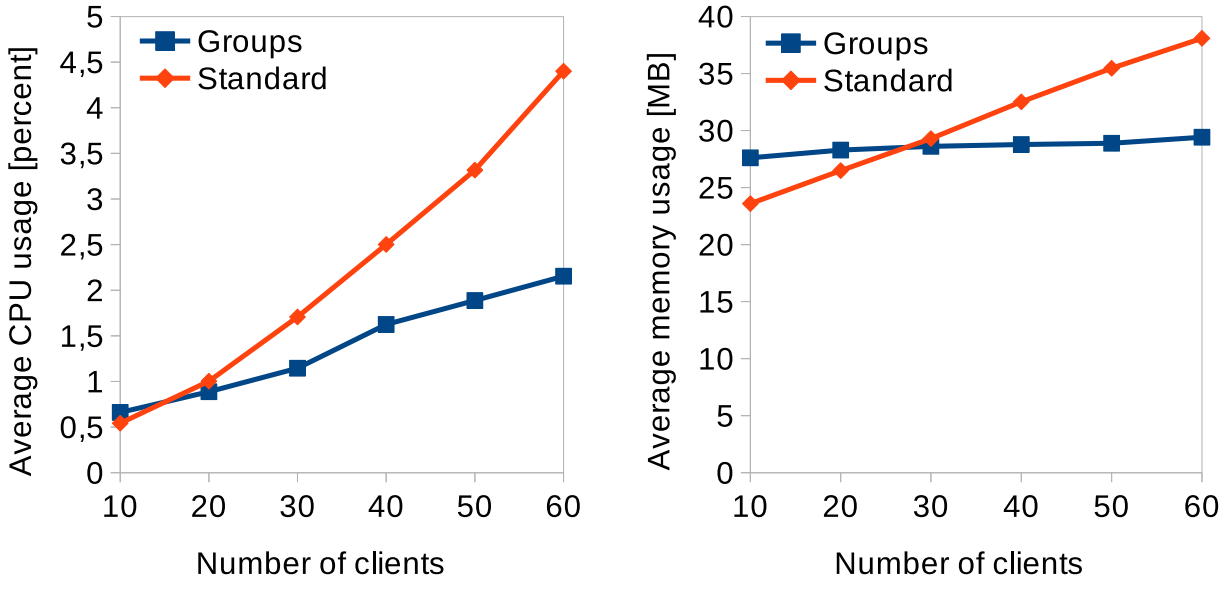
(b) Memory usage

Figure 4.7 – Constant number of 100 clients [138]

Group-Subscription case, the buffer cannot be emptied, because the number of clients is unknown and therefore, constantly twenty publish requests are kept in the queue. This also leads to the fact that for Group-Subscriptions the memory-consumption is nearly constant regardless of the number of clients (see Figure 4.8b). Nevertheless, the CPU usage still increases with the number of clients. The explanation for this effect is based on the *Call Service Set*, which generates a higher CPU load if more calls are made and therefore is responsible for the slight increase in CPU usage.

4.6 RESTful features

This section highlights several RESTful features, which are well-known to web clients and also can be used to gain additional benefits for the REST mapping of OPC UA. In the beginning, Section 4.6.1 presents a concept to access OPC UA directly through a web browser. In the following, an approach is introduced to ensure the generation of unique *NodeIds* across server restarts (Section 4.6.2). Section 4.6.3 outlines the design rationale behind the TaskHandle concept. This concept allows a session-less client to execute long-running service calls and also offers the possibility to cancel such calls. The next section explains the concept behind a RESTful implementation of the *RegisterNode* optimization service and also highlights how every client can benefit from such an optimization (Section 4.6.4). Finally, Section 4.6.5 presents the concept behind the newly designed ResolvePath service and also the usage of this service in distributed environments.



(a) CPU usage

(b) Memory usage

Figure 4.8 – Constant number of 300 MonitoredItems [138]

4.6.1 Browser support

One major reason to use HTTP is the chance to make OPC UA compatible to a standard web browser, without forcing a user to install additional plugins, or placing some gateway server in front of the OPC UA server. This is, of course, no requirement for a RESTful architecture, but might come in handy for use cases like documentation download via a web browser. It is also possible to use this feature for delivering a full-featured OPC UA client based on JavaScript, to gain full access to all features of OPC UA, without installing any plugin (also known under the term code-on-demand).

To enable this feature in the first place, the *Read* service of the REST mapping must be examined in greater detail, which is used to fetch the *Value-Attribute*. If an URL is typed into the browser a HTTP GET request is sent to the specified URL. However, only mapping the *Read* service to HTTP GET would not solve the issue. A browser also interprets the Content-Type HTTP header. If a browser does not know the format, which is often the case for MIME-Types like "application/opcua+uajson", of course, nothing useful can be displayed in the browser window. Because of that, a new optional *Property* for *DataType-Nodes* with the *BrowseName* "MIMEType", containing a string value with the MIME-Type (e.g., "application/pdf"), which shall be used as Content-Type HTTP header is introduced. But again, this is not enough because some native OPC UA clients only understand the MIME-Type "application/opcua+uabinary". To also cover this use case an OPC UA server shall interpret the HTTP Accept header of the request message. If a client specifies some well-known OPC UA MIME-Type as the highest priority, the specified type shall be used to encode the message body. If not, the MIME-Type of the MIME-Type-*Property* shall be returned. Based on the above rules an OPC UA server is now able to deliver, for example, PDFs, which can be directly displayed in the browser, without any additional OPC UA-specific plugin.

Another useful optional feature is the possibility to encode all *RequestHeader* fields as HTTP query parameters. Of course, it also makes sense to be able to encode the fields in HTTP headers, as the OPC Foundation did for the HTTP authorization header (see also OPC UA Part 6 for further information). However, if somebody wants to share a link to an OPC UA *Node*, it must also be possible to encode this token in the URL, otherwise, there is no guarantee, that the value can be fetched in each case. Just consider a simple dashboard web-application, which only allows specifying a URL, but does not allow to set any kind of header like [38].

4.6.2 Unique Runtime Namespace

Some REST paradigms enforce the generation of dynamic nodes. For example, in Section 4.5 a concept is introduced, which generates several *Nodes* for Group-Subscriptions in the information model. During a restart of a typical OPC UA server normally all *Subscriptions* are deleted. In

contrast to a session-less client, a session-based client takes notice of a server restart and can clear cached *NodeIds*. Problematic behavior can now arise if dynamic *NodeIds* are reassigned to different *Nodes* with the same *NamespaceURI*. This is, for example, often the case for *NamespaceIndex* one (the local server URI). A session-less client with cached *NodeIds* could now get other entries as expected (e.g., another *Subscription*). This issue can be addressed through different concepts: (1) Ensure that *NodeIds* of dynamic *Nodes* are not reassigned. (2) Ensure that the *NamespaceURI* of a *Namespace* with dynamic *NodeIds* can be versioned. However, approach one forces the OPC UA server to keep track of the assigned *NodeIds*. In contrast, approach two frees the server from the burden to keep track of dynamic *NodeIds* by simply introducing a new *NamespaceURI*. Based on that this work recommends using approach two wherever possible. The key idea of approach two is to introduce a so-called unique runtime *Namespace*, which can be used for dynamic generated nodes. The *Namespace* is typically generated during runtime of the server and must be unique for the server (including uniqueness across restarts). One way to ensure uniqueness is to generate a GUID (e.g. based on a timestamp and additional information) and use this identifier as part of the *NamespaceURI*. If the *Namespace* is full, or the server crashes, a server shall generate a new unique runtime *Namespace*. Because the *NamespaceURI* contains a GUID, there are no collisions with other or previous runtime *Namespaces*. The URI of an unique runtime *Namespace* could have the following structure: “http://opcfoundation.org/UA/RuntimeNamespace/<GUID>”.

4.6.3 TaskHandles

The OPC UA *Cancel* service is mapped to a REST paradigm. If an operation takes a long time and a client should be able to cancel the request, a server responds to the request with the HTTP Statuscode 201 (Created). The Location header shall be set to the newly generated **TaskHandle**-*Object* (see Figure 4.9).

A TaskHandle-*Object* can offer an event that informs the client if the task is done. In addition, a client can also fetch the *Boolean-Value-Attribute* of the **ResponseIsReady**-*Property* to learn about the status of the request. If the response is ready, a client can fetch the response from

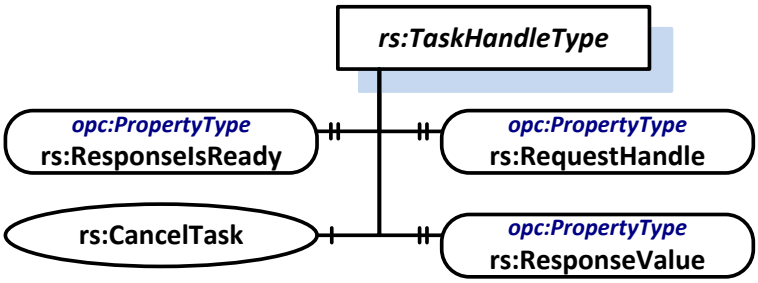


Figure 4.9 – TaskHandleType-*ObjectType*.

the **ResponseValue**-*Property* by using the *Read* service. To abort the request a client is able to use the **CancelTask**-*Method*. The *NodeId* of the **CancelTask**-*Method* shall be standardized. This ensures that a native OPC UA client is able to abort the task with preprogrammed knowledge. The **RequestHandle**-*Property* is based on the *RequestHeader* and can also be used to cancel more than one task (not further described in this work). A server shall always have a policy that includes the maximum availability time for a response. After this time the server is permitted to free resources by deleting the TaskHandle-*Object*. A successful delete request on the TaskHandle-*Object* shall have the same effect as the CancelTask-*Method* execution.

4.6.4 Register Nodes

The *RegisterNodes* service was introduced by the OPC Foundation, to optimize the reoccurring access to a given *Node*. One typical optimization on the server-side is to assign an additional numeric *NodeId* to the *Node* if the canonical *NodeId* is, for example, a large string-based *NodeId*. Of course, a client is already able to use the batch *Methods* of Section 4.3.2 to execute this service. However, in large-scale distributed systems optimization is often the responsibility of the server. For example, if the same *Node* is accessed by different clients it could make sense to optimize the access to this *Node* even if for a single client this makes no sense. Of course, it can also happen that a client that registered a *Node* for optimization is blocking optimization resources which could be used better for other *Nodes*. Another drawback of the *RegisterNodes*-Service is the fact, that

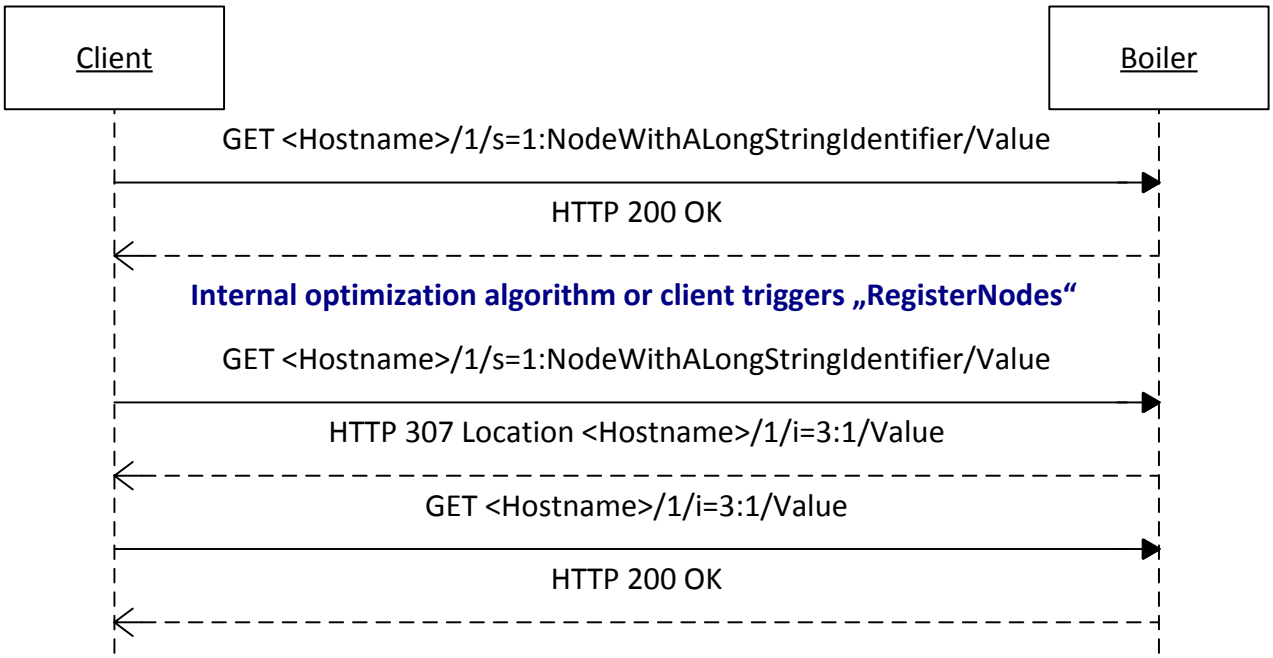


Figure 4.10 – Server-based RegisterNodes optimization.

even if one client uses the *RegisterNode* service, all other clients still use the unoptimized *NodeId* because there is no way for an OPC UA server to notify the other clients about the optimized *NodeId*. Fortunately, the HTTP protocol offers redirect concepts, which can be used for this use case. If an OPC UA server returns the 307 HTTP-StatusCode (temporary redirect), a client shall use the URL which is specified in the Location header to access this *Node*. With this approach, the responsibility for optimization can be transferred from the client to the server. Figure 4.10 shows the sequence diagram of this concept. In the example the *NodeId* is a large *String-NodeId*, which is requested on a regular basis by the client. An internal optimization algorithm of the server detects optimization potential and triggers the *RegisterNodes* service internally. On the next client request, the server answers with a temporary redirect request and specifies the new *NodeId* of the optimized resource. After that, a client can access the *Node* with the new *NodeId*. If the server decides to rollback an optimization procedure the optimized *NodeId* can simply be removed. If a client receives the 404 HTTP Error Code (not found), the canonical *NodeId* shall be used again to fetch the resource.

4.6.5 Resolve Path

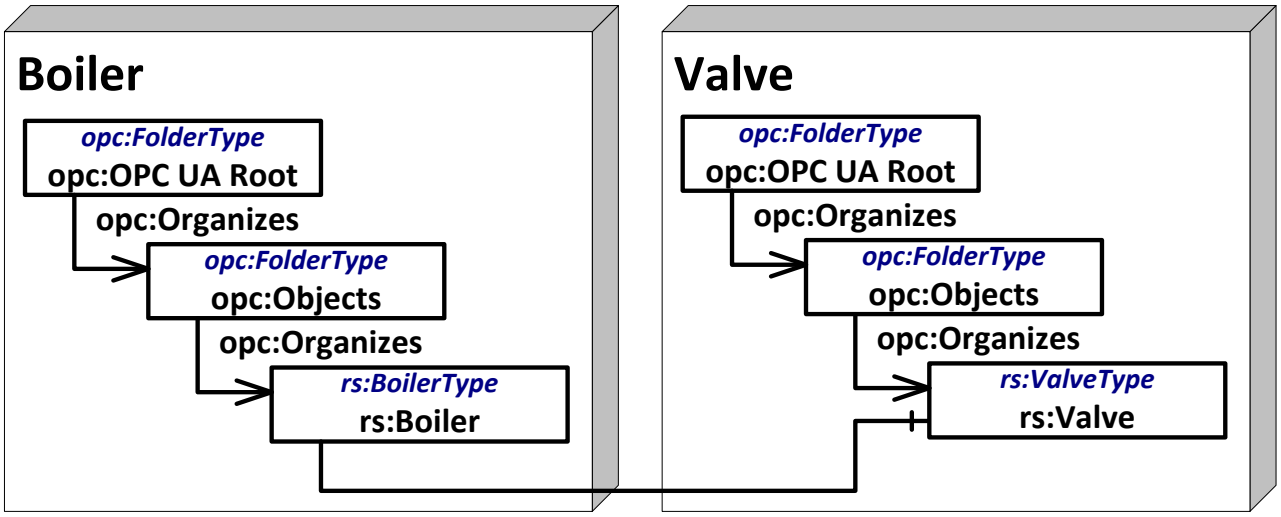


Figure 4.11 – Usage of *ExpandedNodeIds* to build distributed information models.

The ResolvePath service is newly introduced to leverage the especially useful redirect feature of the web. This service is based on the *TranslateBrowsePathToNodeIds* service of OPC UA Part 4 in combination with the *RelativePath* structure of OPC UA Part 4 Annex A. These definitions are modified in such a way, that a *RelativePath* can be formulated as an URL. Figure 4.11 shows an example information model. To fetch the NodeRepresentation of the **Boiler**-*Instance* with this service a GET request has to be issued to "$< Hostname > /1/Objects/2 : Boiler$". The server answers such a request with a HTTP 307 (Temporary Redirect) based on the first *NodeId* entry of

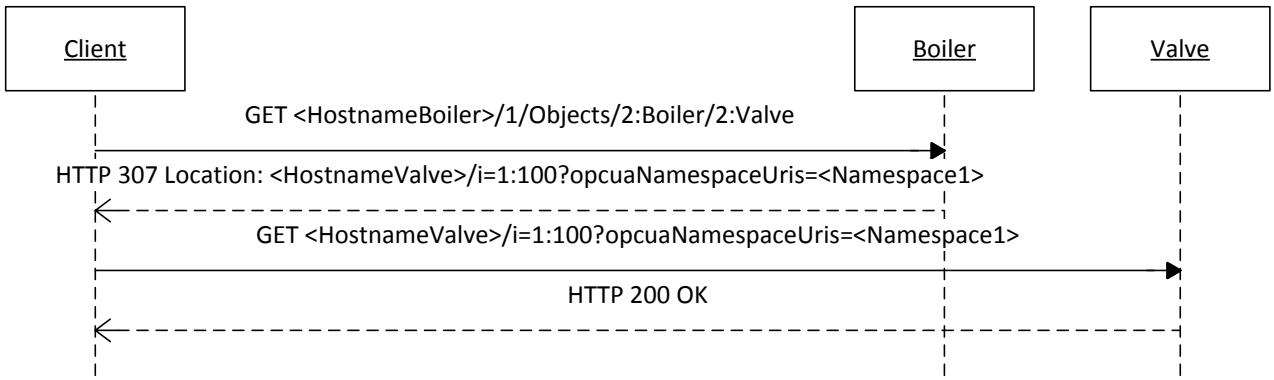


Figure 4.12 – Example of ResolvePath across different OPC UA servers.

the underlying *TranslateBrowsePathToNodeIds* service. This allows a very web-friendly usage of the OPC UA paradigm *programming against the type definition*. However, the ResolvePath service can also be easily applied to distributed OPC UA information models. For example, if the **Valve**-*Instance* of Figure 4.11 should be reached the URL only has to be extended with an additional browse step "$/2 : Valve$". In this case, the Boiler server constructs a redirect URL based on the *ExpandedNodeId* and also includes not resolved steps of the *RelativePath* (see also Figure 4.12). In the end, these concepts allows to automatically browse across several OPC UA servers with only one single ResolvePath call.

4.7 Demonstrator

The demonstrator is based on the Java OPC UA stack implementation of the OPC Foundation. The following features are available:

- *Read* and *Write* service (including batch support)
- *Browse* and *BrowseNext* service
- *TranslateBrowsePathsToNodeIds* service
- *Call* service
- *SessionlessInvoke* Base (*NamespaceUris*)
- *SessionlessInvoke* Optimized (*UrisVersion*)
- (+) MIME-Type handling
- (+) Content negotiation

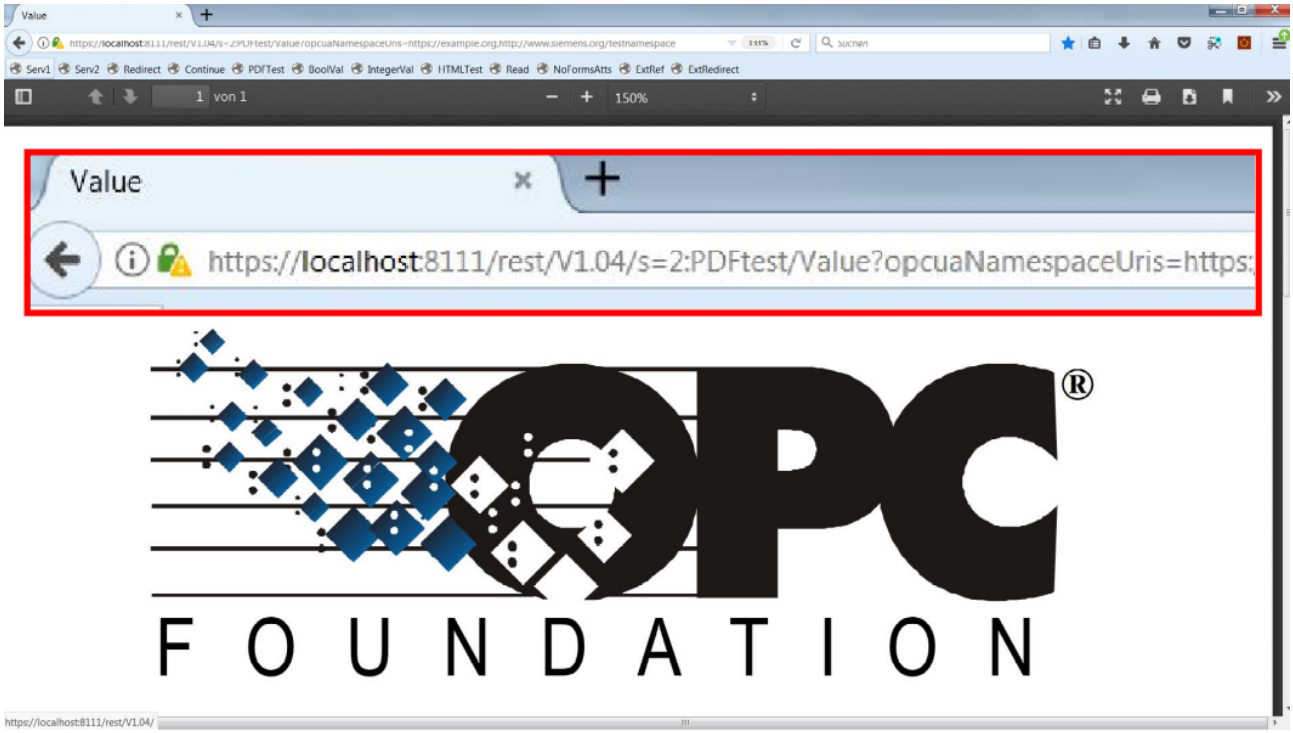


Figure 4.13 – Demonstrator - MIME-Type handling.

- (+) ResolvePath (Redirections)
- (+) ExternalReferences

Most of the services are well-known to the OPC UA community, but it should not be a surprise that the access pattern is sometimes quite different from a standard OPC UA server. For example, to collect all necessary information for a *NodeRepresentation*, more than one OPC UA service must be invoked. The services which are marked with a (+) are explained in greater detail. The term **MIME-Type handling** describes the feature to serve any arbitrary files with the correct MIME-Type, based on the *MIME-Type-DataType-Property* (see also Section 4.6.1 and Figure 4.13). **Content negotiation** is the standard way on the web for a client to request a certain representation of a resource. To be more concrete, a client can specify the encoding, for example, "application/opcua+uajson" or "application/opcua+uabinary". **ResolvePath** is introduced to offer a more comfortable way for the OPC UA concept programming against the *TypeDefinition*. If the following URL is considered "<host>/Objects/0:Server/NamespaceArray". Resolving the URL would lead to a redirect to the URL "<host>/1/i=2255", based on the first entry of the *TranslateBrowsePathsToNodeIds* service response. However, not every *Node* can be addressed by this concept because the *BrowsePath* is not unique (see also OPC UA Part 3 for further details). **ExternalReferences** can be derived from *ExpandedNodeIds* if certain additional restrictions hold. For example, a server-URI has to be a valid URL to the REST endpoint of the OPC UA server.

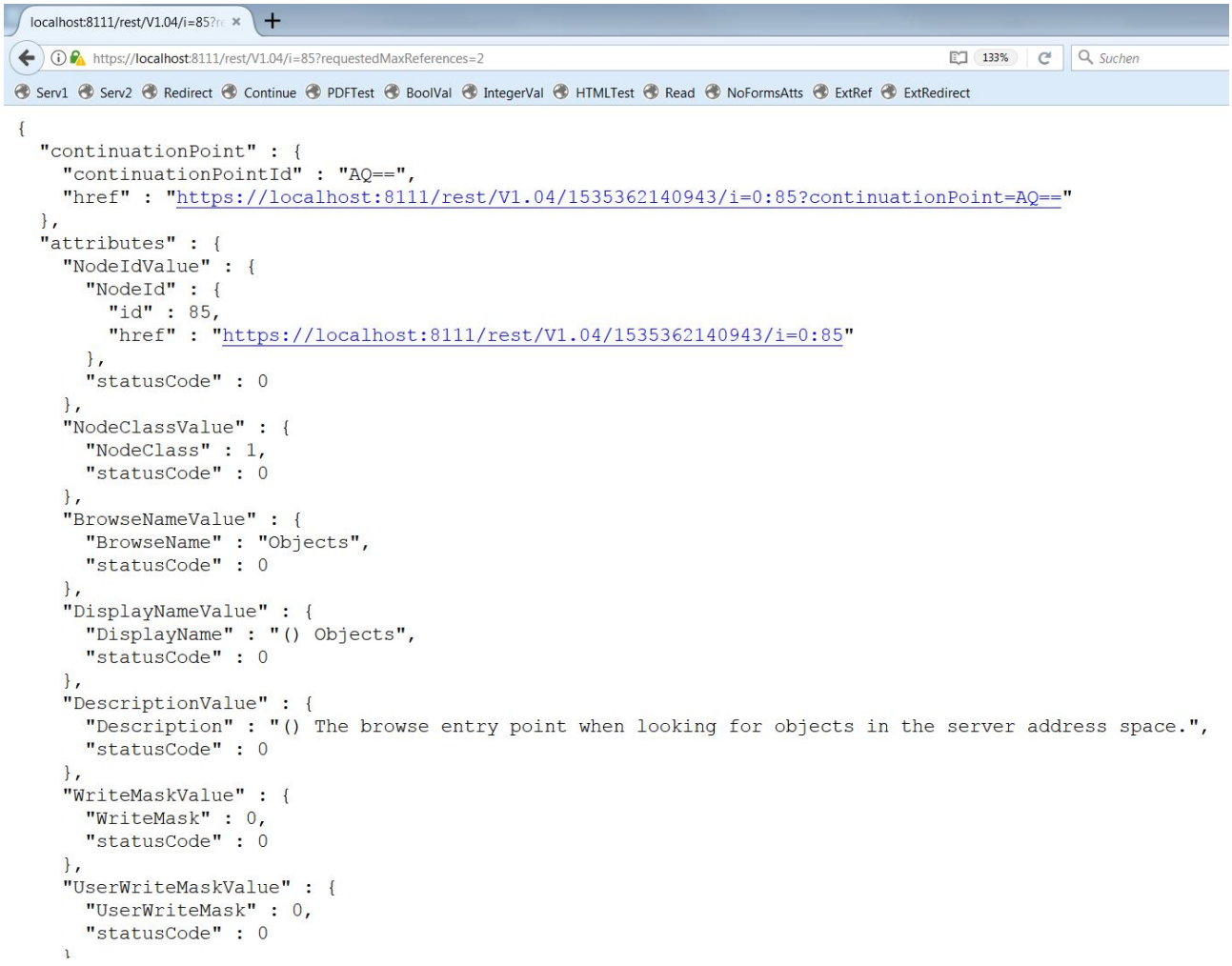


Figure 4.14 – Demonstrator - Read request including continuation point.

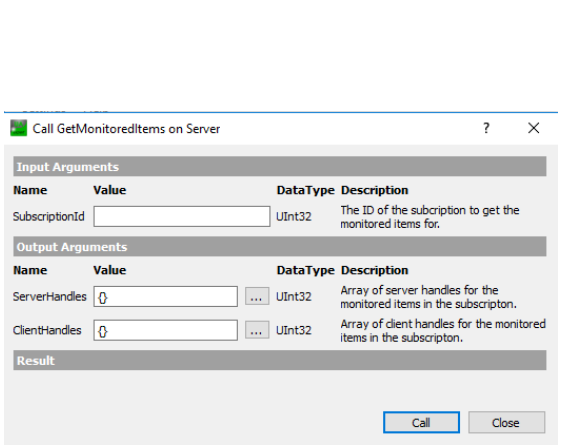
As already mentioned in Section 2.4, for a RESTful OPC UA server it is necessary to define resources and their representations. Within this chapter, several representations are introduced. Figure 4.14 depicts a representation of "application/opcua.NodeRepresentation+json" also including the concept to express continuation points. The basic structure of this representation consists of three parts: **attributes**, **references**, and **forms**. In the attributes section, all *Attributes* of a given *Node* are summarized, while the reference section offers all *References* of the *Node*.

Figure 4.15 shows that also the optional REST feature code-on-demand can be implemented with the current architecture. In the given example an HTML page is stored in the *Value-Attribute* of a *Variable-Node*. The HTML page also contains a very simple JavaScript implementation for the REST API to set a boolean value to different states. Notice that, it is also possible to embed a standard JavaScript-based OPC UA client within such an HTML page like NodeOPCUA [108].

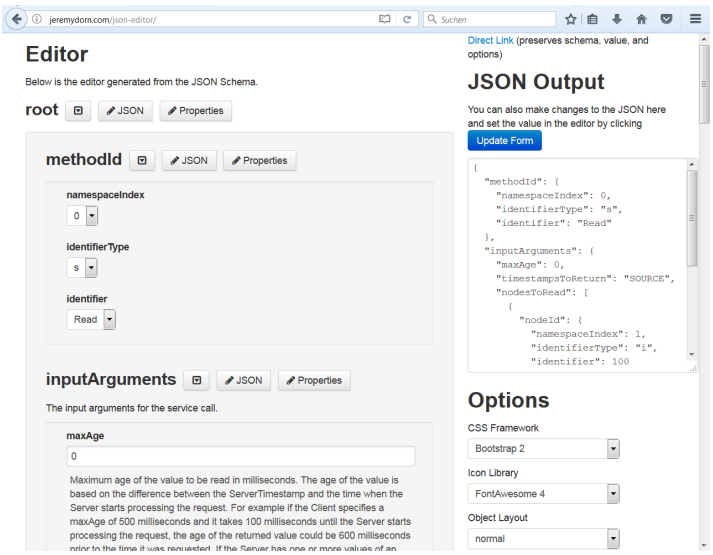
A typical graphical OPC UA Client, like UAExpert [158] (see also Figure 4.16a), often offers some kind of template if the OPC UA *Call* service is invoked by the user. This template is based

Figure 4.15 – Demonstrator - Code-on-demand example.

on well-known OPC UA properties, which are part of each *Method-Node*. However, a typical web client is not aware of these OPC UA specific conventions. Because of that, the form section also provides some additional information if OPC UA *Method-Nodes* are involved. The most important part of such a representation is explained in Section 4.4.5. The representation provides all necessary information, including the target URL, the HTTP method, and the expected Content-Type. The section requestSchema and responseSchema provides a URL to the corresponding schema description for the request/response Content-Type (see also Section A.1.4.4). The demonstrator uses JSON schema [168] for this purpose, which is well-known in the web and also offers client-side



(a) UAExpert.



(b) JSON schema.

Figure 4.16 – Demonstrator - Call service

validation. However, based on HTTP content negotiation it is also possible to serve other schema descriptions. Several different libraries already exist to automatically generate templates out of JSON schemas (see also Figure 4.16b) which can be used for easy development of REST clients to generate OPC UA *Call* requests.

4.8 Evaluation

Within this section, the proposed approach is evaluated against the formulated research challenge (C1) interoperability on the transport layer. Within Section 3.1 the evaluation metrics for the given research challenge are formulated and existing research approaches are checked against these metrics. Based on this evaluation three unsolved problems are identified in the existing research approaches: First, statelessness is not addressed correctly in the actual research. Second, most of the current research struggles with HATEOAS (uniform interface) principles. Third, a lot of OPC UA services cannot be accessed with REST APIs.

Table 4.5 shows the evaluation results of the proposed approach in this thesis. Similar to the existing research **Client-Server**, **Cache**, and **Layered system** is covered very well through the usage of already existing functions within OPC UA. The **Uniform interface** requirement is covered also very well through the introduction of an HTTP mapping (Section 4.4), a hypermedia representation (Section 4.4.5), and further RESTful features (Section 4.6) like, for example, the support of web browsers. **Statelessness** is solved through the introduction of a new standardized service into OPC UA with the name *SessionlessInvoke* (Section 4.2.2). The optional **Code-on-**

Requirements \ Research approaches	this thesis
Client-Server	++
Cache	++
Uniform interface	++
Layered system	++
Statelessness	++
Code-on-demand	++
Sum (18):	18

Table 4.5 – Requirements and evaluation for OPC UA web access (this thesis).

.

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

demand feature is addressed also very well through the usage of correct MIME-Types, which can be used to, for example, deliver HTML pages with JavaScript code.

Table 4.6 shows how many of the different OPC UA services are covered by this thesis. Besides the services of Table 4.6, this thesis introduces also new services like ResolvePath, ModifyReferences as well as a concept for group subscriptions (Section 4.5) to provide a more RESTful user experience for web developers.

In conclusion, this thesis presents a solution for the research challenge (C1) interoperability on the transport layer and contributed the necessary extensions of OPC UA to the OPC Foundation. The three identified open research points: Statelessness, HATEOAS support, and service coverage are successfully solved within this thesis. Some aspects of this work, like the *Sessionless-Invoke* service, were contributed to the OPC UA standard as part of the V1.04 release. In contrast to the HTTP API of the OPC Foundation, which maps all requests to HTTP POST with standard OPC UA-specific serialization (see also Section 4.2.3), in this thesis a REST API is defined according to the work of Fielding [46]. The REST API makes use of the correct semantics behind HTTP operations and also introduces self-describing resource representations (including the HATEOAS concept). Finally, the mapping of this thesis allows, for example, standardized web tools like web browsers to easily access OPC UA. The same is not true for the HTTP API of the OPC Foundation. However, the REST design pattern is most successful in the human web because human-controlled clients can easily adapt to changes in the resource representations without the need to modify the codebase. The

Research approaches / OPC UA Services	this thesis
Discovery (3)	X
SecureChannel (2)	X
Session (4)	X
NodeManagement (4)	X
View (5)	X
Query (2)	X
Attribute (4)	X
Method (1)	X
MonitoredItem (4)	X
Subscription (5)	X
Sum (34):	34

Table 4.6 – Coverage evaluation of OPC UA services for web access (this thesis).

.

Legend: X = mapped (1), " " = not mapped (0), NA = Not Applicable (0)

same is not true for the classic machine web, which is the main use case of OPC UA. In this case, a new link in a resource representation or a new step in a HATEOAS-based state machine could also lead to problems for the clients. If, for example, Amazon introduces an additional step in the order process to select a common delivery date for all products a human could easily complete this additional step in contrast to a machine. In the future, AI-based machines might be able to also execute such jobs in combination with the application of Semantic Web concepts and thus REST might gain additional advantages in the Industrial Internet of Things. But as long as this point is not reached one huge advantage of the REST design pattern cannot be used to gain all the corresponding benefits. Another disadvantage of REST is large resource representations, which are necessary for self-descriptive messages. Of course, such representations can be optimized but some parts always have to remain to support new generic clients. In a highly optimized embedded environment, each additional optional byte also generates higher hardware requirements and, therefore, leads to higher prices. Nevertheless, also this issue might be solved in the future because embedded devices become more and more powerful also in typical OPC UA domains.

5 SEMANTICS OF OPC UA INFORMATION MODELS

This chapter introduces a concept to translate OPC UA based semantics into the formal semantics of the OWL universe. Similar to the REST architecture also formal semantics can be found in a lot of different domains like healthcare [88], smart grid [120], building technology [149], and life science [132]. Based on that it is not surprising that already several translation technologies emerged to ease the transformation of arbitrary data formats into Semantic Web compatible formats [103, 100, 102, 25, 14, 148]. In the area of manufacturing formal semantics could be used to greatly simplify use cases like predictive maintenance [32], plug and produce [118, 42, 141], quality inspection [35], optimization of resource consumption [18, 79], documentation / knowledge management [171], and skill-based engineering [172, 91, 99, 69, 40]. However, a typical factory consists of thousands of devices with sometimes hundreds of data points per device, which leads to high requirements on data throughput. Luckily, also in the area of large data processing different platforms have emerged [1, 74, 86, 169, 87]. Eventually, Semantic Web technology can be considered suitable for the demands of the manufacturing domain. Parts of this chapter are also published in [136].

The remainder of this chapter breaks down the OPC UA to OWL mapping into smaller building blocks:

Section 5.1 presents parts of the OPC UA Meta-Layer and the corresponding translation into OWL classes. All further classes and generation schemes are subtypes of the classes introduced in this section.

Section 5.2 exemplifies the transformation of OPC UA *Attributes* into OWL data properties and OWL annotation properties including, for example, the corresponding domain and range restrictions.

Section 5.3 provides the mapping of basic OPC UA *DataTypes* to OWL data types. Furthermore, a concept is outlined how OPC UA *Structures* and OPC UA *Enumerations* can be translated.

Section 5.4 introduces the relation between OPC UA *ReferenceTypes* and OWL object properties. In addition, the section outlines the concept of OWL object properties for OPC UA *BrowseNames*.

Sections 5.5 - 5.6 summarize the transformation rules for OPC UA *ObjectTypes* and OPC UA *VariableTypes*. Both OPC UA concepts are transformed to OWL classes including general restrictions of OPC UA *Attributes* and type-specific *ModellingRule* restrictions.

Sections 5.7 - 5.8 focus on the mapping of *Object InstanceDeclaration* and *Variable InstanceDeclaration*. Similar to OPC UA *Types* also *InstanceDeclarations* are mapped to OWL classes with restrictions.

Section 5.9 presents how OWL classes with restrictions can be generated from OPC UA *Methods*. The concepts are similar to *Object InstanceDeclarations* but also cover some details, which are only valid for OPC UA *Methods*.

Section 5.10 exemplifies the mapping of OPC UA *Instances* to OWL individuals. In this section all the previously defined concepts are instantiated and used to generate an OWL graph out of OPC UA information models. In addition, the usage of concepts like OWL punning is further explained.

Section 5.11 provides a concept to model restrictions in the context of the OPC UA *ValueRank-Attribute* through OWL. This part of the mapping is mainly addressing the validation use case and could be omitted for most other use cases.

Section 5.12 gives a brief overview of the Java-based Prototype, which transforms OPC UA *NodeSet-Files* into OWL ontologies.

Section 5.13 evaluates the findings and contributions of this chapter against the corresponding research challenge. Furthermore, the section outlines further ideas and thoughts about OPC UA semantics.

5.1 Class Meta-Layer

A Protégé view of the class meta-layer OWL ontology is shown in Figure 5.1. The underlying technical details for URI generation can be found in Annex A.2. The class hierarchy depicts the results of transforming the basic concepts of the meta-layer such as **Base**, **Object**, **Variable**, **Method**, **View**, **VariableType**, **ObjectType**, **DataType**, **ValueRankHelper**; and the Base-Layer and Companion-Layer (sub-concepts of the respective meta-layer concepts). The lower right corner of Figure 5.1 shows the results for transforming parts of the *Mandatory VariableInstanceDeclaration* restrictions. Notice, that in the transformation process also OWL object properties for the *BrowseName* and classes for the *InstanceDeclarations* are generated (e.g., "actPos" and "cmdPos" of Figure 5.1). *InstanceDeclarations* with the "CncPositionVariableType" as *Type* are depicted in the lower left corner (e.g., "CncChannelType_PostTcpBcsA"). *Type-Attributes* are displayed at the upper

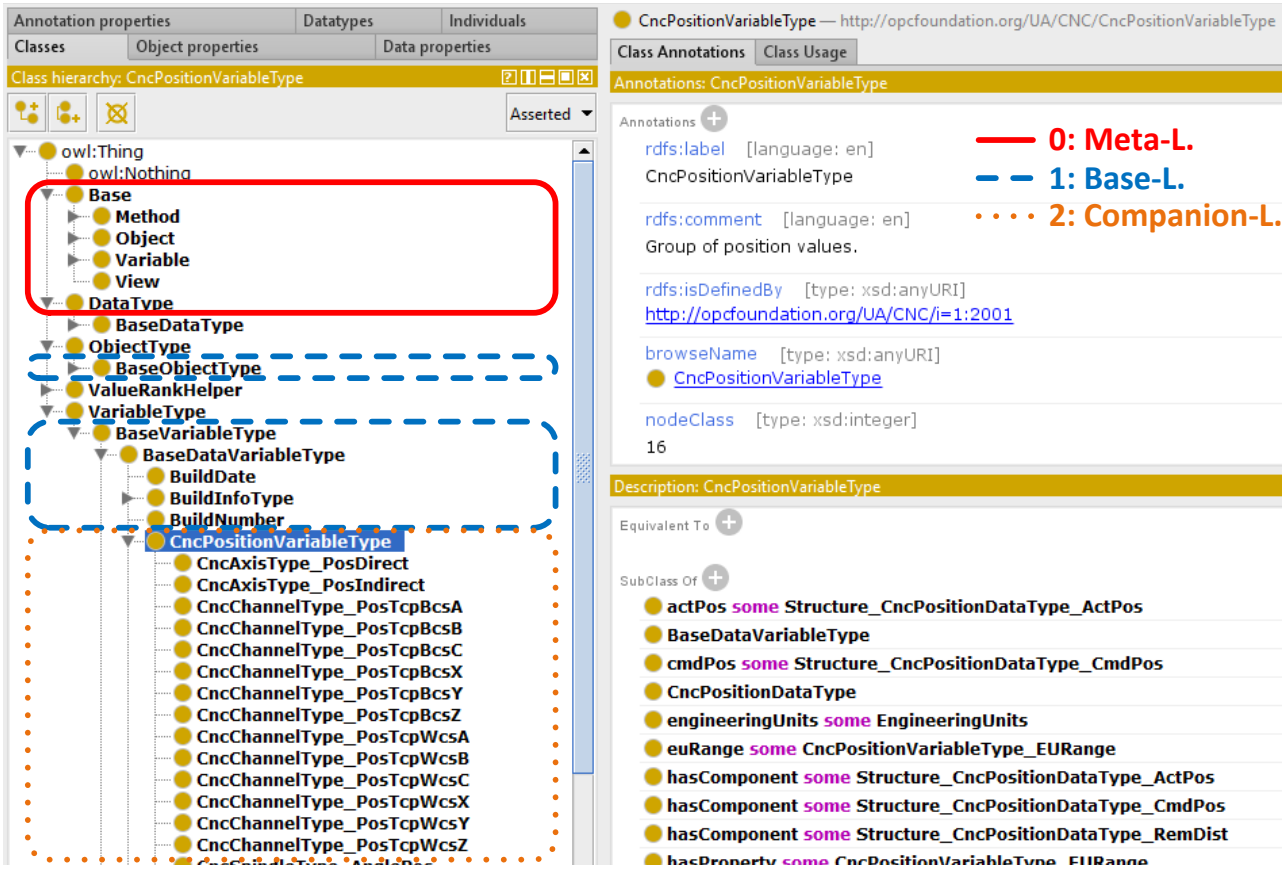


Figure 5.1 – Protégé-View [124] of the generated OWL ontology [136].

right corner of Figure 5.1. Some of the *Type-Attributes* are directly mapped to some well-known annotation properties, for example, the OPC UA *NodeId* to rdfs:isDefinedBy.

Figure 5.2 exemplifies the restrictions of the *Object* OWL class meta-layer concept. OPC UA introduces several distinct restrictions around the eight different *NodeClasses*. In the case of the *Object-NodeClass*, the usage of certain *Attributes* is restricted. Furthermore, in OPC UA it is also possible to define restrictions on what kind of *References* can be used in combination with *NodeClasses*. It is even possible to define such restrictions on *Types* instead of *NodeClasses*. An example of such a restriction is visible in Figure 5.2 based on the inverse *Reference* of the *HasComponent-ReferenceType*. Notice, that even due to the fact that restrictions on *References* are modeled partly on OWL classes, most of these restrictions are attached to the *ReferenceType* and not to the *Object-NodeClass* by OPC UA.

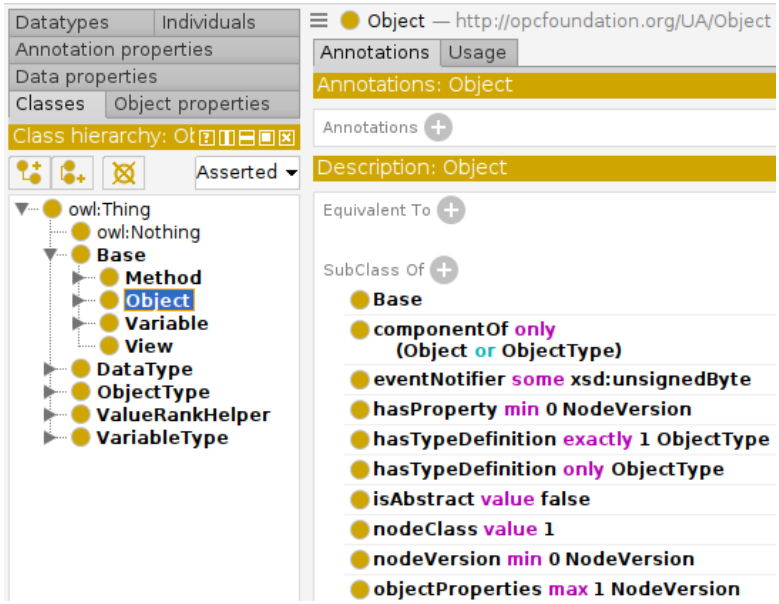


Figure 5.2 – Restrictions on the *Object*-meta-class.

5.2 Attribute mapping

OPC UA-*Attributes* are mapped to annotation and data properties depending on the underlying OWL concept. For example, *VariableTypes* are mapped to OWL classes with OWL annotation properties, while most of the *Variables* are be mapped to OWL individuals in combination with OWL data properties. Table 5.1 defines the mapping in the following way: The **Attribute** column contains the name of the *Attribute*. Column **AP** marks if this *Attribute* is exposed as OWL annotation property (*Namespace* prefix TA of Table A.22) and **DP** marks OWL data properties (*Namespace* prefix IA of Table A.22). Notice, that most of the *Attributes* are modeled as OWL annotation and data properties. The **F** column shows if the given annotation or data property shall be functional. Most of the *Attributes* are functional and, therefore, exactly one value for the given *Attribute* exists for each *Node*. In contrast, the **DisplayName**-*Attribute* is not functional because there might be several different statements for different languages. Another interesting *Attribute* is the *NodeId*. Even due to the fact that the *NodeId* is defined as functional, one *Node* could have several different *NodeIds* assigned. This might come as a surprise because one characteristic of OPC UA is, that the *NodeId* is the only unique address within an OPC UA graph. One use-case can be extracted out of OPC UA Part 4, where the *RegisterNode-Service* is introduced. This service optimizes access to a given *Node* and also provides an optimized numeric *NodeId*. The *Read-Service* of OPC UA always returns the canonical *NodeId*. This results in the fact that it is not possible to retrieve information about additional *NodeIds* for the same *Node* and because of that the *NodeId* is defined functional. However, in the OWL mapping additional *NodeIds* could be reflected through an additional OWL

Table 5.1 – OPC UA Attributes to OWL mapping.

Attribute	AP	DP	F	Domain	Range
AccessLevel	X	X	X	Var	xs:unsignedByte
AccessLevelEx	X	X	X	Var	xs:unsignedInt
ArrayDimensions	X	X	X	Var or VarT	xs:string
AccessRestrictions	X	X	X	All NodeClasses	xs:unsignedShort
BrowseName	X	X	X	All NodeClasses	xs:anyUri
ContainsNoLoops	X	X	X	View	xs:boolean
DataType	X	X	X	Var or VarT	xs:anyUri
DataTypeDefinition	X		X	DataT	xs:string
Description	X	X		All NodeClasses	rdf:PlainLiteral
DisplayName	X	X		All NodeClasses	rdf:PlainLiteral
EventNotifier	X	X	X	Obj or View	xs:unsingedByte
Executable	X	X	X	Met	xs:boolean
Historizing	X	X	X	Var	xs:boolean
InverseName	X			RefT	rdf:PlainLiteral
IsAbstract	X		X	VarT or ObjT or RefT or DataT	xs:boolean
MinimumSamplingInterval	X	X	X	Var	xs:double
NodeClass	X	X	X	All NodeClasses	xs:int
NodeId	X	X	X	All NodeClasses	xs:anyUri
RolePermissions	X	X	X	All NodeClasses	xs:string
Symmetric	X		X	RefT	xs:boolean
UserAccessLevel	X	X	X	Var	xs:unsignedByte
UserExecutable	X	X	X	Met	xs:boolean
UserRolePermissions	X	X	X	All NodeClasses	xs:string
UserWriteMask	X	X	X	All NodeClasses	xs:unsignedInt
Value	X	X	X	Var or VarT	
ValueRank	X	X	X	Var or VarT	xs:int
WriteMask	X	X	X	All NodeClasses	xs:unsingedInt
NOTE: Var = *Variable*, VarT = *VariableType*, DataT = *DataType*, Obj = *Object*, Met = *Method*, RefT = *ReferenceType*					

data and annotation property. The **Domain** column defines the permitted *NodeClasses* for each *Attribute*, while the **Range** column defines the OWL data type. Notice, that while a reasoner is able to validate **Domain** and **Range** restrictions for OWL data properties the same does not apply for OWL annotation properties.

5.3 DataType mapping

Within this section, several aspects of the DataType mapping are discussed. The technical details for the XML data type mapping can be found in Annex A.2.3. In the beginning, Section 5.3.1 provides the basic mapping of OPC UA-*DataType* hierarchies to OWL class hierarchies. Section

5.3.2 focuses on the special handling for *Structure-DataTypes* of OPC UA, while Section 5.3.3 explains the mapping rules for the *Enumeration-DataType*.

5.3.1 Basic concepts

DataTypes are exposed as own class concepts for several reasons: (1) *DataTypes* are one place where semantics is stored (e.g., for complex *Variables*, see also OPC UA Part 3); (2) *DataTypes* restrict the *Value-Attribute* of *Variables* and *VariableTypes*; (3) The *DataType* hierarchy is used for restriction inheritance. The basic concept behind the transformation of *DataTypes* to OWL is depicted in Figure 5.3. The OWL SubClass axioms on the right side of Figure 5.3 are some examples of possible validation restrictions. If, for example, a *Variable* has the *DataType Int16* the value of the *DataType-Attribute* is defined as well as the range of the *Value-Attribute* (in this case xsd:short). Furthermore, more complex restriction patterns are also possible based on RegEx patterns or on number ranges. To also cover the semantics of the *IsAbstract-Attribute* the union concept of OWL can be used (see also Figure 5.3 right side - SubClass Of (Anonymous Ancestors)). This concept only works if the *DataType-Attribute* is not defined for abstract *DataTypes*. However, if the *DataTypes* are further subtyped the OWL union concept is also necessary on the *DataType-*

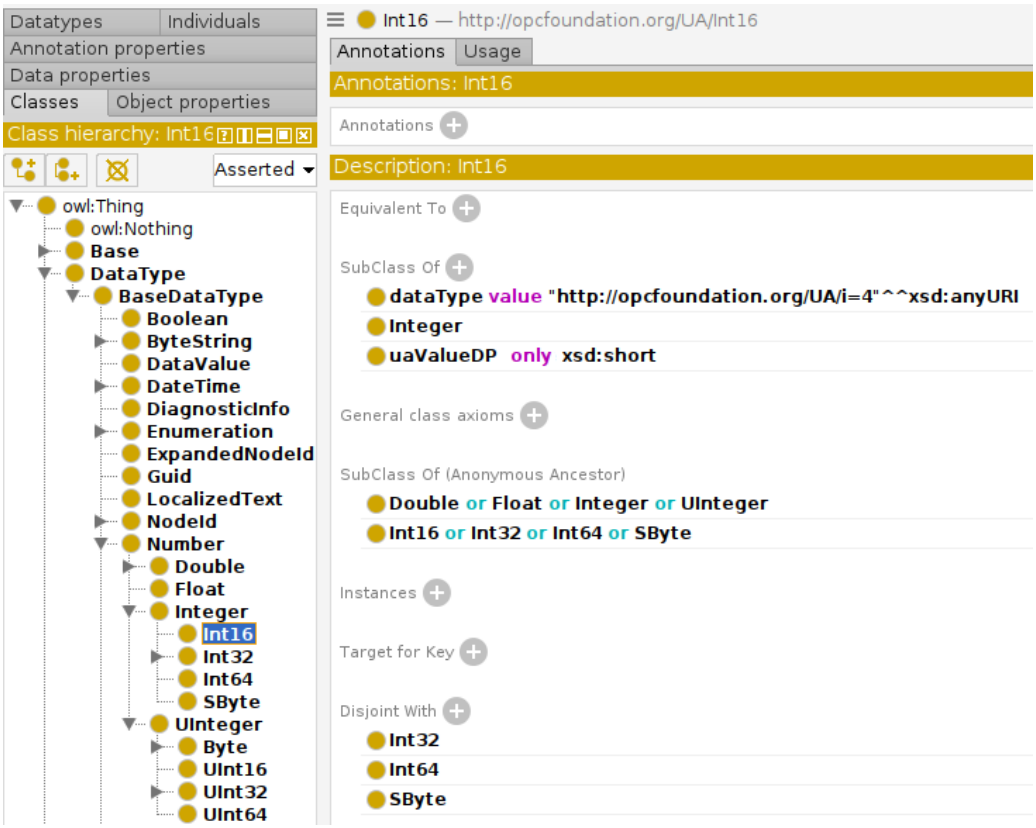


Figure 5.3 – Class concept for *DataTypes*.

Attribute. In conclusion, while the restrictions on the *Value-Attribute* proofed to be very useful the restrictions on the *DataType-Attribute* in combination with the *IsAbstract-Attribute* concept increases the complexity of the ontology by a significant amount and could be more efficiently validated with simple SPARQL queries.

5.3.2 Structures

Besides simple *DataTypes* like *Int16*, OPC UA also defines more complex *DataType* like *Structures*. *Structures* can be used to combine *DataTypes* to a more complex structure. For example, the **CncPositionDataType** of [162] consists of three values: The **ActPos**, which contains the current position value; The **CmdPos**, which contains the setpoint position value; The **RemDist**, which contains the remaining distance. This *Structure-DataType* can be used to easily access all three values in one single transaction context. However, for some use cases, it might be enough to only retrieve the **ActPos** without the other values. Such a use case is addressed through complex *Variables* like the **CncPositionVariableType** of Table 5.2. The **CncPositionVariableType** introduces several *InstanceDeclarations* with *BrowseNames*, which are identical to the *Structure* element names of the **CncPositionDataType**. These *InstanceDeclarations* are all subtypes of the very generic *BaseDataVariableType* without further semantic refinement. The reason for that is another implicit

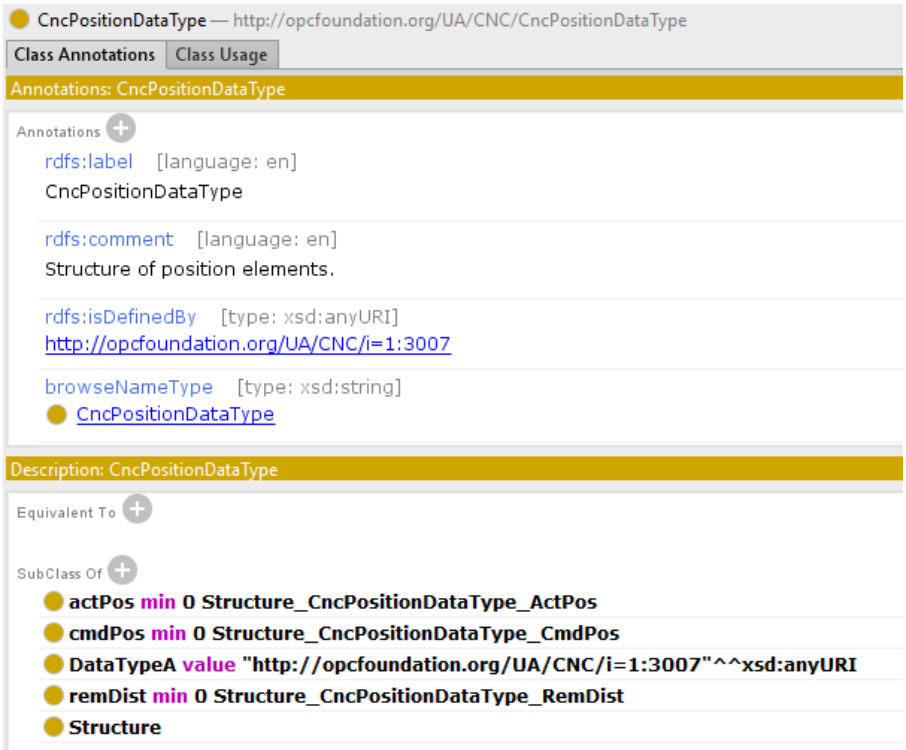


Figure 5.4 – Class concept for **CncPositionDataType**.

Table 5.2 – **CncPositionVariableType** definition (see also [162]).

Attribute	**Value**			
Browse-Name	CncPositionVariableType			
IsAbstract	False			
References	**BrowseName**	**DataType**	**TypeDefinition**	**MRule**
Subtype of the BaseDataVariableType				
HasComp.	ActPos	Double	BaseDataVariableType	Mand.
HasComp.	CmdPos	Double	BaseDataVariableType	Mand.
HasComp.	RemDist	Double	BaseDataVariableType	Mand.
...	...	...	...	...

concept of OPC UA. In the case of complex *Variables*, the semantics of the element *InstanceDeclaration* is defined by the *Structure-DataType* and is assigned through the identical element name. This concept is reflected in the OWL mapping through the generation of *InstanceDeclarations* classes for each *Structure-DataType* (see also Section 5.8). These *InstanceDeclarations* classes are referenced by the corresponding *Structure-DataType* with an optional *ModellingRule* (see also Figure 5.4). The *ModellingRule* can be easily overridden and set to, for example, mandatory by the **CncPositionVariableType**. In addition, also OWL object properties are generated for each structure field name in the corresponding *Namespace*.

5.3.3 Enumerations

Another special *DataType* in OPC UA are *Enumerations*. *Enumerations* in OPC UA restrict the *Value-Attribute* to a set. This restriction can be easily modeled with the OWL axioms of Table 5.3. However, as already stated in Section 5.3.1 the union concept should be used with caution and typically leads to a very high computational complexity in combination with several other properties of the OPC UA ontology generation rules. Of course, also in this case, the OPC UA subtyping rules for *Enumerations* are reflected by the OWL reasoner. In the end, also these rules could be checked more efficiently with SPARQL statements.

Table 5.3 – Enumeration axioms.

Concept name	**Extract of the axiom**
Enumeration	SubClassOf((uaValueDP value 0) or (uaValueDP value 1) or (uaValueDP value 2))

5.4 ReferenceType mapping

Figure 5.5 shows the basic concepts behind how *ReferenceTypes* are translated to OWL object properties. As URI the *BrowseName* is used to simplify the tool-based ontology handling. OPC UA enforces that *BrowseName* of *ReferenceTypes* shall be unique, which guarantees that no naming conflicts arise in the *ReferenceType* hierarchy. However, in OWL the coding guidelines recommend that OWL object properties start with lower case. This can be easily achieved during transformation and further reduces possible naming conflicts with other *Type* hierarchies if for OWL class concepts upper case is enforced (see also Section 5.5 and Section 5.6). In OPC UA one *ReferenceType* can be used to define two directions. For example, the *HasChild-ReferenceType* defines also the inverse direction *ChildOf*. This is done through the *Symmetric-Attribute* in combination with the *InverseName-Attribute*. If the *Symmetric-Attribute* is set to "true", this automatically implies that the *ReferenceType* has the same meaning in both directions, for example, "connectedTo". In this case, the *InverseName-Attribute* shall be omitted. In the other case, the *Symmetric-Attribute* is set to "false" and an *InverseName* shall be provided for non-abstract *ReferenceTypes* (e.g., *HasChild* and *ChildOf*). In OPC UA each edge instance in the graph can be annotated with additional information like the direction of a given *ReferenceType*. RDF does not support to annotate object properties for edge instances on an individual basis. In RDF all object properties share the same information. In contrast to RDF, labeled property graphs would also allow to annotate instances of edges. However, also in OPC UA the triple *SourceNode*, *ReferenceType*, and *TargetNode* is unique. Furthermore, the properties

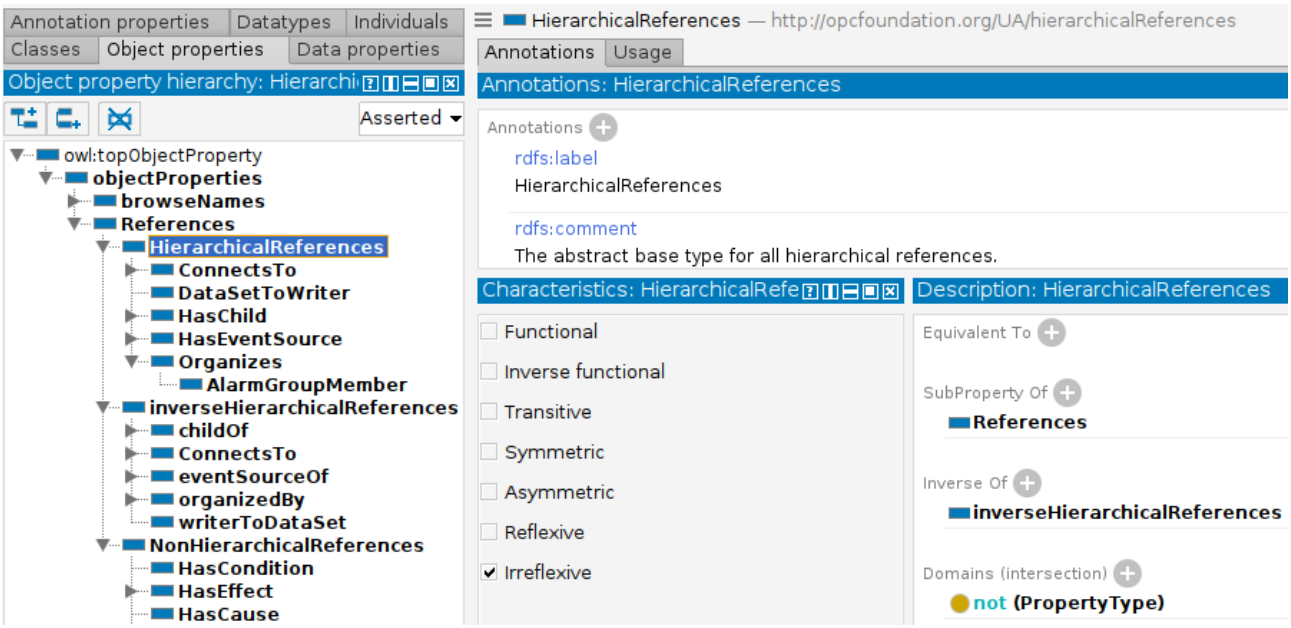


Figure 5.5 – Object property concept for *ReferenceTypes*.

for edge instances are only specified through the OPC UA specification and cannot be extended. In the end, this allows the introduction of a generic mapping to RDF triples for OPC UA. The basic concept to map *ReferenceTypes* to RDF is to generate two object properties for all *Non-Symmetric ReferenceTypes*. Furthermore, an inverse hierarchy named **inverseHierarchicalReferences** (see Figure 5.5 left side) has to be introduced. The inverse hierarchy has the same structure as the original hierarchy. *Symmetric References* are part of both hierarchies, mainly due to the fact that OPC UA does not forbid to subtype *Symmetric ReferenceTypes* with *Non-Symmetric ReferenceTypes*. In general, *Symmetric* references are defined only in the forward direction according to OPC UA Part 4. Furthermore, the two distinct OWL object properties of non *Symmetric ReferenceTypes* shall be connected through an inverse of axiom. It is also possible to assign further annotation properties to easily identify the generated OWL object properties and based on that the direction of the *Reference* on *Instances*.

Besides the *ReferenceType* hierarchy, OPC UA also defines further semantics for *ReferenceTypes*. Sadly, most of the semantics is defined in a textual way and cannot be automatically extracted out of the machine-readable OPC UA *NodeSets*. For example, OWL range and domain restrictions for *ReferenceTypes* are modeled in a textual way only and have to be added manually to the ontology (see also Figure 5.5 right side). Furthermore, OPC UA defines semantics for *ReferenceTypes* similar to OWL functional, inverse functional, irreflexive, and symmetric axioms. Also in this case, only the symmetric axiom is machine-readable. OPC UA also defines the notion of loop freedom (see also OPC UA Part 3 *HasChild-ReferenceType*). This means that if *Node* "A" is the starting point and if only *References* with this characteristic are followed, it is not allowed to reach *Node* "A" again. The reason for that are the roots of OPC UA in the embedded world, where devices might not have enough memory to detect loops during graph navigation. OWL does not have a corresponding axiom for such a characteristic on an OWL object property, but such a restriction could be modeled through the combination of the transitive and irreflexive axiom. Nevertheless, certain feature combinations are prohibited to ensure decidability of reasoning in general [58].

In addition to *ReferenceTypes*, also certain *BrowseNames* are translated into OWL object properties (see also Figure 5.5 browseNames OWL object property). As URI the *BrowseName* is used to simplify the tool-based ontology handling. Similar than for *ReferenceType* OWL object properties also the *BrowseName* OWL object properties shall start with lower case. Notice, that naming conflicts between *BrowseNames* are not resolved as long as the semantics still apply. These object properties have OWL functional and irreflexive characteristics. Of course, it would also be possible to define domain and range restrictions. However, the generation rules for this concept can be considered independent of the underlying OPC UA model and so violations can already be detected during transformation. Further generation rules are introduced in Section 5.7.

5.5 ObjectType mapping

Figure 5.6 shows an example *ObjectType* after the mapping to OWL is applied. Each *ObjectType* generates an OWL class and is integrated into the class hierarchy (see left side of Figure 5.6). As URI the *BrowseName* is used to simplify the tool-based ontology handling (starting with upper case according to OWL coding guidelines). In most information models the *BrowseName* of *Types* is unique. However, Section A.2.1 also introduces an alternative URI generation schema if conflicts are detected. The OWL class hierarchy can be automatically generated based on *HasSubtype-References* of OPC UA. The *Attributes* of *ObjectTypes* are modeled with OWL annotation properties (see right side of Figure 5.6). The main reason for modeling these *Attributes* as annotation properties is grounded in the instantiation of OPC UA *Types*. Each *Instance* is allowed to change most of the *Attribute* values and sometimes even has to change some of them (e.g., *NodeClass-Attribute*). In the end, this does not allow to define the value of the *Attributes* as OWL value class expression because in this case the value could not be changed any longer. This leaves annotation

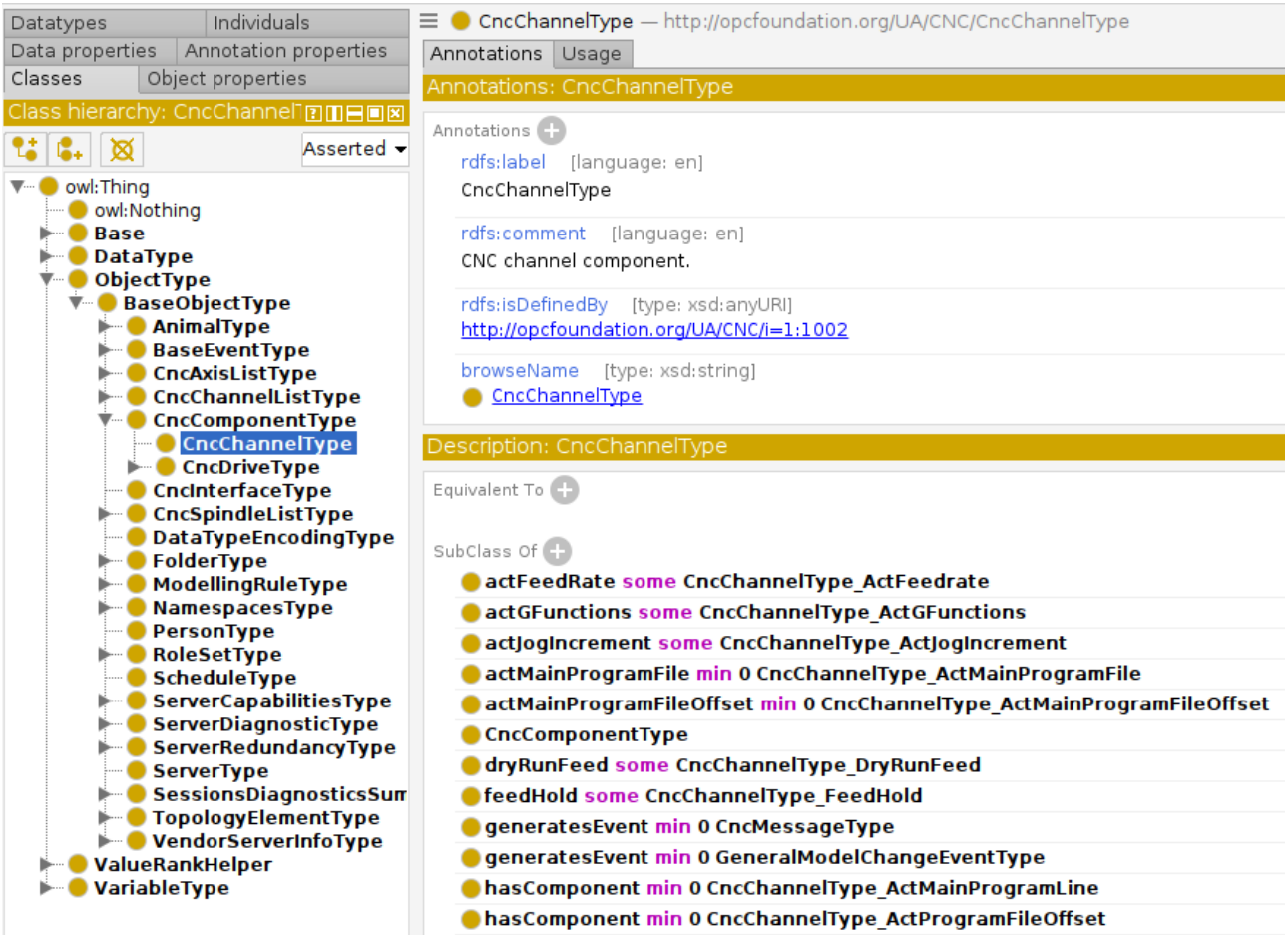


Figure 5.6 – Class concept for *ObjectTypes*.

properties as the last possible choice to store the values of the *Type-Attributes*. Besides values of *Type-Attributes*, an *ObjectType* fulfills another purpose in regards of *Attribute* subtype restrictions. For example, it is possible to define optional *Attributes* as mandatory and enforce this restriction on each subtype instance (not shown in Figure 5.6). The lower right side of Figure 5.6 displays some of the OWL subclass axioms. The main purpose of these axioms is to model *InstanceDeclaration* restrictions based on OPC UA *ModellingRules*. OPC UA defines several different combinations of *ModellingRules* in combination with the *ObjectType*. However, new *ModellingRules* could be added through extensions of the specification. In the following, the *ModellingRules* of the core specification are further investigated and classified into concepts (see also Table 5.4):

- **Concept 1**: *ModellingRule-Mandatory* for *Variables*, *Objects*, and *Methods*
- **Concept 2**: *ModellingRule-Optional* for *Variables*, *Objects*, and *Methods*
- **Concept 3**: *ModellingRule-MandatoryPlaceholder* for *Variables* and *Objects*
- **Concept 4**: *ModellingRule-OptionalPlaceholder* for *Variables* and *Objects*
- **Concept 5**: *ModellingRule-MandatoryPlaceholder* for *Methods*
- **Concept 6**: *ModellingRule-OptionalPlaceholder* for *Methods*

Concept 1 defines that the *InstanceDeclaration* is mandatory for all *Instances* of the given *Type*. This is reflected through several axioms in Table 5.4. The first two axioms define mandatory relations to the *InstanceDeclaration* through two different OWL object properties. One OWL object property reflects the *ReferenceType*, which shall be used to connect both *Instance-Nodes*. Notice, that it would also be possible to replace the *ReferenceType* with a subtype of the *ReferenceType* on an *Instance* as specified by OPC UA (see also Section 5.4). The other OWL object property reflects the *BrowseName* of the *InstanceDeclaration*. This is done to support the OPC UA concept "programming against the *TypeDefinitionNode*" and is especially useful for SPARQL-based queries. The last axiom is optional and ensures that an *Instance* has to reference the same OWL individual with both OWL object properties. This axiom can also be used to ensure further constraints on *Instances* (see also OPC UA Part 3 6.4.3 - Constraints on an Instance). The successful enforcement of such a restriction depends heavily on OWL different individuals axioms and, therefore, might not be very useful in practice.

Concept 2 defines that the *InstanceDeclaration* is optional for all *Instances* of the given type. The main difference to **Concept 1** is that in this case a cardinality axiom is used. Notice, that a *ModellingRule-Optional* can easily be overridden with a *ModellingRule-Mandatory* also in OWL through subtyping (see also OPC UA Part 3 6.4.4.3 - Subtyping Rules for Properties of ModellingRules). In addition, the "min 0" restrictions of Table 5.4 (row three and four) add no additional meaning to the ontology from an OWL point of view and are only introduced to reflect additional information of OPC UA, this is true for all similar (min 0) restrictions of this thesis.

Concept 3 also defines that the *InstanceDeclaration* is mandatory for all *Instances* of the given type. In contrast to **Concept 1**, **Concept 3** does not restrict the *BrowseName-Attribute* of the corresponding *InstanceDeclaration*. This is reflected in Table 5.4 through the introduction of only one axiom. It is very important to notice the fact, that the semantics of *Placeholder-ModellingRules* depends on the *NodeClass* of the *InstanceDeclaration*. The axioms of **Concept 3** can only be applied for *Variables-* and *Object-InstanceDeclarations*.

Concept 4 is identical to **Concept 3** and differs only on the fact that the *InstanceDeclaration* is optional for all *Instances* of the given type.

Concept 5 is identical to **Concept 1** from an axiom perspective. For *Methods* the *Placeholder-ModellingRules* defines that the *InputArguments* and *OutputArguments* are defined on an *Instance* or subtype. *InputArguments*, as well as *OutputArguments*, are defined through *Properties* in OPC UA and, therefore, are reflected as constraints on the *Method-InstanceDeclaration*.

Concept 6 is identical to **Concept 2** from an axiom perspective and is introduced for similar reasons than **Concept 5** only with optional semantics. Besides *InstanceDeclarations*, *Types* can also *Reference* other *Nodes* without *ModellingRules*. These *Nodes* are often used to expose additional metadata for the *Type-Node* and, therefore, can be also covered in form of annotation properties. Furthermore, if non-hierarchical *References* are used in combination with *ModellingRules* the behavior is undefined according to OPC UA (see also OPC UA Part 3 6.4.4 - ModellingRules). Because of that, such relations could be modeled through annotation properties or similar to optional *InstanceDeclarations* regardless of the *ModellingRule*.

Table 5.4 – *ObjectType* axioms - M = Mandatory; O = Optional.

	Concept Number					
Extract of the subclass axiom	**1**	**2**	**3**	**4**	**5**	**6**
<BrowseNameObjectProperty> some <InstanceDeclClass>	M				M	
<ReferenceTypeObjectProperty> some <InstanceDeclClass>	M		M		M	
<BrowseNameObjectProperty> min 0 <InstanceDeclClass>		M				M
<ReferenceTypeObjectProperty> min 0 <InstanceDeclClass>		M		M		M
opcua:topOpcObjectProperty max 1 <InstanceDeclClass>	O	O			O	O

5.6 VariableType mapping

Figure 5.7 shows an example *VariableType* after the mapping to OWL is applied. Identical to *ObjectTypes* also *VariableTypes* generate an OWL class and the corresponding class hierarchy (see left side of Figure 5.7). Equal to *ObjectTypes* also *VariableTypes* use *BrowseName*-based URIs (starting with the upper case according to OWL coding guidelines). The OWL class hierarchy can be automatically generated based on *HasSubtype References* of OPC UA. The *Attributes* of *VariableTypes* are modeled with OWL annotation properties (see right side of Figure 5.7). Also in this case the reason for the usage of OWL annotation properties is grounded in the behavior during instantiation of *Instances* (see also Section 5.5). In contrast to *ObjectTypes*, some *Attribute* restrictions are modeled through the subtyping of other OWL classes. The subtype and instantiation restriction for the *ValueRank-Attribute* is reflected through subtyping the ValueRankHelper concept (see Section 5.11). The restrictions of the *Value-Attribute* introduced through the *DataType-Attribute* are reflected through subtyping the *DataType* concept of Section 5.3. Furthermore, the restrictions of the *ArrayDimensions-Attribute* of OPC UA (see also OPC UA Part 3 6.2.7 - Attribute Handling

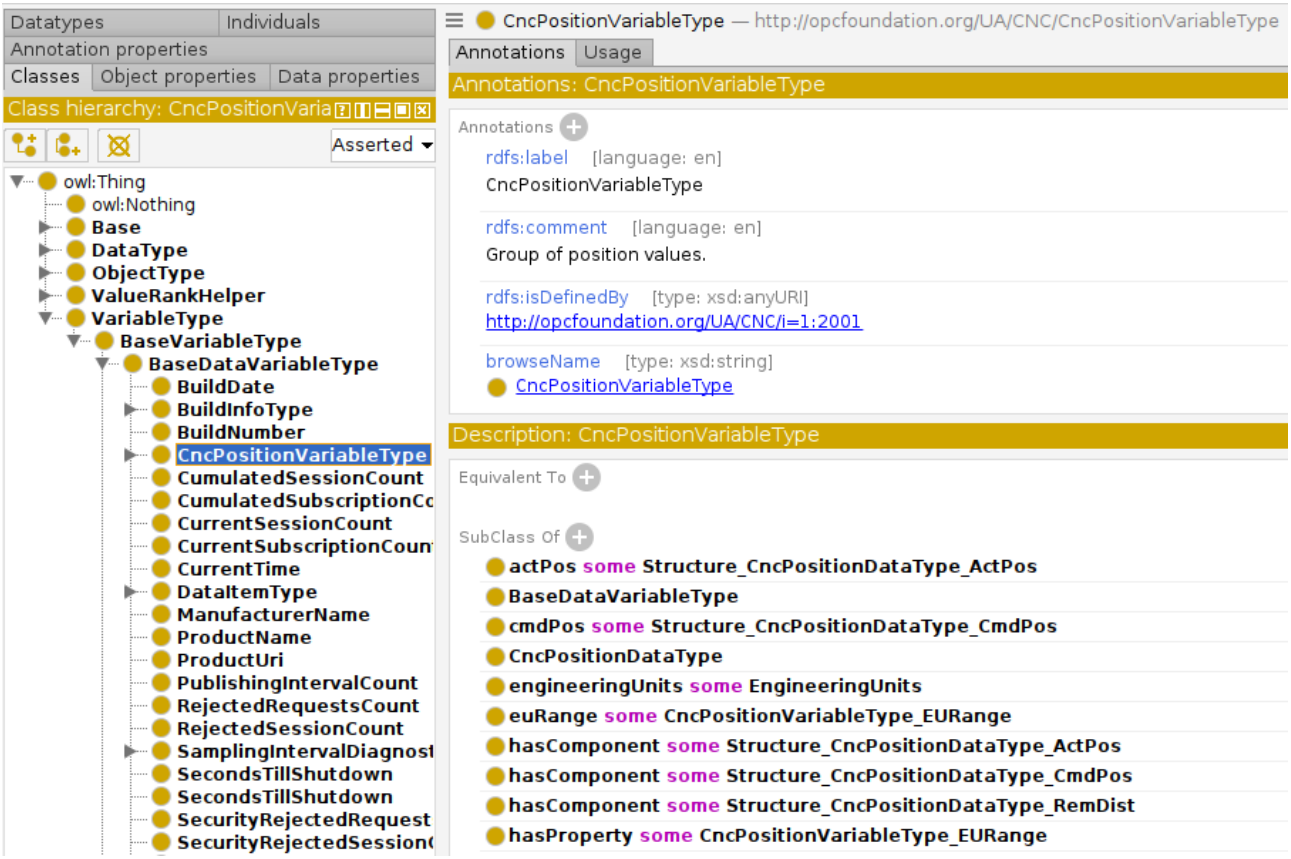


Figure 5.7 – Class concept for *VariableTypes*.

of Variable and VariableTypes) could be modeled through RegEx-pattern restrictions. However, RegEx-based rule enforcement is very expensive and might not scale very well for large graphs in practice. Of course, also the *VariableType* OWL class concept can be used to define optional *Attributes* as mandatory on subtypes or *Instances* (not shown in Figure 5.7). The lower right side of Figure 5.7 outlines some of the OWL subclass axioms. The main purpose of these axioms is to model *InstanceDeclaration* restrictions based on OPC UA *ModellingRules*. OPC UA defines several different combinations of *ModellingRules* in combination with the *VariableType*. However, new *ModellingRules* could be added through extensions of the specification. In the following, the *ModellingRules* of the core specification are further investigated and classified into concepts (see also Table 5.5):

- **Concept 1**: *ModellingRule-Mandatory* for *Variables*
- **Concept 2**: *ModellingRule-Optional* for *Variables*
- **Concept 3**: *ModellingRule-MandatoryPlaceholder* for *Variables*
- **Concept 4**: *ModellingRule-OptionalPlaceholder* for *Variables*
- **Concept 5**: *ModellingRule-ExposesItsArray* for *Variables*

Concept 1 defines that the *InstanceDeclaration* is mandatory for all *Instances* of the given type. This is reflected through several axioms in Table 5.5. The first three axioms are already explained for the *ObjectType* concept in Section 5.5 and can be skipped. The last two axioms are used to add instantiation and subtype restrictions for the *ValueRank*- and *Value-Attribute*.

Table 5.5 – *VariableType* axioms - M = Mandatory; O = Optional.

	Concept Number				
Extract of the subclass axiom	**1**	**2**	**3**	**4**	**5**
<BrowseNameObjectProperty> some <InstanceDeclarationClass>	M				
<ReferenceType> some <InstanceDeclarationClass>	M		M		
<BrowseNameObjectProperty> min 0 <InstanceDeclarationClass>		M			
<ReferenceType> min 0 <InstanceDeclarationClass>		M		M	M
opcua:topOpcObjectProperty max 1 <InstanceDeclarationClass>	O	O			
<DataTypeClass>	M	M	M	M	M
<ValueRankClass>	M	M	M	M	M

Concept 2 defines that the *InstanceDeclaration* is optional for all *Instances* of the given type. The main difference to **Concept 1** is that a cardinality axiom is used.

Concept 3 defines that the *InstanceDeclaration* is mandatory for all *Instances* of the given type. In contrast to **Concept 1**, **Concept 3** does not restrict the *BrowseName-Attribute* of the corresponding *InstanceDeclaration*. Table 5.5 reflects that fact through the introduction of only one axiom.

Concept 4 is identical to **Concept 3** and differs only in the fact that the *InstanceDeclaration* is optional for all *Instances* of the given type.

Concept 5 introduces the *ModellingRule-ExposesItsArray* for usage in combination with *Variables* only. The idea behind this *ModellingRule* is to expose the entries of an array also in form of single *Nodes*. Of course, this concept could also be generalized in OWL for each array to ensure easy access to the different array elements. Nevertheless, while this concept is very useful to access different array elements in an easy way, the order of the elements (e.g., the first or third entry) cannot be retrieved with this concept. In the end, **Concept 5** is identical to **Concept 4** from an axiom point of view. Furthermore, *Nodes* without *ModellingRules* or *Nodes* which are connected through non-hierarchical *References* can be addressed in the same way as already explained in Section 5.5.

5.7 Object InstanceDeclaration mapping

An example of an *Object-InstanceDeclaration* after applying the OWL mapping is depicted in Figure 5.8. *InstanceDeclarations* are mapped to OWL classes because of the semantics attached to this concept. In contrast to *Types*, *InstanceDeclarations* make use of *NodeId*-based URIs. This is mainly due to the fact, that a typical OPC UA information model generates far too many naming conflicts if the *BrowseName* would be used as an URI. For example, every time an *InstanceDeclaration* is overridden or a *Type* is reused to construct another *Type* also the *InstanceDeclarations* are duplicated in most of the cases. While OPC UA offers the possibility to reuse *InstanceDeclarations* if the semantics and default-values should not be changed, most modeling tools automatically generate new *InstanceDeclarations* during modeling instead of following a more restrictive policy like "copy on write". From a semantic point of view, this is a very bad modeling practice because now several semantic identical classes are introduced, which only differ in the URI. Of course, it would be possible to analyze the graph and remove duplicates but this introduces other problems. For example, if an OPC UA engineer searches for a particular *NodeId* of an *InstanceDeclaration*, which is marked as duplicate and removed from the graph, the search would return empty. Of course, it would be possible to assign more than one *NodeId* to a given *InstanceDeclaration* (see also Section 5.2). However, the fact that *InstanceDeclarations* can also be overridden introduces further checks

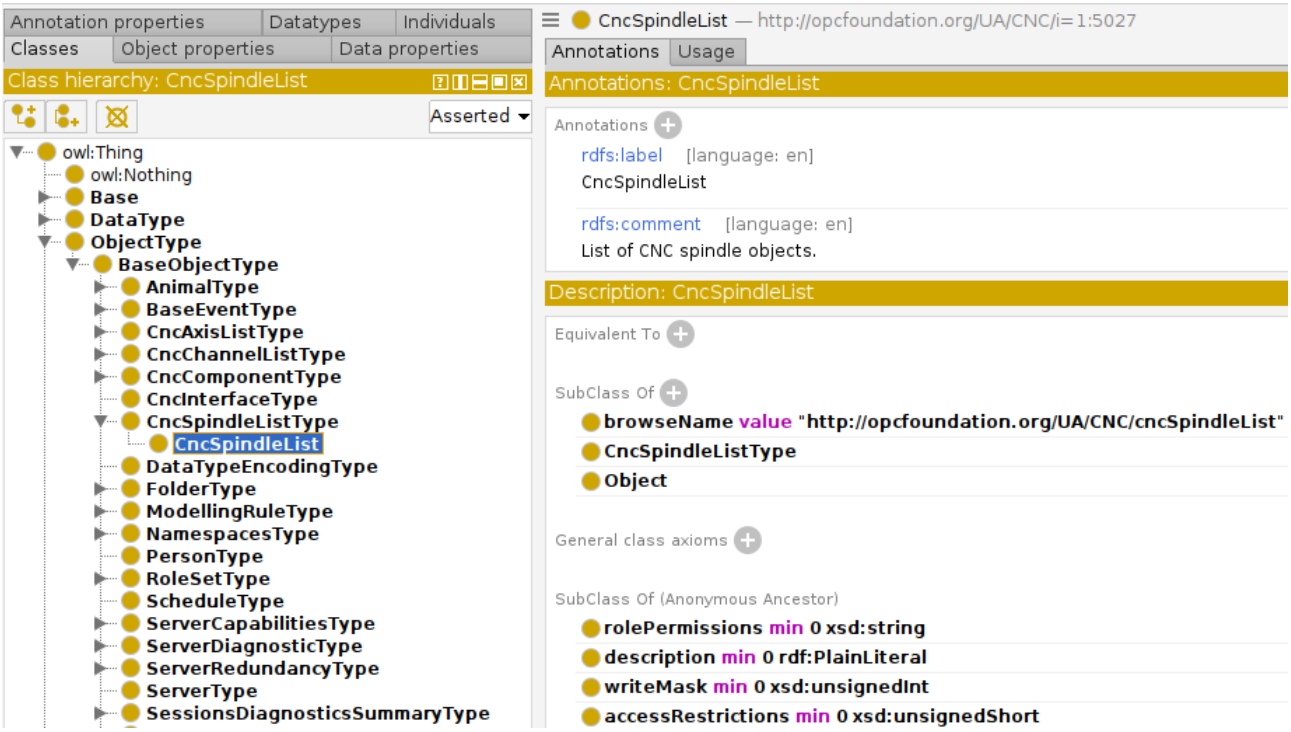


Figure 5.8 – Class concept for *Object-InstanceDeclaration*.

on the graph transformation tool. Another more simple variant would be to introduce an additional *InstanceDeclaration* super concept, which is shared by all similar *InstanceDeclarations*. In the end, the best solution is if such kinds of semantic classifications are done through the OPC UA modeling experts in the first place rather than with complex analytic and transformation tools in retrospect. Most *Attributes* of *InstanceDeclarations* are modeled with OWL annotation properties (see right side of Figure 5.8). Also in this case the reason for the usage of OWL annotation properties is grounded in the behavior during instantiation of *Instances* (see also Section 5.5). In contrast to *Types*, mandatory *InstanceDeclarations* also define restrictions on the *BrowseName-Attribute* (see also Figure 5.8 right side). Figure 5.8 outlines some of the OWL subclass axioms in the lower right corner. The main purpose of these axioms is to model additional *InstanceDeclaration* restrictions based on OPC UA *ModellingRules*, which are not already covered by *Types*. OPC UA defines several different combinations of *ModellingRules* in combination with the *Object-InstanceDeclaration*. However, new *ModellingRules* could be added through extensions of the specification. In the following, the *ModellingRules* of the core specification are further investigated and classified into concepts (see also Table 5.6):

- **Concept 1**: *ModellingRule-Mandatory*

- **Concept 2**: *ModellingRule-Optional*

- **Concept 3**: *ModellingRule-MandatoryPlaceholder*

- **Concept 4**: *ModellingRule-OptionalPlaceholder*

Concept 1 as well as **Concept 2** are identical from an axiom point of view. Table 5.6 shows the three axioms for the concepts. In both cases, an additional OWL object property is generated based on the *BrowseName* (see also Section 5.4 for further information). This OWL object property shall be used in addition to the *ReferenceType* OWL object property to *Reference* such *InstanceDeclarations* (see also Section 5.5) or *Instances* of such *InstanceDeclarations* (see also Section 5.10). The main reason for the introduction of this OWL object property is to support the OPC UA concept "programming against the *TypeDefinitionNode*", which is especially useful for SPARQL-based queries. Besides the *BrowseName* OWL object property concept also a *BrowseName* OWL data property restriction is created (see also Table 5.6 and Figure 5.8). This axiom ensures that each *Instance* has the same value for the *BrowseName-Attribute*. The second axiom is based on the *HasTypeDefinition-ReferenceType* and ensures restriction inheritance of the *Type* because each *Object-InstanceDeclaration* always is an *Instance* of an *ObjectType*. The third axiom of Table 5.6 imports the restrictions of the object meta-layer class (see also Section 5.1).

Concept 3 and **Concept 4** differ from the other concepts only in the fact that neither an OWL object property nor an OWL data property restriction is introduced for the *BrowseName*. Of course, it is possible that an *Object-InstanceDeclaration* itself *References* further *InstanceDeclarations*. In this case, the generation rules for axioms of Section 5.5 shall also be used on *Object-InstanceDeclarations*. If an *InstanceDeclaration* shall be overridden the overriding *InstanceDeclaration* must also be a subtype of the overridden *InstanceDeclaration*. This ensures a proper inference of the restriction axioms on super classes. Notice, that for only changing the *ModellingRule-Optional* to *ModellingRule-Mandatory* it is not necessary to introduce a new *InstanceDeclaration* class.

Table 5.6 – *Object-InstanceDeclaration* axioms - M = Mandatory; O = Optional.

Extract of the subclass axiom	**Concept Number**			
	1	**2**	**3**	**4**
ia:browseName value <"browseName">	M	M		
<TypeDefinitionClass>	M	M	M	M
Object	M	M	M	M

5.8 Variable InstanceDeclaration mapping

OPC UA defines two main concepts around *Variables*. The first concept is called *DataVariables* and the second concept is called *Properties*. Both concepts differ in the way how semantics is exposed and also offer different substructure and subtyping capabilities. While the semantics of *Properties* is mainly defined through their *BrowseName*, the semantics of *DataVariables* are mainly defined through the corresponding *VariableType*. Furthermore, the corresponding *Types* of *DataVariables* can be subtyped and also used to expose complex *Variables*. In contrast, *Properties* can neither be subtyped nor be used to expose complex *Variables*. Based on these large differences, two different concepts for *DataVariables* (Section 5.8.1) and *Properties* (Section 5.8.2) are presented.

5.8.1 DataVariables

Figure 5.9 shows an example *Variable-InstanceDeclaration* after the mapping to OWL is applied. Similar to *Object-InstanceDeclarations* also *Variable-InstanceDeclarations* are mapped to OWL classes. *Variable-InstanceDeclarations* make use of *NodeId*-based URIs (see also Section 5.7 for further arguments). Most *Attributes* of *InstanceDeclarations* are modeled with OWL annotation properties (see right side of Figure 5.9). Also in this case the reason for the usage of OWL annotation properties is grounded in the behavior during instantiation of *Instances* (see also Section 5.5). Similar to *Object-InstanceDeclarations* also *Variable-InstanceDeclarations* define restrictions on the *BrowseName-Attribute* (see also Figure 5.9 right side). Figure 5.9 outlines some of the OWL subclass axioms in the lower right corner. The main purpose of these axioms is to model additional *InstanceDeclaration* restrictions based on OPC UA *ModellingRules*, which are not already covered by *Types*. OPC UA defines several different combinations of *ModellingRules* in combination with the *Variable-InstanceDeclaration*. However, new *ModellingRules* could be added through extensions of the specification. In the following, the *ModellingRules* of the core specification are further investigated and classified into concepts (see also Table 5.7):

Table 5.7 – *Variable-InstanceDeclaration* axioms - M = Mandatory; O = Optional.

	Concept Number					
Extract of the subclass axiom	**1**	**2**	**3**	**4**	**5**	**6**
ia:browseName value <"browseName">	M	M				M
<TypeDefinitionClass>	M	M	M	M	M	M
Variable	M	M	M	M	M	M
<DataTypeClass>	M	M	M	M	M	M
<ValueRankClass>	M	M	M	M	M	M
<PropertyClass>						O

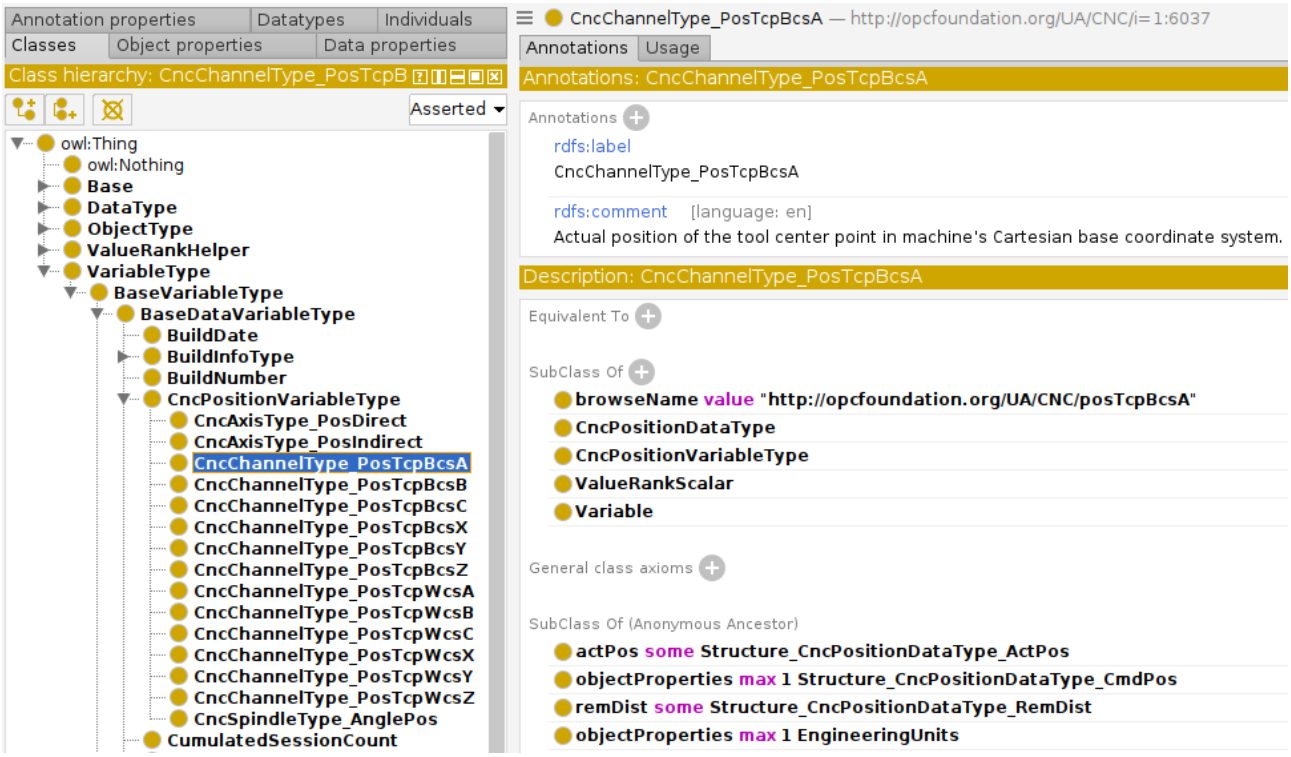


Figure 5.9 – Class concept for *Variable-InstanceDeclaration*.

- **Concept 1**: *ModellingRule-Mandatory* without Properties
- **Concept 2**: *ModellingRule-Optional* without Properties
- **Concept 3**: *ModellingRule-MandatoryPlaceholder*
- **Concept 4**: *ModellingRule-OptionalPlaceholder*
- **Concept 5**: *ModellingRule-ExposesItsArray*
- **Concept 6**: *ModellingRule-Mandatory and -Optional* in combination with Properties

Concept 1 and **Concept 2** are identical in their axioms. Table 5.7 shows the five axioms for the concepts. In both cases, an additional OWL object property is generated in combination with an OWL data property restriction based on the *BrowseName*. The generation rules

are identical to *Object-InstanceDeclarations* and are discussed in Section 5.7. Similar to *Object-InstanceDeclarations* also *Variable-InstanceDeclarations* inherit restrictions from the *Type*. In contrast to *Object-InstanceDeclarations*, the variable meta-layer class is used for restriction import instead of the object meta-layer class (see also Section 5.1). Of course, also *Variable-InstanceDeclarations* inherit the restrictions for the *DataType*- and *ValueRank-Attribute* (see Section 5.6 for further information). This is necessary because an *InstanceDeclaration* could further tighten the corresponding restrictions by, for example, defining a more concrete *DataType*, which is a subtype of the *DataType* specified in the *VariableType*.

Concepts 3, 4, and 5 also share identical axioms. The only difference to **Concept 1** is that neither an OWL object property nor an OWL data property restriction is introduced for the *BrowseName*.

Concept 6 is introduced because of the special semantics around the OPC UA *PropertyType*. In contrast to *DataVariables*, where the semantics is defined through the *TypeDefinitionNode*, the semantics of *Properties* in OPC UA is defined through the *BrowseName* (see also OPC UA Part 3 A.4.2 - Properties or DataVariables). This means if a *Property* is defined multiple times with the same *BrowseName* also the semantics should be identical. This special construct is covered through the introduction of an additional class concept, which is further discussed in Section 5.8.2. The only difference between the axioms of **Concept 1** and the axioms of **Concept 6** is the inclusion of this additional *Property* concept class. Furthermore, this axiom is also optional because a reasoner is able to infer this relationship automatically. However, for use cases without reasoners, the translation tool could easily insert such an axiom directly into the ontology. Of course, it is possible that an *Variable-InstanceDeclaration* itself *References* further *InstanceDeclarations*. In this case, the generation rules for axioms of Section 5.6 shall also be used on *Variable-InstanceDeclarations*. As already discussed in Section 5.7, also for *Variable-InstanceDeclarations* the same concepts for overriding apply.

5.8.2 Properties

As already discussed in Section 5.8.1 the semantics of OPC UA *Properties* are defined through the *BrowseName*. If different *Properties* share the same *BrowseName* also the semantics should be the same. Figure 5.10 shows how an additional OWL class concept for *Properties* is introduced. Such a class is introduced for each unique *BrowseName* of a OPC UA *Property*. The general class axiom of this concept (see Figure 5.10 lower right corner) ensures that each OPC UA *Property* is automatically assigned to this concept based on the *BrowseName-Attribute*. As URI the *BrowseName* is used to simplify the tool-based ontology handling. Due to the fact, that OWL object property *BrowseNames* (see Section 5.4) URIs start with lower case and the OWL class *Property* concept URI starts with upper case no naming conflicts are introduced. Notice, that this concept is only

introduced to classify semantics and has no further additional restrictions attached to it. To reduce the computing effort for any given ontology this axiom can also be directly introduced through the transformation tool without the usage of a general class axiom.

5.9 Method InstanceDeclaration mapping

The *Method NodeClass* in OPC UA is one of the *NodeClasses* without a corresponding *Type*. This means that the semantics of *Methods* are mainly defined by the owning *Object* or *ObjectType*. However, OPC UA defines some exceptions in OPC UA Part 3 5.5.4 - Client-side creation of *Objects* of an *ObjectType*. If a *Method* has the special *BrowseName* "Create" than the *Method* semantics is that an *Object* of the given *ObjectType* shall be created on the execution of the *Method*. At the moment this rule is unique in the OPC UA universe and could be covered similar to the OPC UA *Property* concept of Section 5.8.2. An example of an *Method-InstanceDeclaration* after applying the OWL mapping is depicted in Figure 5.11. Also *Method-InstanceDeclarations* are mapped to OWL classes and use *NodeId*-based URIs (see also Section 5.7 for further arguments). Similar to *Object-InstanceDeclarations* also *Attributes* of *Method-InstanceDeclarations* are modeled with OWL annotation properties (see right side of Figure 5.11). Also in this case, the reason for the usage of OWL annotation properties is grounded in the behavior during instantiation of *Instances* (see also Section 5.5). Identical to *Object-InstanceDeclarations* also *Method-InstanceDeclarations* define restrictions on the *BrowseName-Attribute* (see also Figure 5.11 right side). Some of the OWL subclass axioms are depicted in Figure 5.11 in the lower right corner. The main purpose of these axioms is to model additional *InstanceDeclaration* restrictions based on OPC UA *ModellingRules*, which are not already covered by *Types*. OPC UA also defines several different *ModellingRules* and restrictions for *Method-InstanceDeclarations*. However, also in this case new *ModellingRules* could be added through extensions of the specification. In the following, the *ModellingRules* of the core specification are further investigated and classified into concepts (see also Table 5.8):

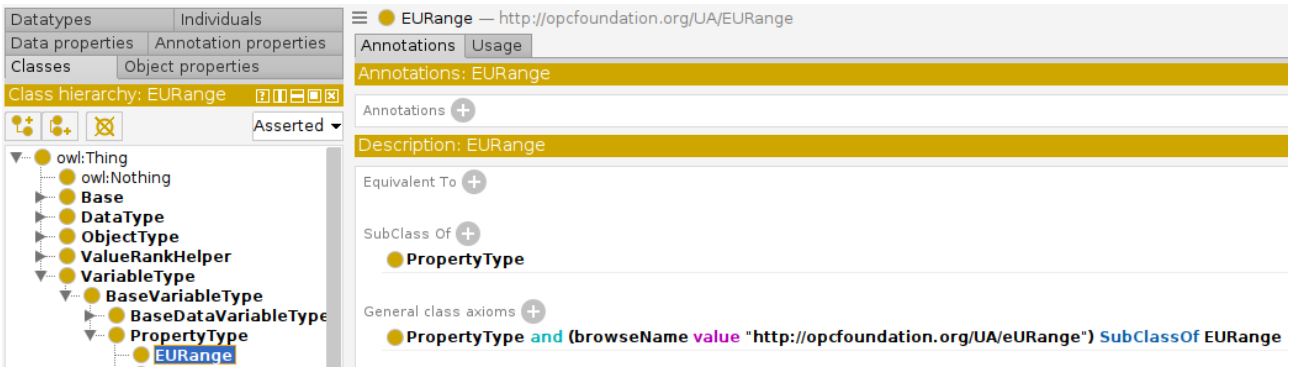


Figure 5.10 – Class concept for *Properties*.

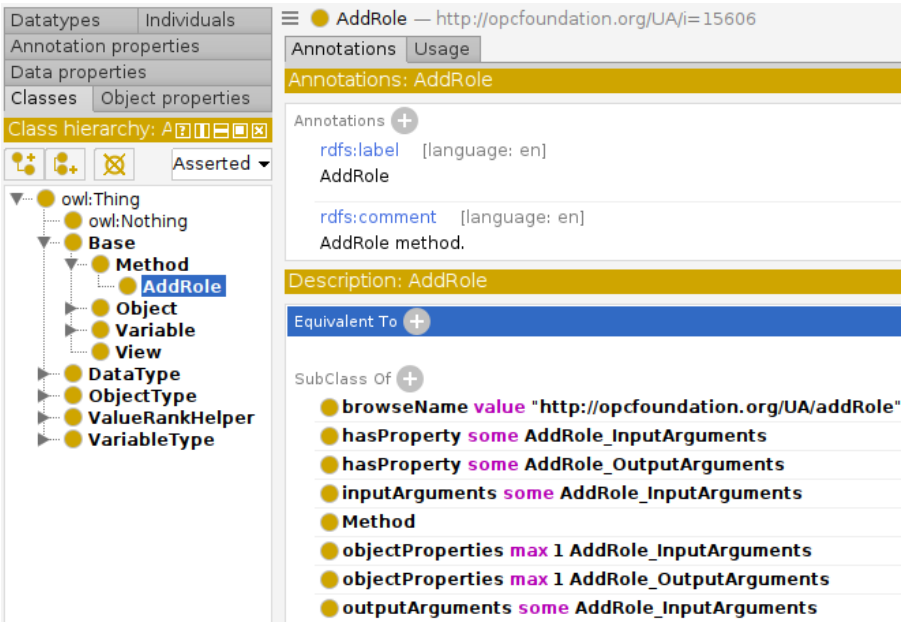


Figure 5.11 – Class concept for *Method-InstanceDeclarations*.

- **Concept 1**: *ModellingRule-Mandatory*
- **Concept 2**: *ModellingRule-Optional*
- **Concept 3**: *ModellingRule-MandatoryPlaceholder*
- **Concept 4**: *ModellingRule-OptionalPlaceholder*

In contrast to *Object-* and *Variable-InstanceDeclarations* the axioms for *Method-InstanceDeclarations* are identical for **Concept 1, 2, 3,** and **4**. This is mainly due to the fact that OPC UA defines *ModellingRule-Placeholder** different on *Method-InstanceDeclarations* compared to the other *InstanceDeclarations* concepts (see also Section 5.5). In each case, an additional OWL object property is generated in combination with an OWL data property restriction based on the *BrowseName*. The generation rules are identical to *Object-InstanceDeclarations* and are discussed in Section 5.7. In contrast to *Object-InstanceDeclarations*, *Method-InstanceDeclarations* do not have a *Type* (referenced through a *HasTypeDefinition-Reference*) but also inherit restrictions from the corresponding method meta-layer class (see also Section 5.1). Another difference to the other *InstanceDeclarations* is the possibility to change a *ModellingRule-OptionalPlaceholder* to a *ModellingRule-MandatoryPlaceholder* or a *ModellingRule-Mandatory* and a *ModellingRule-MandatoryPlaceholder*

to a *ModellingRule-Mandatory*. Also in the proposed OWL transformation this is possible because the *BrowseName* concepts are present in all four concepts. Notice, that in the case of *Object*- and *Variable-InstanceDeclarations* it would make not much sense to allow such behavior because the *BrowseNames* of the super class typically are defined in some placeholder syntax. Of course, it is possible that an *Method-InstanceDeclaration* itself *References* further *InstanceDeclarations* (e.g., "InputArgument" and "OutputArgument" *InstanceDeclarations*). In this case, the generation rules for axioms of Section 5.5 shall also be used on *Method-InstanceDeclarations*. As already discussed in Section 5.7 also for *Method-InstanceDeclarations* the same concepts for overriding apply.

Table 5.8 – *Method-InstanceDeclaration* axioms - M = Mandatory; O = Optional.

	Concept Number			
Extract of the subclass axiom	**1**	**2**	**3**	**4**
ia:browseName value <"browseName">	M	M	M	M
Method	M	M	M	M

5.10 Instance mapping

Figure 5.12 shows an example *Variable-Instance* after the mapping to OWL is applied. For every *Instance* an OWL individual is generated. The URI for the individuals is based on the *NodeId* schema. As depicted on the right side of Figure 5.12 an OWL individual of the OPC UA mapping consists of OWL annotations, class assertions, object property assertions, and data property assertions. OWL annotations are used to express certain meta information of the OPC UA *Node* in a standardized manner. For example, the label and comment annotation are well-known in the OWL ecosystem an can be mapped to the *DisplayName*- and *Description-Attribute*. Of course, both of these *Attributes* are also reflected as data property assertions (see also Figure 5.12 lower right side). However, also several other annotations are generated through the transformation to simplify the usage of the resulting ontology and also to cover special OPC UA concepts. The **nodeExists** annotation assertion of Figure 5.12 is introduced because in OPC UA it is possible to model a relation between two *Nodes* even if one of the *Nodes* is not defined at all. In OWL this can be compared with the fact that the subject or object of a triple has a URI but this URI cannot be resolved. The problem in OPC UA is that even due to the fact that the *Node* may not exist, it is possible to store information like *BrowseName*, *DisplayName*, *NodeClass*, and *TypeDefinition* on the edge. However, OPC UA also states that if the *Node* is not part of the actual server some of the information may be not provided or out of date. Nevertheless, in the proposed OPC UA to OWL mapping information like *DisplayName* and *NodeClass* are modeled on the OWL class and not on the OWL object property. To be able to cover this concept the solution is to introduce shadow *Nodes*. These *Nodes* are modeled

with all the available information and are annotated with **nodeExists** OWL annotation properties with the value set to "false". This allows to distinguish between generated *Nodes* and explicitly modeled *Nodes*. *References* to other *Nodes* are expressed through OWL object property assertions (see also Figure 5.12 right side). If the *Reference* is part of an *InstanceDeclaration* concept, which also defines a *BrowseName* OWL object property, an additional OWL object property assertion is inserted into the ontology (not shown in Figure 5.12). Notice, that there is at most one target OWL individual for such an assertion. The OWL object property assertions highlighted in yellow are automatically generated by the reasoner based on the OWL object property hierarchy of Section 5.4 and the corresponding axioms. In the middle of Figure 5.12 OWL class assertions are expressed. Depending on the underlying modeling concept (e.g., *Object*, *Variable*, *Method*, and *View*) different class assertions are present on an OWL individual. In the following, the differences between the modeling concepts are discussed in greater detail.

Table 5.9 states the axioms for the different modeling concepts. An example for the **Variable** concept of an *InstanceDeclaration* is depicted in Figure 5.12. In this case, several OWL class assertions are generated. The first OWL class assertion imports semantics and restrictions of the corresponding *InstanceDeclaration* class. Notice, that only if the *Node* is based on an *InstanceDeclaration* this OWL class assertion is introduced. In all other cases, this particular OWL class assertion shall be omitted. The second OWL class assertion imports the *TypeDefinitionNode*. Every *Variable*

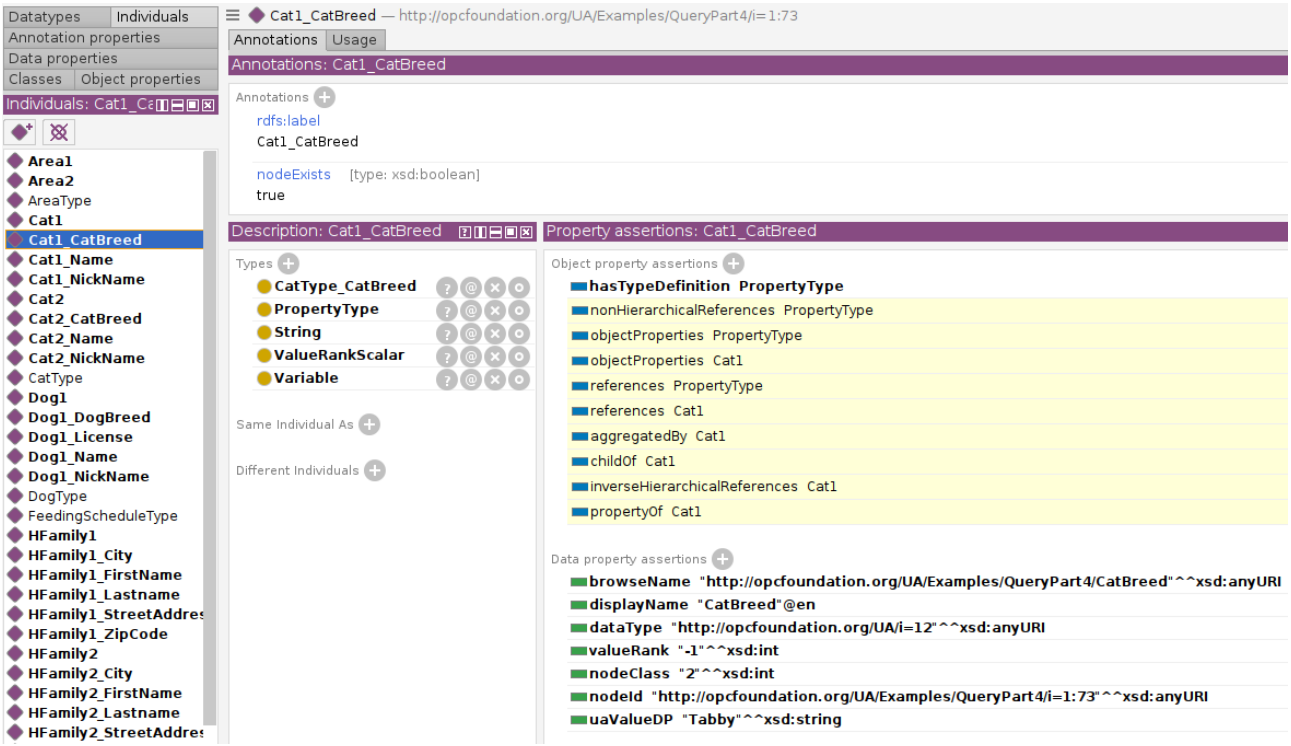


Figure 5.12 – Individual concept for *Instances* (not complete).

Table 5.9 – *Instance* axioms - M = Mandatory; O = Optional.

Extract of the axiom	Obj	Var	Met	Views
ClassAssertion(<TypeDefinitionClass>)	M	M		
ObjectPropertyAssertion(<ReferenceTypeObjectProperty> <TargetNodeIndividual>)	M	M	M	M
ObjectPropertyAssertion(<BrowseNameObjectProperty> <TargetNodeIndividual>)	M	M	M	
DataPropertyAssertion(<AttributeDataProperty> "<AttributeValue>")	M	M	M	M
AnnotationPropertyAssertion(<AttributeAnnotationProperty> "<AttributeValue>")	O	O	O	O
ClassAssertion(Object)	M			
ClassAssertion(Variable)		M		
ClassAssertion(Method)			M	
ClassAssertion(View)				M
ClassAssertion(<InstanceDeclarationClass>)	M	M	M	
ClassAssertion(<DataTypeClass>)		M		
ClassAssertion(<ValueRankClass>)		M		

has exactly one *TypeDefinitionNode* and, therefore, this assertion is always present. Of course, as mentioned in Section 5.7 the *InstanceDeclaration* class also is a subtype of the OWL *TypeDefinitionNode*, resulting in the fact that the *TypeDefinitionNode* is already imported through the *InstanceDeclaration* concept. However, also an *Instance* is allowed to further refine the *InstanceDeclaration* (e.g., through the usage of a subtype). The third and fourth OWL class assertions import the restrictions of the *DataType-* and *ValueRank-Attribute*. Also in this case the *Instance* is allowed to further refine the restrictions of the *TypeDefinitionNode* and if applicable *InstanceDeclaration* concept (see also OPC UA Part 3 6.4.3 - Constraints on an Instance). Last but not least, the variable meta-class concept is imported through a class assertion. The main difference between **Objects** and **Variable** axioms of Table 5.9 is that the meta-class object is used instead of the meta class variable. Furthermore, an **Object** does not add any *DataType* or *ValueRank* axioms. For **Methods** the axioms compared to **Objects** are reduced even further because a **Method** does not have a *TypeDefinitionNode*. **Views** only introduce OWL data and object property assertions and, of course, an OWL class assertion for the view meta-layer class. Besides the previously discussed generation rules, Figure 5.12 also shows a *HasTypeDefinition* OWL object property with the *PropertType* OWL individual as the target. However, in Section 5.8.2 the *PropertyType* is introduced as OWL class and not as OWL individual. The idea behind this is grounded in OWL punning [115, 56], which allows to model OWL classes also as OWL individuals to support meta modeling. In the case of the OPC UA to OWL mapping punning is used to simplify queries and also can be used to enable a better validation of the OPC UA type model. Finally, OWL punning has to be used if an

InstanceDeclaration (which is modeled as OWL class) is also referenced by other *Instances*, which are mapped to OWL individuals. Notice, that OWL annotation properties are shared automatically if punning is used. Other concepts like OWL data and object property assertions have still to be added to the OWL individual.

5.11 ValueRankHelper

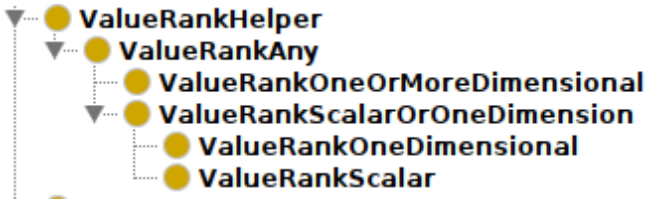


Figure 5.13 – Class hierarchy overview of ValueRankHelper.

The concept of **ValueRankHelper** is introduced for OWL-based validation purposes. Figure 5.13 depicts the OWL class hierarchy of the different restrictions and Table 5.10 lists the essential restriction axioms for the different classes. The OWL subtype hierarchy in combination with the axioms of Table 5.10 covers the rules for *Variables* used as *InstanceDeclarations* or *VariableTypes* if they shall be overridden or instantiated according to OPC UA Part 3 6.2.7 - Attribute Handling of Variable and VariableTypes. The constraint from OPC UA Part 3 6.2.7 c) "The *ArrayDimensions Attribute* may be added if it was not provided or when modifying the value of an entry in the array from 0 to a different value. All other values in the array shall remain the same." can be mapped to a RegEx expression (see also last entry of Table 5.10). In the end, RegEx-based validation can be considered very expensive and the usage should be limited to small parts of the ontology or even replaced with a more efficient way of validation.

Table 5.10 – ValueRankHelper OWL class axioms.

Concept name	**Extract of the axiom**
ValueRankHelper	SubClassOf(valueRank some xsd:integer)
ValueRankAny	SubClassOf(valueRank some xsd:integer[>=-3])
ValueRankOneOrMoreDimensions	SubClassOf(valueRank some xsd:integer[>=0])
ValueRankScalarOrOneDimension	SubClassOf((valueRank value -1) or (valueRank value 1))
ValueRankOneDimension	SubClassOf(valueRank value 1)
ValueRankScalar	SubClassOf(valueRank value -1)
Example for OPC UA Part 3 6.2.7 c)	SubClassOf(arrayDimensions some xsd:string[pattern "\\[1,[0-9]0,3,1\\]"])

5.12 Demonstrator

In the following, the basic concept of the transformation tool is presented (see also Figure 5.14). Starting at the upper left corner with an OPC UA information model, which is transformed into the machine-readable XML format of OPC UA (lower left corner). This XML format is the input for the OPC UA to OWL transformation tool (implemented with Apache Jena [11]). Within the transformation tool, the recursive OPC UA to OWL transformation is applied and as result, an OWL ontology is generated and serialized (lower right corner of Figure 5.14). After the generation of the OWL ontology, this ontology can be imported into several tools from the Semantic Web ecosystem like Protégé (upper right corner of Figure 5.14) for further processing of the formalized semantics. Within this chapter, an OWL mapping is exemplified with the focus on the *Instance* layer of OPC UA. This mapping is especially useful to work with the product-specific implementation of different *Companion Specifications* and can be used for analytics, query, and validation. The transformation rules presented in this thesis can generate ontologies with OWL 2 DL expressivity and lower, which also allow inferring statements about the expressivity of OPC UA information models. Notice, that OWL 2 DL expressivity is the maximum expressivity if all identified OPC UA concepts shall be translated into the corresponding OWL concepts. The expressivity and the corresponding complexity can be reduced by large amounts for use cases like query and analytics. In contrast, the validation use case depends on nearly all the transformation rules if OWL reasoners shall be used for validation. However, as already explained in Section 5.5 most *Type-Attributes* are translated to OWL annotation properties and so cannot be validated with OWL reasoners. Instead

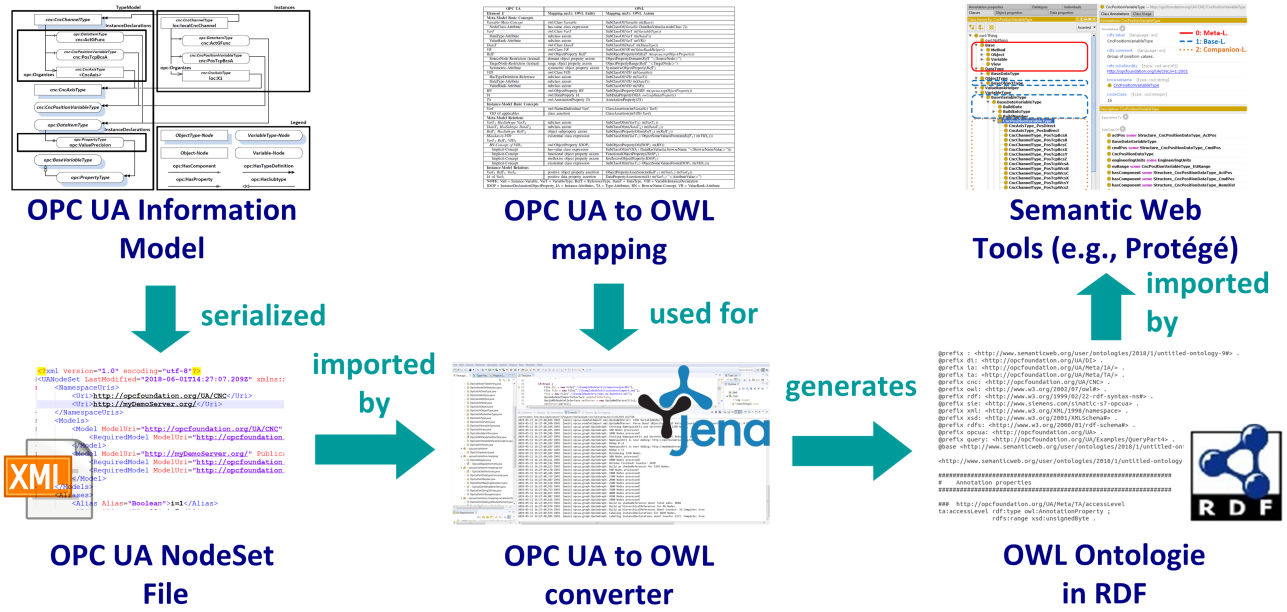


Figure 5.14 – Architecture overview OWL transformation tool-chain.

of OWL reasoners, of course, also SPARQL could be used to validate OWL annotation properties. Nevertheless, it is also possible to slightly alter the ontology generation and optimize it for the target use case (e.g., validation of the *Type* model instead of the *Instance* model). The semantic points identified in Section 3.2 are valid for each transformation and it is also possible to reuse the transformation concepts introduced in this chapter in a slightly different manner (e.g., generation of OWL individuals for *Types* and *InstanceDeclarations*). Furthermore, it is also possible to generate SHACL shapes instead of OWL restrictions or insert further annotation properties for additional use cases. Also in this case the identified concepts in OPC UA for OWL restrictions can be reused.

Figure 5.15 depicts parts of the Java-based implementation of the OWL transformation tool. The project is structured in several packages.

The **opcua.graph** package contains classes and interfaces for the different *NodeClasses*. Furthermore, an OPC UA graph implementation is part of this package. The implementation is able to build graphs from different XML *NodeSets* including reindexing of, for example, *NodeIds* if more than one *NodeSet* should be imported. In addition, the graph also offers further functions like the identification of hierarchical *ReferenceTypes* and *InstanceDeclarations*.

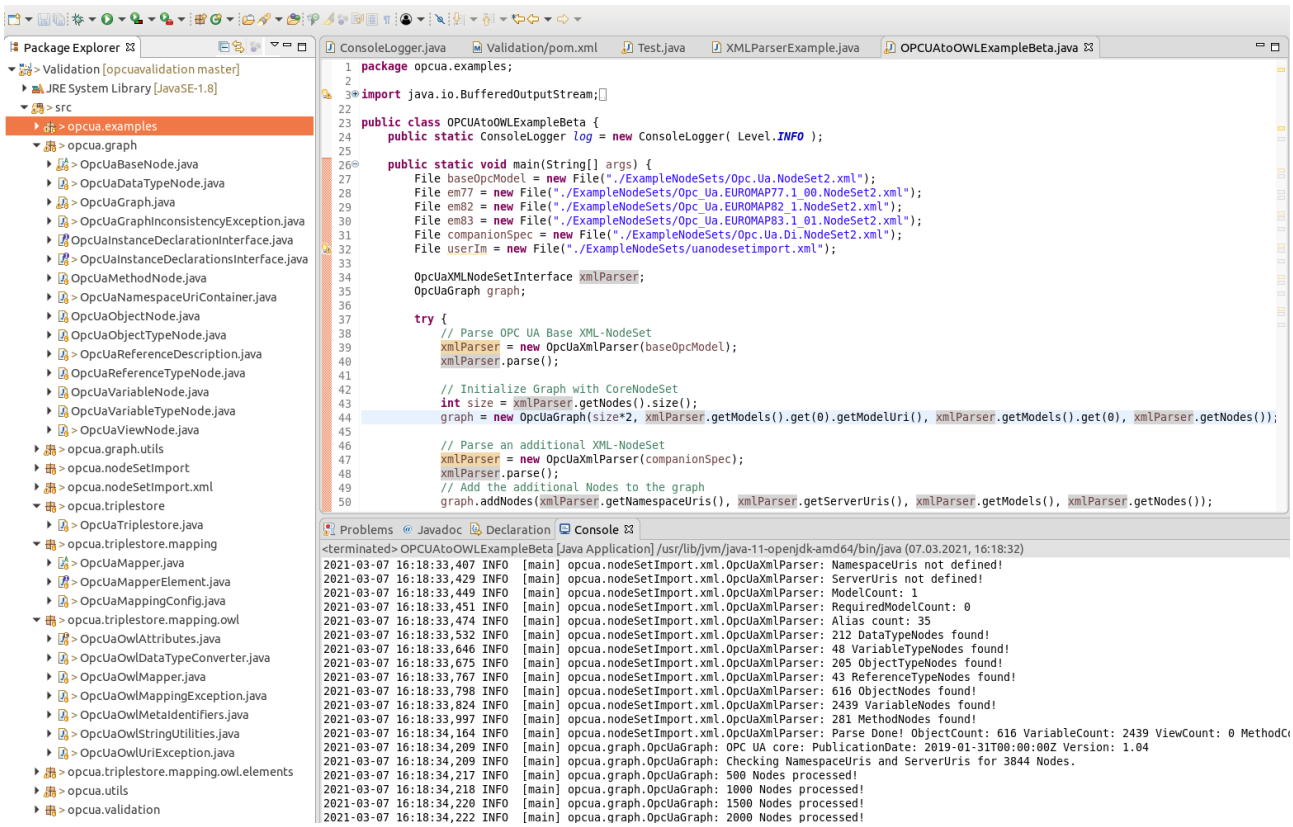


Figure 5.15 – Demonstrator implementation.

The **opcua.graph.utils** package contains classes and interfaces for filtering purposes like *InstanceDeclarations*.

The **opcua.nodeSetImport** and **opcua.nodeSetImport.xml** contains classes and interfaces which are used in the **opcua.graph** package to import *NodeSets*. The current implementation only supports the import of XML-based *NodeSets*. In the future, it is also planned to import *NodeSets* through an OPC UA client from running servers.

The packages **opcua.triplestore**, **opcua.triplestore.mapping**, **opcua.triplestore.mapping.owl**, and **opauc.triplestore.mapping.owl.elements** contain the mapping from OPC UA to OWL. The current architecture translates the graph step by step, starting with the generation of the meta elements, followed by the transformation of *ReferenceTypes*, *DataTypes*, and the other *NodeClasses*. Furthermore, the architecture allows chaining the different translation elements, which can be used to generate different profiles of the OWL ontology (e.g., a smaller ontology without certain *Attributes* for embedded devices).

Finally, the package **opcua.examples** contains examples of the tool usage. The first step is always the import of the OPC UA core *NodeSet* (see also Figure 5.15 right side). After the core *NodeSet* is imported, other *NodeSets* (e.g., *Companion Specifications*) can be also imported. If all *NodeSets* are imported the graph must be prepared for the OWL transformation (e.g., labeling *InstanceDeclarations*). As soon as the graph is properly annotated the OWL transformation chains can be executed. The result of the complete process is an RDF file with an OWL ontology.

5.13 Evaluation

This section evaluates the proposed approach of this thesis against the formulated research challenge (C2) interoperability on the semantic layer. Within Section 3.2 the evaluation metrics for the given research challenge are formulated and existing research approaches are checked against these metrics. Based on this evaluation three unsolved problems are identified in the existing research approaches: First, the concept behind *InstanceDeclarations* is not covered at all in actual research. Second, the modeling of OPC UA constraints is only covered in parts by some researchers. Third, a complete translation of all OPC UA semantics into the Semantic Web is not presented in one single concept.

Table 5.11 shows the evaluation results of the proposed approach in this thesis. Section A.2.1 provides a concept of how **Namespaces** can be automatically translated into OWL also including *Companion Specifications* based on the recursive mapping of Chapter 5. Furthermore, Section A.2.2 even provides first ideas to improve the versioning of OPC UA information models. **Attributes** (Section 5.2), as well as **DataTypes** (Section 5.3), are mapped well including restrictions like domain and range but the proposed mapping of this thesis also has certain limitations.

While *Structures* and *Enumerations* are mapped very well through Section 5.3.2 and Section 5.3.3 aspects like arrays and the handling of time series data could be improved further through other researchers. **ReferenceTypes** are mapped very well including type hierarchies and characteristics like symmetric, inverseOf, and even domain and range constraints (domain and range constraints are added manually and cannot be extracted automatically). The **ObjectType** (Section 5.5) and **VariableType** (Section 5.6) mapping is also very well and introduces mappings for types (including type hierarchies) as well as constraints. Section 5.7 and Section 5.8 provides the very well mapping for **InstanceDeclarations** with features like types, type hierarchies, constraints, as well as specialized concepts to cover the semantics of *DataVariables* (Section 5.8.1) and *Properties* (Section 5.8.2. **MethodInstanceDeclarations** are covered well through Section 5.9. In comparison to the authors of [82] the proposed concept of this thesis does not cover pre- and postconditions of methods. Eventually, **Instances** (Section 5.10) are mapped very well to OWL individuals also including constraints.

In conclusion, this thesis presents a solution for the research challenge (C2) interoperability on the semantic layer. The three identified open research points: InstanceDeclarations, OPC UA constraints, and coverage of OPC UA semantics are addressed very successfully within this thesis. An interesting finding of the mapping is that several semantic concepts in OPC UA can be mapped to a single semantic concept in OWL. OWL prevents duplicate concepts through the underlying mathematical model of the language. In contrast, OPC UA is not grounded in formal semantics,

Requirements \ Research approaches	this thesis
Namespaces	++
Attributes	+
DataType	+
ReferenceType	++
ObjectType	++
VariableType	++
InstanceDeclaration	++
MethodInstanceDecl	+
Instances (Object, Variable, ...)	++
Sum (27):	24

Table 5.11 – Requirements and evaluation for OPC UA semantics (this thesis).

.

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

which makes it much harder to exactly define the boundaries between the different concepts and thus leaves plenty of room for different interpretations.

A further difference between OWL and OPC UA is the underlying assumption about the completeness of information. OWL is based on the open-world assumption (OWA) and does not imply anything if information is missing because the information could just be modeled somewhere else. This leads to the necessity to explicitly state in the ontology that a certain information is not modeled in this ontology and also nowhere else. In comparison, OPC UA follows the opposite philosophy and considers missing information by default as absent. Both approaches come with benefits and drawbacks. While the OWL approach is especially useful in distributed loosely coupled environments, the OPC UA approach is more useful if only a single device is considered. Of course, also in the future OPC UA might shift to a more distributed paradigm and has to rethink some of the initial design decisions.

Finally, the presented recursive mapping can be considered quite complex and also yields quite expressive OWL ontologies. However, the differences between a very simple and the proposed mapping become quite obvious if some of the other standardized OPC UA mappings are considered. For example, the OPC UA UML mapping of OPC UA Part 3. This mapping introduces a base UML class and defines inheritance relations between this base class and the eight different OPC UA *NodeClasses*. Notice, that OPC UA *Types* (e.g., *BaseObjectType*) are not modeled as their own UML classes, instead there are modeled as instances of the corresponding *NodeClass* UML class. While this mapping is quite simple and allows a fast generation of UML models for OPC UA this comes at a certain price. One drawback is the OPC UA *HasSubtype-ReferenceType*, which is modeled the same way as a *HasProperty-ReferenceType*, even due to the fact that this particular *ReferenceType* is the counterpart of the UML inheritance relation. While an UML modeling tool would be able to offer some further support in displaying class hierarchies and features like restriction inheritance, these features cannot be used for OPC UA *Type* hierarchies due to the simplistic UML mapping. Another example is the XML-based OPC UA *NodeSet* file (see also OPC UA Part 6). While XML offers concepts of how namespaces can be introduced and used in a standardized way, OPC UA defines its own proprietary way to express namespaces. Of course, also this leads to problems in the XML-based ecosystem because transformation tools are no longer able to identify namespaces correctly. In the end, all these shortcuts allow to generate very simple mappings from an OPC UA perspective but also restrict OPC UA from most of the benefits of the surrounding target ecosystems.

6 QUERYING OF OPC UA INFORMATION MODELS

Querying OPC UA information models is a dream of the OPC Foundation since the early days of OPC UA. The dream also has the name "OPC UA Query Service" as part of OPC UA Part 4. Finally, ten years later this chapter provides the first SPARQL-based prototype of such a service. Two different concepts are presented. The first concept (Section 6.3) highlights how a native SPARQL query executed against the OWL ontology of Chapter 5 could look like. In contrast, the second concept provides transformation rules to translate the "OPC UA Query Service" into SPARQL (Section 6.4). From an architectural point of view, the main targets of a query service are the aggregating layers [62, 19, 144, 143, 76] on edge- and cloud-platforms. However, the underlying technology also could be scaled down if necessary [26]. Besides practical considerations, like available stable implementations and expressivity, the main challenge always is the user-facing side of the interface. Only if queries can be formulated very easily the technology is used in practice. Also in this area, the Semantic Web technology stack offers concepts to improve the usability through visualization [129, 159, 105, 154, 3, 10] or even through concepts based on natural language like [73, 153, 37], which can be used to implement, for example, chatbots [2, 145, 83]. Parts of this chapter are also published in [139].

The remainder of this chapter is structured as follows:

Section 6.1 gives a brief architecture overview of the Java-based prototype.

Section 6.2 focuses on the overall design decisions like *Views*, security, and the mapping of special OPC UA concepts like programming against the *TypeDefinitionNode*.

Section 6.3 presents an example of how a native SPARQL query executed against an OWL ontology based on the transformation rules of Chapter 5 looks like.

Section 6.4 exemplifies the transformation of the OPC UA query language to SPARQL. This includes, for example, the mapping of OPC UA filters as well as scalability consideration of the concept for very huge aggregated OPC UA information models.

Section 6.5 provides an OPC UA information model based concept to implement efficient caches on aggregating servers.

Section 6.6 evaluates the validity of the presented concepts against the example queries of OPC UA Part 4 Annex B (complex examples) and against the corresponding research challenge. In addition, the section outlines further ideas and thoughts about querying OPC UA information models and also highlights some issues of the OPC UA query language.

6.1 Architecture and Demonstrator

In this section, an architecture to query OPC UA information models is discussed. Typically, an automation device, like a CNC machine, has more than one OPC UA device (e.g., a PLC and some drive controllers) which together form a machine. Even more important is the fact, that a factory has not only one single machine, instead most factories have a lot of similar machines. This leads to the question of how a potential query architecture must look like to also cover use cases like "Find all machines which are currently low on material A". During the study two main requirements to offer a simple and powerful query interface are identified: (1) Standardized semantics, which is introduced through *Companion Specifications* in OPC UA. This simplifies the formulation of queries by a huge amount because then the user does not have to formulate different queries for each machine of a different manufacturer; (2) An edge-/cloud-layer which aggregates the underlying OPC UA information models of the machines. This is necessary because an expressive query language also needs a lot of resources, which might not be available on all OPC UA devices. Figure 6.1 shows such an architecture, which allows connecting devices without the necessary resources for query to a device with a query-engine. On the left side of the picture several clients/apps are

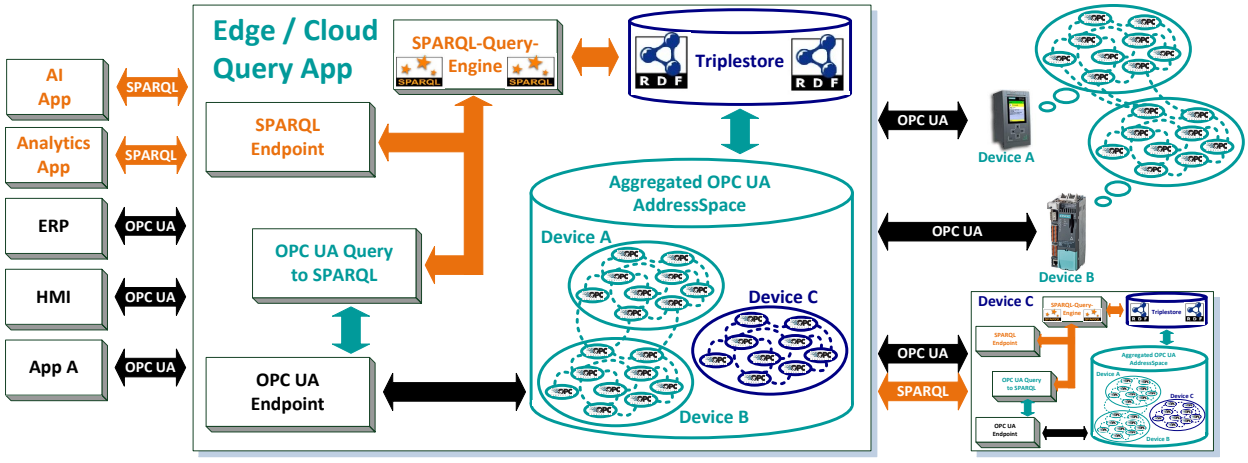


Figure 6.1 – Possible query architecture for the cloud-/edge-layer [139].

depicted, which want to query the information model. The prototype (the middle part of Figure 6.1) offers two different query languages for clients: SPARQL and OPC UA *Query*. Internally only the SPARQL query language is used and so OPC UA queries have to be translated to SPARQL queries. The SPARQL-Query-Engine then executes the query against the triplestore. The triplestore contains parts of the OPC UA information model in a triple format based on the OPC UA to OWL mapping (see Chapter 5). OPC UA information models can be categorized into two parts: The static part like the *Type-Hierarchy*, which is translated into triples and after that stored in the triplestore; The dynamic part like the *Value-Attribute* of a *VariableNode*, which is fetched on-demand directly from the aggregating OPC UA server. The aggregated OPC UA *AddressSpace* is synchronized with the underlying devices (which could also be another query application, see also Figure 6.1 right side) and offers access to the OPC UA graph, including live data for *Node-Attributes*. Nevertheless, static in this context only means that the static data is transformed into triples and synchronized with the triplestore. If the static data changes (e.g., the OPC UA graph structure is updated) also the triplestore must be updated. This can be achieved by using the *ModelChangeEvent* concept of OPC UA Part 3, instead of periodically browsing the whole graph for distinctions.

6.2 Design Decisions

In this section, the main design decisions of the native SPARQL interface is introduced. The native SPARQL interface also acts as the basis for the transformation of OPC UA *Queries* to SPARQL queries. Because of that, all design considerations of this section are valid for both approaches. The main key for a good SPARQL query interface is the underlying data model. For example, it makes a huge difference if OPC UA *References* are modeled as OWL object properties or as OWL data properties. The same is true for *BrowseNames* of OPC UA *InstanceDeclarations*. If the OPC UA data model is transformed in a certain way, it is possible to reduce the complexity to formulate queries by a huge amount. For the prototype, OWL ontologies are used including inferred knowledge based on OWL reasoners (see also Chapter 5 for a sketch of the transformation rules).

Views in OPC UA are introduced to offer different perspectives on a machine (e.g., for maintenance or monitoring purposes). If a certain *View* is selected only the *Nodes* which are contained in the *View* are be returned. In addition, it is also possible to restrict the visibility of *References* based on the selected *View*. To cover all these use cases *Views* are modeled with named graphs in SPARQL, which can be picked based on the *View-NodeId*. Nevertheless, introducing a new named graph for every *View* comes with the price of high resource consumption. Because of that, there exists also a more lightweight concept to introduce *Views* if different *Views* do not restrict the visibility of *References*. In this case, it is possible to identify the *Nodes* within a *View* based on an

additional "inView" OWL object property. This object property is automatically introduced between the *View-Node* and all the *Nodes* which shall be contained within the *View*.

Furthermore, **security** is an integral part of the OPC UA specification and is, of course, also part of OPC UA *Query*. The security features can be modeled in SPARQL similar to how *Views* are modeled. For example, for each role an own named graph could be generated, containing only the accessible *Nodes* and *References* for the group. Another idea is to automatically introduce additional SPARQL statements in each *Query* to also evaluate the access permission of a given client. However, these ideas are not implemented nor fully evaluated under the aspect of full coverage of all OPC UA security requirements within this thesis.

Another specialty of OPC UA are ***References* to non-existing *Nodes***. However, because *References* are modeled as OWL object properties in the OWL ontology the *TargetNode* must exist. To also support *References* to non-existing *Nodes*, shadow *Nodes* are introduced into the ontology, if the target *Node* is not defined. A shadow *Node* contains all *Attributes* which can be returned in a *ReferenceDescription*. Normal *Nodes* can be distinguished from shadow *Nodes* based on an OWL annotation property with the name "nodeExists" and the value "false" (see also Section 5.10).

OPC UA introduces a concept which is called ***programming against the TypeDefinitionNode***. In a nutshell, this concept is used to identify *Instances* of an *InstanceDeclaration*. Normally, such *Instances* are identified by their *BrowseName*. However, OPC UA also allows defining of multiple *Nodes* with the same *BrowseName* in the context of the same *Instance-Node*. In this case, the *TranslateBrowsePathsToNodeIds Service* of OPC UA Part 4 returns the *Node* which is based on the *InstanceDeclaration* as the first entry in a list. However, SPARQL does not offer a similar concept. Because of that, an additional *BrowseName* OWL object property is introduced into the ontology

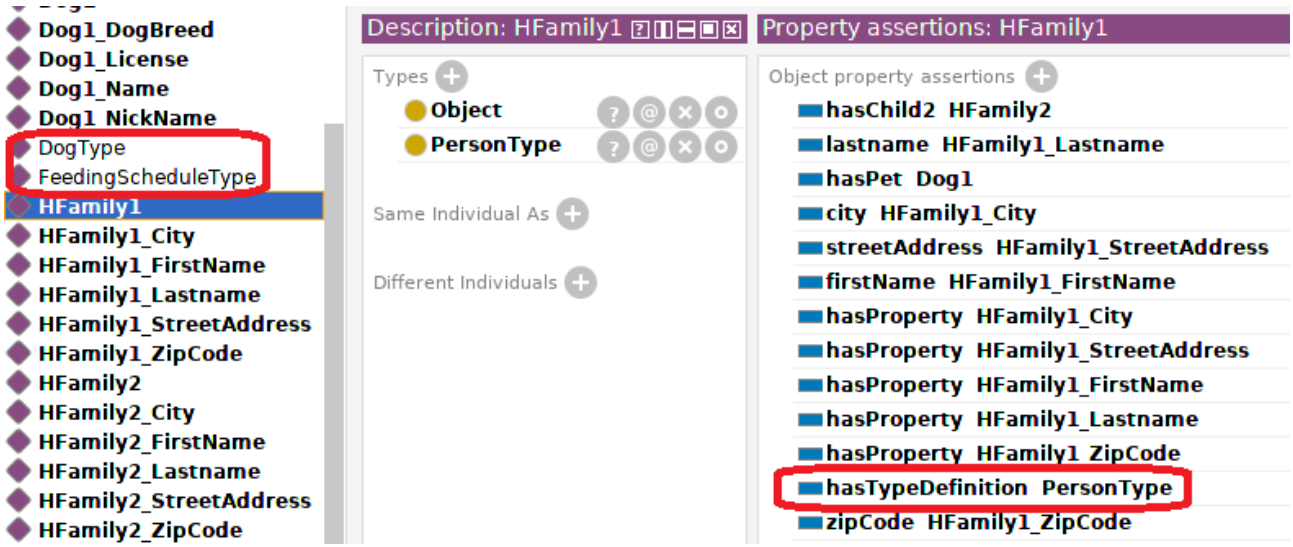


Figure 6.2 – OWL Punning for querying without subtypes on an inferred graph.

for most *InstanceDeclarations* (see also Section 5.4). This OWL object property not only allows to support the *Programming against the TypeDefinitionNode* concept, furthermore, it replaces three SPARQL filter statements through only one statement.

OPC UA *Query* also allows to only return ***Instances* of a certain *Type* without the inclusion of subtypes**. However, because the OWL ontology assigns up to five OWL classes per OWL individual the extraction of the OPC UA *Type* without subtypes is somehow challenging. To simplify this use case the OWL punning concept is used [115, 56]. Based on this concept it is possible to treat classes as instances of meta-classes. This means an OWL class can also be expressed as an OWL individual and so it is possible to introduce the "HasTypeDefinition" reference in OPC UA to allow easy queries without subtypes on the inferred graph (see also Figure 6.2).

6.3 Native SPARQL example

Previously, the design decisions for a native SPARQL interface are discussed. If these design decisions can be translated into a working prototype, SPARQL could be used to natively query OPC UA information models in a similar way than promised by OPC UA *Query*. The usability and expressivity of this mapping shall now be demonstrated based on Example B.2.6 of OPC UA Part 4 Annex B. The corresponding *Type* hierarchy and *Instances* of OPC UA Part 4 Annex B are presented in Annex A.3.2. Example B.2.6 outperforms all other examples in filter complexity and thus can be used as a perfect example to show the usability of the mapping (see also Figure 6.3). The textual form of the filter can be formulated in the following way: Find all *Instances* of PersonType, where a PersonType is connected to an AnimalType with a HasPet *Reference* and additionally the AnimalType must be connected to a FeedingScheduleType through a HasSchedule *Reference*. Furthermore, the PersonType *Instance* shall have a ZipCode-*Property* with the value "02138". Finally, the FeedingScheduleType shall have a Period-*Property* with the value "Daily" or "Hourly" and an Amount-*Property* with a value greater than "10" (Figure 6.3).

The corresponding *NodeTypeDescription* of example B.2.6 is shown in Table 6.1. The *QueryDataDescription* (**dataToReturn**) can be formulated in the following way: Return the Lastname

Table 6.1 – OPC UA Part 4 Example B.2.6 - *NodeTypeDescription* (NodeTypes[]) [77].

Type-DefinitionNode	**Include Subtypes**	**QueryDataDescription**	
		Relative Path	**Att.**
PersonType	FALSE	".12:Lastname"	value
		"<12:HasPet>12:AnimalType.12:Name"	value
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:FeedingScheduleType.12:Period"	value

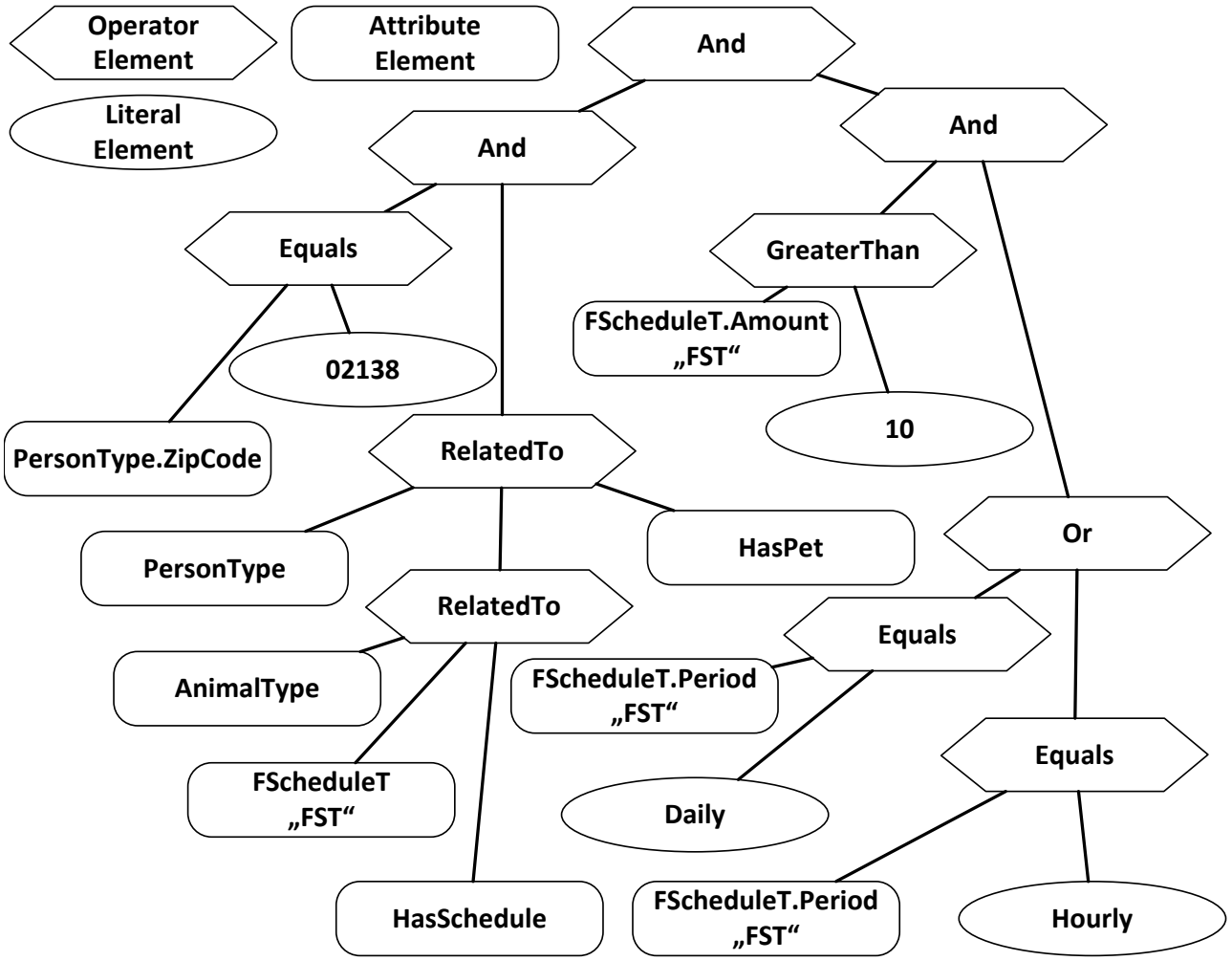


Figure 6.3 – OPC UA Part 4 Example B.2.6 - Filter [77].

Property of the PersonType *Instance* and the Name *Property* of the corresponding AnimalType *Instance* and the Period *Property* of the FeedingScheduleType *Instance*.

Figure 6.4 depicts how this query is formulated in SPARQL natively. Lines 1-3 of Figure 6.4 define OPC UA *Namespaces*, where *Namespace* "12" of OPC UA Part 4 Annex B is mapped to "query" (see also Section A.2.1). Line 3 defines the prefix for *Attribute* OWL data properties (see also Section 5.2) of the OPC UA to OWL mapping and Line 2 stands for the standard OPC UA *Namespace*. The filter statement is described with the Lines 7-12. The *QueryDataDescription* (**dataToReturn**) is depicted with the Line 5 and Lines 14-15. Notice that, in SPARQL it is possible to reuse filter statements in the result statement (e.g., the **periodValue** of Figure 6.4). The results (see the lower part of Figure 6.4) are exactly as specified by the OPC UA specification. Nevertheless, this SPARQL query is not totally equal to the corresponding OPC UA *Query*. For example, if the Lastname-*Property* for JFamily1 is not defined the whole query would fail, while in contrast, OPC UA *Query*

```
prefix query: <http://opcfoundation.org/UA/Examples/QueryPart4/>
prefix opcua: <http://opcfoundation.org/UA/>
prefix ia: <http://opcfoundation.org/UA/Meta/IA/>

SELECT DISTINCT ?nodeId ?typeNodeId ?lastnameValue ?nameValue ?periodValue
WHERE {
  ?animal a query:AnimalType. ?schedule a query:FeedingScheduleType.
  ?animal query:hasSchedule ?schedule. ?person a query:PersonType.
  ?person query:hasPet ?animal. ?person query:zipCode/ia:value ?zipCodeValue.
  Filter(?zipCodeValue = "02138"). ?schedule query:period/ia:value ?periodValue.
  Filter((?periodValue = "Hourly") || (?periodValue = "Daily")).
  ?schedule query:amount/ia:value ?amountValue. Filter(?amountValue > 10).

  ?person ia:nodeId ?nodeId. ?person opcua:hasTypeDefinition ?typeNodeId.
  ?person query:lastname/ia:value ?lastnameValue. ?animal query:name/ia:value ?nameValue.
}
LIMIT 25
```

QUERY RESULTS

Table | Raw Response

Showing 1 to 2 of 2 entries

	nodeId	typeNodeId	lastnameValue	nameValue	periodValue
1	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType	"Jones"	"Rosemary"	"Hourly"
2	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType	"Jones"	"Basil"	"Daily"

Figure 6.4 – Example B.2.6 - Native SPARQL Query with results (Apache Fuseki) [139].

would only return a null-value for the particular *QueryDataDescription*. The same behavior can easily be modeled through adding an OPTIONAL statement in SPARQL (e.g., OPTIONAL{?person query:lastname/ia:value ?lastnameValue.}). However, there are still some other major differences between the OPC UA *Query* of Example B.2.6 and the native SPARQL query of Figure 6.4, which are further discussed in Section 6.6.2.

6.4 OPC UA Query to SPARQL

In the previous section, insights are given on how some special aspects of OPC UA data models can be addressed with SPARQL directly. However, because most OPC UA stacks already support

the sending and receiving of OPC UA *Query* messages, the focus is now shifted to how OPC UA *Query* requests can be translated into SPARQL queries, starting with the mapping of OPC UA *FilterOperands* to SPARQL (Section 6.4.1). In the following, Section 6.4.2 introduces concepts to translate OPC UA *NodeTypeDescriptions* to SPARQL, while Section 6.4.3 highlights scalability considerations. Finally, Section 6.4.4 exemplifies how the mapping is applied based on an example.

6.4.1 OPC UA FilterOperands to SPARQL

The main focus of this section is the mapping of OPC UA *FilterOperands* to SPARQL expressions. Table 6.2 contains the complete *FilterOperator* list of OPC UA Part 4 and the corresponding SPARQL mapping. Notice that, most of the operators shall return "false" if the implicit conversion fails. This is, for example, modeled through a COALESCE statement. However, OPC UA *Query* also implicitly converts, for example, a *String*-value into a *Byte*-value (see also Annex A.3.2). This is not true for SPARQL. Because of that, additional algorithms are necessary to cover all implicit OPC UA

FilterOperator	**Operands**	**SPARQL Mapping**
Equals	2	$COALESCE((OP0 = OP1), false)$
IsNull	1	$!BOUND(OP0)$
GreaterThan	2	$COALESCE((OP0 > OP1), false)$
LessThan	2	$COALESCE((OP0 < OP1), false)$
GreaterThanOrEqual	2	$COALESCE((OP0 \geq OP1), false)$
LessThanOrEqual	2	$COALESCE((OP0 \leq OP1), false)$
Like	2	$COALESCE(REGEX(OP0, OP1), false)$
Not	1	$!OP0$
Between	3	$COALESCE((OP0 \geq OP1)\&\&(OP0 \leq OP2), false)$
InList	2..n	$COALESCE(((OP0 = OP1) \mid\mid (OP0 = OPn)), false)$
And	2	$(OP0\&\&OP1)$
Or	2	$(OP0 \mid\mid OP1)$
Cast	2	$OP1(OP0)$ (not complete)
InView	1	See Section 6.2
OfType	1	$TargetNode\ a\ OP0.$ $FILTER(OP0 = opc:ObjectType \mid\mid$ $OP0 = opc:VariableType \mid\mid EXISTS\{$ $OP0\ rdfs:subClassOf+$ $opc:ObjectType\} \mid\mid EXISTS\{OP0$ $rdfs:subClassOf + opc:VariableType\})$
RelatedTo	6	See Section 6.4.1
BitwiseAnd	2	not mapped
BitwiseOr	2	not mapped

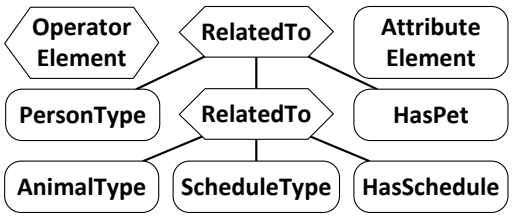
Table 6.2 – *FilterOperator* to SPARQL mapping.

```
SELECT DISTINCT ?result ?equal ?string ?byte ?castOk ?op1Type ?op2Type ?op1 ?op2
WHERE {
  BIND(xsd:string(10.0) as ?string)
  BIND(xsd:float(10.0) as ?byte)
  BIND(IF(DATATYPE(?string)=DATATYPE(?byte),true,false) as ?equal)

  BIND(IF(DATATYPE(?string)=xsd:double,0,
       IF(DATATYPE(?string)=xsd:float,1,
         IF(DATATYPE(?string)=xsd:long,2,-1))) as ?op1Type).
  BIND(IF(DATATYPE(?byte)=xsd:double,0,
       IF(DATATYPE(?byte)=xsd:float,1,
         IF(DATATYPE(?byte)=xsd:long,2,-1))) as ?op2Type).
  BIND((?op1Type>?op2Type) || (?op2Type>?op1Type) as ?castOk).
  BIND(IF(?op1Type>?op2Type,?string,
       IF(?op2Type=0,xsd:double(?string),
         IF(?op2Type=1,xsd:float(?string),
           IF(?op2Type=2,xsd:long(?string),?string)))) as ?op1).
  BIND(IF(?op2Type>?op1Type,?byte,
       IF(?op1Type=0,xsd:double(?byte),
         IF(?op1Type=1,xsd:float(?byte),
           IF(?op1Type=2,xsd:long(?byte),?byte)))) as ?op2).

  BIND(IF(?equal,?string=?byte,
       IF(?castOk,?op1=?op2,"")) as ?result).
}LIMIT 25
```

Listing 6.1 – Example SPARQL algorithm for implicit conversion of a String to Float.

Query conversion rules. An example of such an algorithm is depicted in Listing 6.1. This algorithm converts a *String*-value into a *Float*-value. However, not all implicit conversions can be supported by this algorithm. For a similar reason, the cast operator cannot be fully supported because the data type model of OPC UA is extensible, while the OPC UA to OWL mapping is limited to certain XSD types (see also Section 5.3). Finally, the *BitwiseAnd* and *BitwiseOr* filter operators also have no direct counterpart in SPARQL.

The RelatedTo filter operator (see also Annex A.3.2) contains up to six operands, which sometimes lead to large SPARQL representations (e.g., if Operand[3] is "0"). Table A.25 shows the definition of the different operators. In Figure 6.5 an example is given how a complex *RelatedTo* filter operator can be translated to a SPARQL expression. The left side of the figure depicts an example filter in the OPC UA graphical notation of OPC UA Part 4. The filter of Figure 6.5a can be formulated textual in the following way: Find all *Instances* of PersonType, where the *Instances* are connected to an *Instance* of AnimalType with a HasPet *ReferenceType*. In addition, the AnimalType *Instance* must



(a) Example B.2.4 - Filter [77].

```
1 ASK
2 WHERE {
3   ?person a query:PersonType.
4   ?animal a query:AnimalType.
5   ?sched a query:ScheduleType.
6   ?animal query:hasSchedule{1} ?sched.
7   ?person query:hasPet{1} ?animal.
8 }
```

Listing (6.2) Example B.2.4 - SPARQL Filter.

Figure 6.5 – Example SPARQL mapping for the RelatedTo operator.

be connected to a ScheduleType *Instance* with a HasSchedule *ReferenceType*. As the first step, the lower *RelatedTo* operator shall be translated. Operand[0] is translated to the SPARQL expression of Line 4 (Listing 6.2), while Line 5 contains Operand[1]. Line 6 specifies the relationship between Operand[0] ("?*animal*"), Operand[1] ("?*schedule*"), Operand[2] ("*query* : *hasSchedule*"), and Operand[3] ("{1}"). Operand[3] specifies the number of hops and is set to the value "1" in this example. This means Operand[0] and Operand[1] should be directly related. Notice that, the formulation of Operand[3] is not part of standard SPARQL but is supported by some of the SPARQL engines like Apache Fuseki [12]. Based on the standardized COUNT expression it is also possible to formalize an equal statement with standard SPARQL expressions. Furthermore, if the hop count is set to "1" the "{1}" statement can also be omitted. If Operand[4] and Operand[5] are set to "true". If Operand[4] would be set to "false" the predicate of Line 3 and Line 4 ("*a*") would be replaced by the predicate "*opcua* : *hasTypeDefinition*", or the pattern has to be executed on a graph without inference. If Operand[5] is set to "false", Line 6 must be executed on a graph without inference. Since SPARQL allows to combine patterns of different graphs within one query,

```
1 SELECT DISTINCT ?sn
2 WHERE {
3   {SELECT ?sn ?tn (COUNT(?int1) as ?hops1) (COUNT(?int3) as ?hops3)
4     WHERE {
5       ?sn a query:PersonType. ?sn query:hasChild* ?int1.
6       ?int1 query:hasChild ?int2. ?int2 query:hasChild* ?tn.
7       ?tn a query:PersonType.
8       OPTIONAL{?int3 a query:PersonType. FILTER(?int1 = ?int3)}
9     } GROUP BY ?sn ?tn}
10  FILTER(?hops1 = ?hops3)
11 }
```

Listing 6.3 – Example mapping if the *RelatedTo* Operand[3] is set to "0".

it is also possible to combine the inferred graph with another graph without inference. In the next step, Operand[0] of the upper *RelatedTo* operator is expressed by Line 3, while Operand[1] is based on the *Instances* of the lower *RelatedTo* operator ("?*animal*"). Finally, Line 7 expresses the relationship between the other operands.

As specified in Annex A.3.2, if Operand[3] is set to "0" the semantics of the *RelatedTo* filter operator change drastically. In this case, an undefined number of hops shall be followed in the forward direction and each *Node* in the path shall be of the *Type* specified by Operand [0]. Listing 6.3 presents a concept of how such a *RelatedTo* filter operator can be translated into SPARQL. The basic idea is to define intermediate SPARQL variables (e.g., **?int1** and **?int2** of Listing 6.3), which allow checking for the corresponding *Types* of all intermediate *Nodes* on the path. The last open point is now to count the number of intermediate *Nodes* and compare this number with the number of intermediate *Nodes* of the specified *Type*. If both numbers are equal, each intermediate *Node* has the correct *Type* and the *Node* can be added to the result set.

The Like Operator returns "true" if the pattern defined in Operand[1] matches Operand[0]. Furthermore, OPC UA defines its own language about how patterns should be specified. Table 6.3 describes the mapping of OPC UA wildcard characters to RegEx characters, which can be used in SPARQL queries.

The Attribute Operand can be considered as some kind of super set of all operands supported by OPC UA *Query* (see also Annex A.3.2). Some of the transformation steps can also be reused for the *QueryDataDescription* (e.g., the transformation for the RelativePath). Listing 6.4 contains an example of the mapping rules. Note that, the given query of Listing 6.4 contains only parts of

```
1 ?nodeId a query:AnimalType. # NodeId
2 BIND(?node as ?alias_Name). # Alias
3 ?targetNode ia:value|ta:value ?valueAttribute. # AttributeID
4 # BrowsePath TargetName = TypeNodeId
5 ?targetNode a query:AnimalType.
6 # BrowsePath Relation
7 ?sourceNode opcua:organizes ?targetNode.   # IsInverse = false
8 ?sourceNode opcua:organizedBy ?targetNode. # IsInverse = true
9 opcua:organizes owl:inverseOf ?inverseRefType.
10 # BrowsePath TargetName = BrowseName
11 ?targetNode ia:browseName|ta:browseName ?browseNameValue.
12 FILTER(?browseNameValue = "browseName"^^xsd:anyURI).
13 # BrowsePath TargetName = empty
14 ?sourceNode ?refType ?targetNode. # Graph without inference
15 FILTER(?refType = opcua:organizes || # IncludeSubtypes = true
16   EXISTS{?refType rdfs:subPropertyOf+ opcua:organizes}).
```

Listing 6.4 – An example for the *Attribute* Operand.

OPC UA	SPARQL	Meaning
%	.*	To ensure "String starts with main" also ^main.* can be used, which does not match "something main something".
_	.{1}	
\		Escape characters must be interpreted by the parser before the RegEx transformation. In addition, also reserved RegEx chars must be escaped by the parser.
[]	[]	
[^]	[^]	

Table 6.3 – OPC UA wildcard characters to SPARQL mapping.

a SPARQL query. The **nodeId** parameter is translated based on Line 1 of Listing 6.4, depending on how the *NodeId* is expressed in the underlying OWL-mapping (see Chapter 5). Line 2 shows how the **alias** parameter can be translated to the BIND concept of SPARQL. However, it would be also possible to simply reuse the SPARQL variable. The **attributeId** parameter is mapped by Line 3 based on the predicate statement (e.g., "ia:value|ta:value"). This statement would return the *Value-Attribute* in this particular case. If, for example, the *BrowseName-Attribute* should be returned the predicate has to be changed to "ia:browseName|ta:browseName". Line 5 has to be introduced in a query if the **targetName** is a *NodeId* (e.g., "query:AnimalType") instead of a *BrowseName* (see also OPC UA Part 4). The Lines 7, 8, and 9 address the relation. The **referenceTypeId** parameter is encoded in the predicate statement (e.g., "opcua:organizes"). As already introduced in Section 5.4 asymmetric *ReferenceTypes* are automatically generate two OWL object properties. In some cases, the inverse name is well-known and can be directly entered (Line 8). However, it is also possible to retrieve the inverse OWL object property for a given *ReferenceType* based on Line 9. Lines 11-12 are present if the **targetName** parameter is a *BrowseName*. The **targetName** parameter itself is inserted in Line 12 (e.g., "browseName"). Finally, if the **targetName** is empty (only allowed in some cases) the *BrowseName* is not important and is not used for filter purposes. Line 14 identifies the modeled *References* and must be executed on a graph without inference. This is especially important if later on *ReferenceDescriptions* shall be returned. The next step is to filter the *References* based on the **referenceTypeId** parameter (Line 15-16). If subtypes shall be included Line 16 must be present.

The other possible filter operand parameters *Element*, *Literal*, and *SimpleAttribute* can be easily derived from the already discussed concepts and are not further explained in this thesis.

6.4.2 NodeTypeDescription to SPARQL

The *NodeTypeDescription* of the OPC UA *Query* service has two purposes: (1) Selecting the *Instances* based on a *Type* (**typeDefNode** and **includeSubtypes**); (2) Specifying the data which shall be

```
1 # TypeDefNode – IncludeSubtypes true
2 ?sourceNode a query:PersonType.
3 # TypeDefNode – IncludeSubtypes false
4 ?sourceNode opcua:hasTypeDefinition query:PersonType.
5 # DataToReturn
6 {} UNION {
7   ?sourceNode opcua:aggregates ?targetNode1.
8   ?targetNode1 ia:browseName|ta:browseName ?targetNode2.
9   FILTER(?targetNode2 = "BrowseName"^^xsd:anyURI).
10  ?targetNode1 ia:value|ta:value ?element0.
11 } UNION {
12  ?sourceNode ?refType ?targetNode3. # Graph without inference
13  FILTER(?refType = opcua:organizes || # IncludeSubtypes true
14    EXISTS{?refType rdfs:subPropertyOf+ opcua:organizes}).
15  ?refType ta:nodeId ?rdNodeId.
16  ?refType ta:isForward ?rdIsForward.
17  ?targetNode3 ia:nodeId|ta:nodeId ?rdTargetNodeId.
18  ?targetNode3 ia:browseName|ta:browseName ?rdBrowseName.
19  ?targetNode3 ia:displayName|ta:nodeId ?rdDisplayName.
20  ?targetNode3 ia:nodeClass|ta:nodeClass ?rdNodeClass.
21  ?targetNode3 opcua:hasTypeDefinition ?rdTypeDef.
22 }
```

Listing 6.5 – An example *NodeTypeDescription*.

returned (**dataToReturn**). Listing 6.5 exposes an example transformation. The **typeDefNode** is expressed by the Lines 2 and 4. If **includeSubtyes** has the value "false" Line 4 must be inserted. Lines 6-10 and Lines 11-21 depict two elements of a *QueryDataDescription*. Note that, each element is encapsulated in an UNION statement. In the end, this ensures that not the complete query fails if a single element in the *QueryDataDescription* fails. The former array element (Lines 6-10) contains the mapping for a defined *BrowseName*. The later array element (Lines 11-21) shows the transformation if the *relativePath* argument ends on a *Reference*. In this case, a *ReferenceDescription* shall be returned as result. Line 12-14 are used to fetch the modeled *References* for the given path. Line 12 must be executed on a graph without inference to ensure that only the actual modeled references are returned. If subtypes of the defined *ReferenceType* shall also be included Line 14 must be added. The Lines 15 and 16 extract the **referenceTypeId** and the **isForward** parameter based on an OWL object property. Line 17-21 are used to collect the remaining information for a *ReferenceDescription* (**browseName**, **displayName**, **nodeClass**, and **typeDefinition**).

```
1 SELECT DISTINCT ?subClasses ?subProperties
2 WHERE {
3   ?subClasses rdfs:subClassOf* query:AnimalType.
4   ?subProperties rdfs:subPropertyOf* opcua:hierarchicalReferences.
5 }LIMIT 25
```

Listing 6.6 – SPARQL based graph inference.

6.4.3 Scalability considerations

In the examples of the previous sections sometimes the inferred graph and sometimes the native graph has to be used. Normally, the inferred graph is the more useful graph and should also be the preferred graph for standard queries. Nevertheless, the inferred graph is generated by an OWL reasoner which might run into some limitations for very huge information models with high complexity. Further research must be carried out to exactly determine the outer bounds of this concept. However, it is also possible to slightly reformulate the SPARQL queries to get the same results as for an inferred graph on a native graph. Line 3 of Listing 6.6 shows how all OPC UA subtypes of the AnimalType, including the AnimalType itself, can be bound to the "?*subClasses*" SPARQL variable. The result on an inferred as well as on the native graph would be: AnimalType, CatType, DogType, and PigType for the information model of Annex A.3.2. Line 4 is the equivalent statement to include subtypes for a given *Reference*. Based on this concept it is also possible to easily formulate queries with subtype inclusion on a native graph (without inference) and thus allows to restrict scalability considerations to SPARQL itself. Companies like Amazon Web Services, for example, claim to be able to store billions of relations and access them on a millisecond basis with SPARQL (see also Amazon Neptune documentation [4]). Therefore, it can be assumed that the underlying technology stack, which supports SPARQL, is powerful enough to support even cloud-based OPC UA scenarios, where a whole factory should be queried. Another idea would be to distribute the work-load across several SPARQL endpoints through the standardized SPARQL federated query concept.

6.4.4 Example mapping from OPC UA Query to SPARQL

In the former sections, the mapping from OPC UA *Query* to SPARQL is specified based on several translations of the different service elements. In this section, an example is given how the single parts of the mapping can be used to form complete queries based on Example B.2.4 from OPC UA Part 4 (see also Annex A.3.2 for the definition of the OPC UA information model).

The *Content-Filter* of Example B.2.4 can be formulated in the following way: Find all *Instances* of PersonType, where the *Instances* are connected to an *Instance* of AnimalType with a HasPet

Table 6.4 – Example B.2.4 - *NodeTypeDescription* (NodeTypes[]) [77].

Type-DefinitionNode	Include Subtypes	QueryDataDescription	
		Relative Path	**Att.**
PersonType	FALSE	".12:Lastname"	value
		"<12:HasPet>12:AnimalType.12:Name"	value
		"<12:HasPet>12:AnimalType<12:HasSchedule> 12:Schedule-Type.12:Period"	value

ReferenceType. In addition, the AnimalType *Instance* must be connected to a ScheduleType *Instance* with a HasSchedule *ReferenceType* (see also Figure 6.5a).

The *QueryDataDescription* (**dataToReturn**) of Example B.2.4 can be formulated in the following way: Return the Lastname *Property* of the PersonType *Instance* and the Name *Property* of the corresponding AnimalType *Instance* and the Period *Property* of the ScheduleType *Instance* (Table 6.4).

```
SELECT DISTINCT ?nodeId ?typeDefinitionNode ?element0 ?element1 ?element2
WHERE {
  ?sn a query:PersonType. ?sn opcua:hasTypeDefinition query:PersonType.
  ?sn ta:nodeExists ?exists. Filter(?exists). BIND(?sn as ?sn1).
  OPTIONAL{?sn2 a query:AnimalType. ?tn2 a query:ScheduleType.
    ?sn2 query:hasSchedule ?tn2. BIND(BOUND(?sn2) as ?result1).}
  BIND(IF(?result1, ?sn2, "") as ?tn1). OPTIONAL{?sn1 a query:PersonType.
    ?sn1 query:hasPet ?tn1. BIND(BOUND(?sn1) as ?result0).}
  FILTER(?result0).
  ?sn ia:nodeId ?nodeId. ?sn opcua:hasTypeDefinition ?typeDefinitionNode.
  {} UNION {?sn opcua:aggregates ?tn3.
    ?tn3 ia:browseName|ta:browseName ?tn4.
    FILTER (?tn4 = "http://opcfoundation.org/UA/Ex/Lastname"^^xsd:anyURI).
    ?tn3 ia:value | ta:value ?element0. # PersonType–LastName–Property
  } UNION {?sn query:hasPet ?tn5. ?tn5 a query:AnimalType.
    ?tn5 opcua:aggregates ?tn6. ?tn6 ia:browseName | ta:browseName ?tn7.
    FILTER (?tn7 = "http://opcfoundation.org/UA/Ex/Name"^^xsd:anyURI).
    ?tn6 ia:value | ta:value ?element1. # AnimalType–Name–Property
  } UNION {?sn query:hasPet ?tn8. ?tn8 a query:AnimalType.
    ?tn8 query:hasSchedule ?tn9.
    ?tn9 a query:ScheduleType. ?tn9 opcua:aggregates ?tn10.
    ?tn10 ia:browseName | ta:browseName ?tn11.
    FILTER (?tn11 = "http://opcfoundation.org/UA/Ex/Period"^^xsd:anyURI).
    ?tn10 ia:value | ta:value ?element2. # ScheduleType–Period–Property
  }
}LIMIT 25
```

Listing 6.7 – Example B.2.4 - OPC UA *Query* to SPARQL mapping.

Annex B of OPC UA Part 4 also specifies the results which should be returned for the query executed against the information model of Figure A.4. In the following, the SPARQL representation of Example B.2.4 (see also Listing 6.7) is highlighted. Notice that, the SPARQL COALESCE statements of Table 6.2 are omitted for readability purpose because no implicit casts are necessary. Line 3 expresses the **typeDefNode** parameter of the *NodeTypeDescription*. In this case *Instances* of PersonType shall be returned without subtypes. However, because the PersonType defines no subtypes the result **includeSubtypes** parameter has no effect at all. Line 4 is not discussed in the previous sections and has to be introduced to exclude shadow *Nodes* from the results (see also Section 5.10). The filter is translated in form of the Lines 5-9. It should be mentioned, that the translation differs a little bit from the translation of the *RelatedTo* mapping definition of Section 6.4.1. The lower *RelatedTo* operator is transformed to the Lines 5 and 6, while the upper *RelatedTo* operator is marked by the Lines 7 and 8. Both operators are encapsulated by SPARQL OPTIONAL statements, generate an additional boolean value ("?result0" and "?result1"), and are interconnected through SPARQL BIND statements. Finally, the filter statement is evaluated in Line 9. The reason for this is grounded in SPARQL and the fact that between each statement automatically and *And* connection is inserted. If, for example, both *RelatedTo* operators would be interconnected through an *Or* operator instead of the direct combination, the query should succeed if any of the *RelatedTo* operators return "true". However, if the statements would not be encapsulated in SPARQL OPTIONAL clauses the whole query would fail if only one *RelatedTo* operator returns false. Line 10 assigns the **nodeId** and **instanceTypeDefNode** parameter for each **queryDataSets** entry. The three **dataToReturn** elements of the **nodeTypes** parameter are depicted by the Lines 11-24. Each of the elements is encapsulated in a SPARQL UNION clause because OPC UA *Query* shall also return results if a single element fails. Finally, Line 25 shows how the maximum number of results can be restricted in SPARQL. Notice that, this statement cannot be directly mapped to the parameters **maxDataSets** and **maxReferences** because in SPARQL exists no service similar to *QueryNext*. It would be possible to combine the SPARQL LIMIT statement with the SPARQL OFFSET statement to implement some kind of pagination in SPARQL. Nevertheless, this only works in combination with an additional SPARQL ORDER BY statement and a more or less static graph. Figure 6.6 shows the result if the query is executed against the example information model of Figure A.4. The results exactly match the results which shall be returned for query B.2.4 according to OPC UA Part 4 Annex B.

In conclusion, besides the few restrictions on some of the operators explained in previous sections, most of the features of OPC UA *Query* could be covered and thus allows fast prototyping of OPC UA *Query*. Moreover, SPARQL supports additional constructs like "IF"-statements, aggregation, subqueries, and also federated queries, which are currently not available in OPC UA *Query*.

Showing 1 to 10 of 10 entries

	nodeId	typeDef	element0	element1	element2
1	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType			
2	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:42"^^xsd:anyURI	query:PersonType			
3	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType	"Jones"		
4	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:42"^^xsd:anyURI	query:PersonType	"Hervey"		
5	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType		"Rosemary"	
6	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType		"Basil"	
7	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:42"^^xsd:anyURI	query:PersonType		"Oliver"	
8	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType			"Hourly"
9	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:30"^^xsd:anyURI	query:PersonType			"Daily"
10	"http://opcfoundation.org/UA/Examples/QueryPart4/i=1:42"^^xsd:anyURI	query:PersonType			"Daily"

Figure 6.6 – Example B.2.4 - Results (Apache Fuseki) [139].

6.5 Synchronization of OPC UA graphs

To synchronize two OPC UA graphs several steps must be executed.

The first step is the registration for *ModelChangeEvents*. *ModelChangeEvents* are introduced by OPC UA Part 3 and can be used to notify clients about structural changes in OPC UA graphs. A structural change is, for example, adding or deleting of *References* or *Nodes*. In addition, *ModelChangeEvents* are also emitted if the *DataType Attribute* of a *VariableType* or *Variable* changes.

The second step is to copy the information model from the source OPC UA server to the target OPC UA server. This can be done in several ways: (1) Browsing the whole OPC UA information model with the *Browse* service; (2) Based on the *NamespaceFile* of the *NamespaceMetadataType*; (3)

Using a proprietary manufacturer-specific protocol. Each approach has its benefits and drawbacks. For example, the first approach can typically be applied to each OPC UA server but also comes with higher resource consumption. The second approach is more resource-efficient than the first one but is often not applicable to today's OPC UA server. The third approach could offer more performance as well as more functionality but offers the lowest interoperability. Especially the synchronization of *Views* is very resource inefficient with standardized OPC UA concepts. The only available standardized solution to synchronize *Views* is to traverse the whole OPC UA graph for each *View*. It should be noted, that it is not possible to retrieve the *View* information from the OPC UA XML *NodeSet*-File because there is no concept defined to express *View*-specific *References* for *Nodes*.

The third step is to keep both graphs synchronized based on *Events* (e.g., *ModelChangeEvent*) and *Subscriptions*. Nevertheless, according to the OPC UA specification, the synchronization is very expensive. The reason for that is because in general each *Attribute* of OPC UA must be considered as dynamic. For example, in most OPC UA information models the *BrowseName* and the *DisplayName* are static (e.g., *Type-Model*). Up to now, OPC UA offers no standardized machine-readable concept to express what kind of *Attributes* are dynamic (e.g., *Value-Attribute*) and which ones are static (e.g., *NodeId-Attribute*). In the following, a concept is introduced how OPC UA *Attributes* can be efficiently synchronized. The concept consists of two main parts: (1) A concept of how static/dynamic *Attributes* can be expressed within OPC UA information models; (2) A concept of how changes in the static *Attributes* can be communicated.

Static/Dynamic Attributes: Each *Namespace* in OPC UA has a corresponding *NamespaceMetaDataType Instance* to provide further information for the given *Namespace*. This standardized

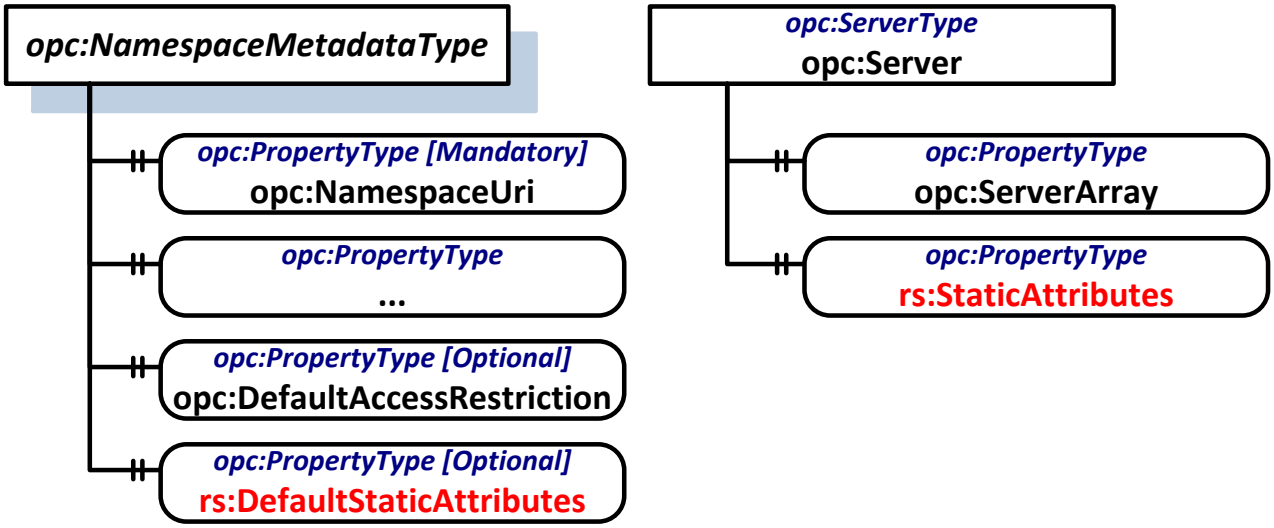


Figure 6.7 – DefaultStaticAttributes-Property and StaticAttributes-Property.

ObjectType shall be extended with a new standardized "DefaultStaticAttributes" *Property* (See Figure 6.7 left side). The *Value-Attribute* of this *Property* contains a bitmask, which marks what kind of *Attributes* are static. The bitmask shall be defined identically to *AttributeWriteMask* of OPC UA Part 3. The specified configuration automatically applies to all *Nodes* of the *Namespace*. However, to override default values an additional standardized "StaticAttributes" *Property* can be added to each *Node* (See also Figure 6.7 right side). If the "DefaultStaticAttributes" *Property* shall be overridden for *Properties*, this could be done through a special "StaticAttributesType" *VariableType*, which could be defined in a similar way.

Changes in static Attributes: The term static in this case means mostly static. For example, it can be expected that in most cases the *DataType-Attribute* never changes or only very seldom. *Attributes* which fall below this category shall also be marked as static, even if changes are possible. Based on this concept it is possible to also cache such *Attributes* without additional synchronization overhead. Nevertheless, because of this approach, an additional concept is needed to communicate changes in static *Attributes*. Such changes shall be communicated with newly introduced *Events*, which are similar to the *ModelChangeEvent* of OPC UA Part 3 (see also Figure 6.8).

BaseStaticAttributesChangeEvents are *Events* of the BaseStaticAttributesChangeEventType. The BaseStaticAttributesChangeEventType is the base type for StaticAttributesChangeEvents and does not contain information about the changes, instead, only indicates that changes have occurred. Therefore, the client shall assume that any or all static *Attributes* of the *Nodes* may have changed.

GeneralStaticAttributesChangeEvents are *Events* of the GeneralStaticAttributesChangeEventType. The GeneralStaticAttributesChangeEventType is a subtype of the BaseStaticAttributesChangeEventType. It contains information about the *Node* for which a static *Attribute* has changed. In the most generic version of this *Event* only the *NodeId* shall be transmitted. In a more detailed Version of this *Event* for each *NodeId* also a bitmask shall be transmitted, which marks the changed static

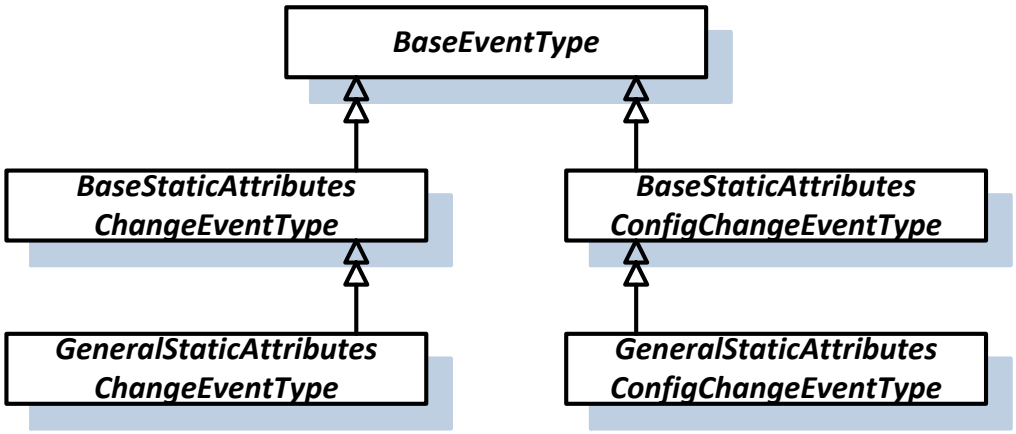


Figure 6.8 – Default Event Types for static Attribute changes.

Attributes. To also allow *Event* compression similar to the *ModelChangeEvent*, a StaticAttributesChangeEvent contains an array of changes.

BaseStaticAttributesConfigChangeEvents are *Events* of the BaseStaticAttributesConfigChangeEventType. The BaseStaticAttributesConfigChangeEventType is the base type for StaticAttributesConfigChangeEvents and does not contain information about the changes, instead, only indicates that changes have occurred. Therefore, the client shall assume that any or all *Value-Attributes* (bitmask) of "StaticAttribute" or "DefaultStaticAttributes" *Nodes* may have changed.

GeneralStaticAttributesConfigChangeEvents are *Events* of the GeneralStaticAttributesConfigChangeEventType. The GeneralStaticAttributesConfigChangeEventType is a subtype of the BaseStaticAttributesConfigChangeEventType. It contains information about the *Node* for which the *Value-Attribute* (bitmask) of "StaticAttribute" or "DefaultStaticAttributes" *Nodes* have changed. In the most generic version of this *Event* only the *NodeId* shall be transmitted. In a more detailed version of this *Event* for each *NodeId* also the new value of the bitmask (*Value-Attribute* of the *Node*) shall be transmitted. To also allow *Event* compression similar to the *ModelChangeEvent*, a StaticAttributesConfigChangeEvent contains an array of changes.

6.6 Evaluation

This section evaluates the validity of the presented concepts (including the OWL mapping of Chapter 5) against the example queries of OPC UA Part 4 Annex B (Section 6.6.1) and against the corresponding research challenge (Section 6.6.2).

6.6.1 OPC UA Part 4 Annex B

To demonstrate the validity of the approach all complex examples of OPC UA Part 4 Annex B are now mapped with the native SPARQL mapping as well as the OPC UA *Queries* transformation. Table 6.5 gives an overview of the evaluation results. It should be noted, that it is possible to execute all complex examples with both approaches without any problems. However, during the research of this thesis, several faults within OPC UA Part 4 Annex B could be detected and had to be fixed first before some of the queries could be executed. In addition, it is identified that not all examples always return the expected results if the OPC UA *Query* service is used. This is mainly due to some architectural decisions of OPC UA *Query*, which are discussed further in Section 6.6.2. Below, the most important findings for each of the nine example queries are discussed including also some details of the mapping itself. The details of the mapping are based on the native SPARQL mapping and return the same results as the OPC UA *Query* service would return for the information model of Annex A.3.2. Nevertheless, these queries are similar but not equal to the queries formulated in OPC UA Part 4 and, for example, do not make use of further parameter checks or the implicit

conversion algorithm of Section 6.4 to improve the readability of this thesis. The main purpose of these queries is to show how certain concepts of OPC UA *Query* can be translated into SPARQL.

Example B.2.4: This example is already used in Section 6.4 to exemplify the OPC UA *Query* to SPARQL transformation. The native SPARQL transformation is also straight forward and must not be further discussed. The reason why this example is marked as not correct is based on some typos in the *RelativePath* column (e.g., "12:Schedule" instead of "12:ScheduleType").

Example B.2.5 makes use of two different *NodeTypeDescriptions* and shows that an array can be received. Different *NodeTypeDescriptions* are modeled through a SPARQL UNION statement. Listing 6.8 depicts the native SPARQL query. The Lines 3-6 model the filter and the return data for the PersonType while Lines 8-11 are responsible for the CatType. The transformation rules for two *NodeTypeDescriptions* in the OPC UA *Query* to SPARQL mapping are based on the same basic concept. However, the query would still look different because, for example, the filter cannot be automatically divided into two UNION statements. Because of that, the full filter is present in both UNION statements.

Example B.2.6 is already discussed in Section 6.3 for the native SPARQL mapping. Also in this example, some copy and paste faults are identified. Furthermore, the usage of the alias concept seems to differ from Example B.2.8. Finally, also this very complex filter can be translated with the OPC UA *Query* to SPARQL mapping.

Table 6.5 – Evaluation based on OPC UA Part 4 Annex B.

Example	Query Properties	Spec. faulty	Native SPARQL	OPC UA Query m.
B.2.4	• Combined RelatedTo operator • Several dataToReturn statements • Multiple hops in dataToReturn statements	No	Yes	Yes
B.2.5	• Two NodeTypeDescriptions • More than one result is returned	Yes	Yes	Yes
B.2.6	• Very complex filter	No	Yes	Yes
B.2.7	• Operand[3] of RelatedTo operator >1	Yes	Yes	Yes
B.2.8	• Usage of the alias concept	No	Yes	Yes
B.2.9	• Return of a ReferenceDescription	No	Yes	Yes
B.2.10	• Return of a ReferenceDescription • Inclusion of a previous Browse result	No	Yes	Yes
B.2.11	• Usage of the View concept • Otherwise identical to B.2.5	Yes	Yes	Yes
B.2.12	• Usage of the View concept • Similar to B.2.5 • Nodes are requested through relative paths	No	Yes	Yes

```
1 SELECT DISTINCT ?nodeId ?typeNodeId ?lastnameValue ?nameValue
2 WHERE {{
3     ?person a query:PersonType. ?person2 a query:PersonType.
4     ?person query:hasChild ?person2.
5     ?person ia:nodeId ?nodeId. ?person opcua:hasTypeDefinition ↘
          ?typeNodeId.
6     ?person query:lastname/ia:value ?lastnameValue.
7   } UNION {
8     ?cat a query:CatType. ?schedule a query:FeedingScheduleType.
9     ?cat query:hasSchedule ?schedule.
10    ?cat ia:nodeId ?nodeId. ?cat opcua:hasTypeDefinition ?typeNodeId.
11    ?cat query:name/ia:value ?nameValue.
12 }}LIMIT 25
```

Listing 6.8 – OPC UA Part 4 Example B.2.5 - Native SPARQL Query.

Example B.2.7 specifies more than one hop for an *RelatedTo* operator. Section 6.4.1 explains in great detail how Operand[3] of an *RelatedTo* filter operator shall be translated. Based on these transformation rules this example can be translated very easily and is not further discussed within this thesis.

Example B.2.8 shows the usage of the alias concept. In Section 6.4.1 the mapping of the *Attribute* operand, also including the mapping of the alias concept, is explained. However, this query should not return any result at all. The reason for that is based on the filter, which enforces that a child has the same first name as a parent of theirs. OPC UA Part 4 Annex B specifies HFamily1 as a valid result for this query. Nevertheless, according to the definition of the *Equals* operator, the operands have to match completely. As depicted in Annex A.3.2, the first name of HFamily1-3 is "Paul", "Paul (Jr.)", and "Sara". Because the *String-Value* "Paul" cannot be considered equal to the *String-Value* "Paul (Jr.)" the query is not allowed to return any result for this information model. One solution would be to replace the *Equals* operator with a *Like* operator. However, the SPARQL mappings return the correct results in both cases.

Example B.2.9: The main focus of this example is to return *ReferenceDescriptions*. A sketch of how a *NodeTypeDescription* shall be translated is introduced in Section 6.4.2. However, Listing 6.9 provides more details based on Example B.2.9 of OPC UA Part 4 Annex B. Lines 4-5 model the filter, while Line 6 fetches the relations of Lines 7-8 from a named graph without inference. Line 9 depicts how the inverse *ReferenceType* can be identified if it exists (SPARQL OPTIONAL statement). Lines 10-13 are only to show how the same statement can be introduced in a generic way if the inverse direction should be followed. The Lines 14-17 extract the necessary information for the *ReferenceDescription* directly from OWL object properties or from the target *Node*. Notice that, this query does not return the same results as specified for Example B.2.9 because it is only partly

```
SELECT DISTINCT ?refTypeNodeId ?isForward ?targetNodeId ?browseName
?displayName ?nodeClass ?typeDef
WHERE {
  ?person2 a query:PersonType. ?animal a query:AnimalType.
  ?person2 query:hasAnimal ?animal. ?person a query:PersonType.
  ?person query:hasChild ?person2.
  GRAPH gr:native{
    ?person2 ?refType ?animal.}
  OPTIONAL{query:hasAnimal owl:inverseOf ?inverseRefType.}
  FILTER(?refType = query:hasAnimal
    || EXISTS{?refType rdfs:subPropertyOf+ query:hasAnimal}
    || ?refType = ?inverseRefType
    || EXISTS{?refType rdfs:subPropertyOf+ ?inverseRefType}).
  ?refType ta:nodeId ?refTypeNodeId.  ?refType ta:isForward ?isForward.
  ?animal ia:nodeId ?targetNodeId.  ?animal ia:browseName ?browseName.
  ?animal ia:displayName ?displayName.  ?animal ia:nodeClass ?nodeClass.
  ?animal opcua:hasTypeDefinition ?typeDef.
}LIMIT 25
```

Listing 6.9 – OPC UA Part 4 Example B.2.9 - Native SPARQL Query (not complete).

modeled to keep the focus on the most essential part of the mapping. However, based on this example it should now be very easy to formulate a query which returns the same results as Example B.2.9. Furthermore, also Example B.2.9 has several typos/copy and paste faults (including a wrong result table).

Example B.2.10 is used to provide an example of information model browsing in the filter statement. In general, this example does not really add new concepts which must be explained in greater detail. Nevertheless, besides some typos and copy and paste faults (including a wrong result table) the filter is also faulty (including the table as well as the graphical representation). Figure 6.9 shows the original filter as it is specified in OPC UA Part 4 Annex B. The red color marks the problem. According to OPC UA Part 4 the *RelatedTo* operator only accepts *NodeIds* or

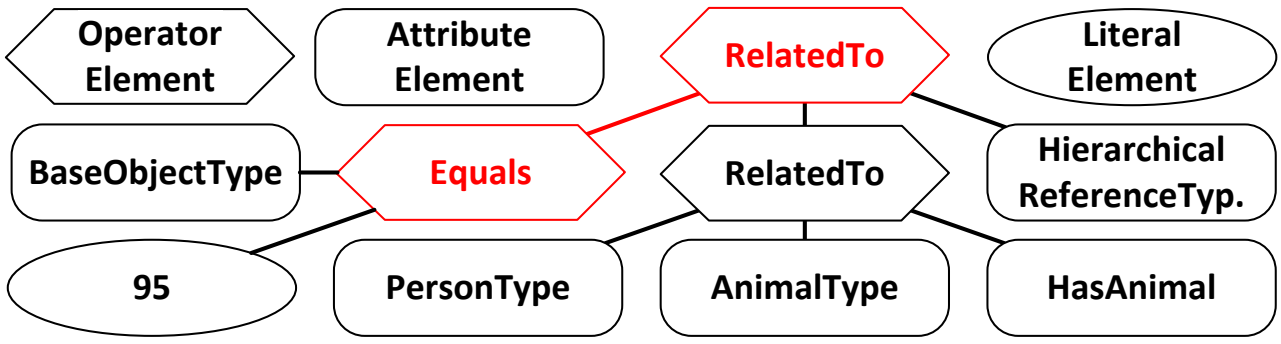


Figure 6.9 – OPC UA Part 4 Example B.2.10 - Filter [77].

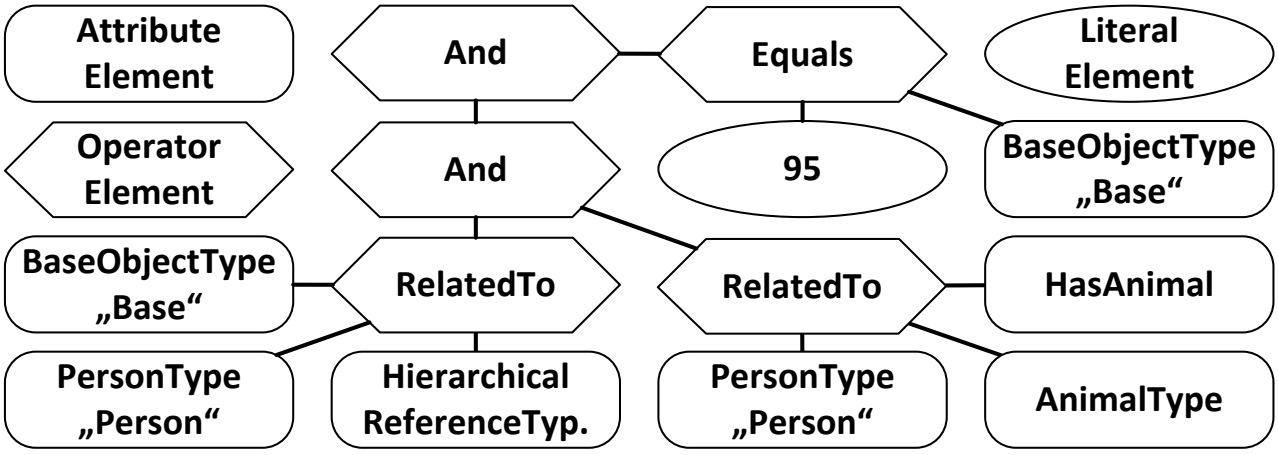


Figure 6.10 – OPC UA Part 4 Example B.2.10 - Corrected Filter.

ExpandedNodeIds of an *ObjectType* or *VariableType* as Operand[0] or Operand[1]. Furthermore, other *RelatedTo* operators can also be accepted as Operand[0] or Operand[1]. In contrast, an *Equals* operator returns a boolean value and, therefore, is not a valid operand neither for Operand[0] nor for Operand[1]. For the evaluation the filter is corrected in the following way: (1) The upper *RelatedTo* operator is deleted; (2) A new *RelatedTo* operator between the BaseObjectType and the PersonType is introduced and connected through an *Add* operator with the lower *RelatedTo* operator; (3) The *Equals* operator is connected to the *And* operator through another *And* operator; (4) Aliases are introduced for the BaseObjectType and the PersonType. The final filter is shown in Figure 6.10.

Example B.2.11 is besides the *View* parameter identical to Example B.2.5. As already explained in Section 6.2 *Views* can be modeled with named graphs, or with the additional "inView" OWL object property. The second approach is more lightweight but only works if the visibility of *References* is not defined by *Views*. Based on the results of Example B.2.12 it is possible to infer that in OPC UA Part 4 Annex B the *Views* do not hide *References* and because of that, the lightweight concept can be applied.

Example B.2.12 is similar to Example B.2.11 but has a slightly altered *NodeTypeDescription*. In this case, the *RelativePath* is used to traverse *Nodes* outside of the *View* (including the return of data). Finally, also Example B.2.12 has some typos and copy and paste faults.

6.6.2 Research challenge efficient querying of information

This section evaluates the proposed approach of this thesis against the formulated research challenge (C3) efficient querying of information. Within Section 3.3 the evaluation metrics for the given research challenge are formulated and existing research approaches are checked against these metrics. Based on this evaluation two unsolved problems are identified in the existing research

approaches: First, certain features of an OPC UA query language are not addressed at all like the **Filter**, **Limits**, and **Part 4 Annex B** requirements or covered only in parts like the **View** requirement. Second, features like the *NodeTypeDescription* can be covered well indirectly through the presented Semantic Web mappings. However, also this feature is not covered completely by any of the presented research approaches mainly due to the fact that a full-featured OPC UA query API is not the main goal for the presented approaches.

Table 6.6 shows the evaluation results of the proposed approach in this thesis. Section 6.2 presents different concepts to map **Views** to concepts like named graphs. Within Section 6.6.1 another concept for **Views** is shown. Eventually, **Views** are covered very well through the concepts introduced by this thesis. The **NodeTypeDescription** parameter is covered very well through Section 6.4.2. The **Filter** (Section 6.4.1), as well as the **Limit** (Section 6.4.4) parameter, are covered well but certain features could not be mapped completely like bitwise filter operations and the correct behavior of the **Limit** parameter for dynamic graphs. Finally, **Part 4 Annex B** is covered very well through the concepts of this thesis and is extensively discussed in Section 6.6.1.

In conclusion, this thesis presents a solution for the research challenge (C3) efficient querying of information. The two identified open research points: Missing query features and missing coverage of certain query features are addressed very successfully within this thesis. It is shown that with the introduced mapping it is possible to translate OPC UA *Queries* (with some restrictions) automatically to SPARQL queries. This concept can be used for rapid product development of the OPC UA *Query* service. Nevertheless, it is also highlighted how a native SPARQL query executed against an OPC UA information model can look like. Furthermore, a concept is introduced to synchronize OPC UA graphs. In the following, some crucial differences between the formulation of OPC UA *Queries* and native SPARQL queries, starting with the query of Section 6.4.4 are highlighted.

Research approaches / Requirements	this thesis
View	++
NodeTypeDescription	++
Filter	+
Limits	+
Part 4 Annex B	++
Sum (15):	13

Table 6.6 – Requirements and evaluation for OPC UA *Query* (this thesis).

Legend: ++ = very well (3), + = well (2), - = partly (1), - - = not possible (0), NA = Not Applicable (0)

In Example B2.4 of OPC UA Part 4 Annex B, the filter ensured that only *TypeDefinitionNodes* are considered where a person has a pet and this pet has a schedule. As a result, two pets are returned for Jones (Rosemary and Basil). Of course, both pets also have a schedule, which is hourly for Rosemary and daily for Basil. However, because OPC UA *Query* does not allow to define any dependency between two different **dataToReturn** statements, different result arrays must be considered independent (including the order of the results within the array). This means, that it is not possible to match the schedule period to the corresponding pet name, because it should also be allowed to reverse the order of the second result array without violating the OPC UA specification. This also becomes clearer if the fact is considered, that the *Browse* service of OPC UA is allowed to return the *References* of a *Node* in a different order for each call as long as not a special *ReferenceType* named "HasOrderedComponent" is used. If the assumption is made that the *Query* service does not analyze the **dataToReturn** statement for equal intermediary *Nodes* the *BrowsePaths* are evaluated separately and because of that, the result order might change.

Example B2.6 of OPC UA Part 4 Annex B (see also Section 6.3) has a very complex filter statement. However, in OPC UA the **filter** statement and the **dataToReturn** statement are only connected through the *Instance* of the *TypeDefinitionNode*. An example of the range of this architectural decision can be given by only changing the filter of Example B2.6 (see Figure 6.3) from $FScheduleT.Amount > 10$ to $FScheduleT.Amount > 50$. Surprisingly, the result would not change for OPC UA *Query*. The reason for this strange behavior is a consequence of the chosen OPC UA *Query* architecture. In the above case, the filter is no longer true for Rosemary, because the amount is below 50. However, the *Instance-Node* Jones is still a valid *Instance* because Basil fulfills all **filter** statements and so Jones is included in the result list. After the filtered *Instances* are determined the **dataToReturn** statement is applied against these *Instances*. In this case, also Rosemary is a valid target again, because the *BrowsePath* from Jones to Rosemary is still valid. Nevertheless, most people probably would have assumed that only Basil would be returned as result. In contrast, the native SPARQL query of Figure 6.4 would only return Basil, because in SPARQL it is possible to interconnect the **dataToReturn** statement with the **filter** statement. Furthermore, in Figure 6.4 also the period value can be mapped to the animal name because it is also possible to define dependencies between different **dataToReturn** statements in SPARQL.

Based on the results of this chapter, it can be stated that the formulation of native OPC UA *Queries* is not as easy as it looks. Several queries of OPC UA Part 4 Annex B probably have to be refined to ensure the expected behavior in each case. If the assumption is made that this annex was written by the only available experts for OPC UA *Query*, it can be inferred that new query users, which are not familiar with OPC UA at all, probably have a hard time with OPC UA *Query*. In addition, more than ten bugs within OPC UA Part 4 Annex B like the filter of example B.2.10, where a *RelatedTo* operator assigns a *Boolean*-value to operand[0], which is forbidden according to the *RelatedTo* definition are identified. In contrast, a concept is presented how OPC UA information

models can be queried with SPARQL natively. With the presented approach the size of the queries can be reduced (Example B.2.4 formulated in OPC UA *Query* based on the OPC UA C++ SDK of Unified Automation needs about 100 lines of code (see also [59])), as well as, the complexity of formulating queries because, for example, the filter statement can be directly interconnected with the result statement. Finally, all nine example queries of OPC UA Part 4 Annex B (complex examples) could be executed with the correct results for both approaches.

A further interesting research challenge for OPC UA based SPARQL is the handling of data streams. In contrast to the typical static SPARQL, parts of an OPC UA information model often represent senors values which are dynamic by nature. For example, in some use cases not only the actual value of a given sensor might be interesting, instead, the historical data should also be part of the SPARQL query. Several interesting concepts exist in this particular research area [93, 27, 20, 33, 9, 92, 90, 130, 28]. Another promising research area for OPC UA queries would be a better integration of arrays into SPARQL [6, 7].

7 SUMMARY, CONCLUSION, AND OUTLOOK

In this thesis, the efficient web access to OPC UA semantics is investigated. This chapter summarizes the findings presented in previous chapters and provides a conclusion for the underlying research goals (Section 7.1). Finally, possible directions for future research in this area are outlined (Section 7.2).

7.1 Summary and Conclusion

The fourth industrial revolution promises to provide mass production for highly configurable and individualized products based on four design principles: interconnection, information transparency, decentralized decisions, and technical assistance [67]. Within this thesis, a concept is explored how one of the most important key technologies for the fourth industrial revolution, OPC UA, could be combined with web technology to provide solutions for some of the Industry 4.0 challenges like interconnection and information transparency.

Web access to OPC UA information models: Within Section 3.1 several research challenges are identified to enable the integration of OPC UA into the World Wide Web.

First, statelessness is not addressed correctly in the actual research and still is an open research challenge. Second, most of the current research struggles with HATEOAS principles (Uniform interface). Third, a lot of OPC UA services cannot be accessed with REST APIs.

These challenges are addressed throughout Chapter 4. In the beginning, the concept behind OPC UA sessions is explained. In a nutshell, the OPC UA concept behind the *NamespaceArray* always introduces a state between client and server. Based on the fact that the *NamespaceArray* concept is used for addressing in OPC UA (*NodeIds*), most services cannot be considered stateless. This challenge is addressed through a concept for session-less service calls which is also contributed to the standardization (see also Section 4.2.2). Section 4.4 introduces the HTTP mapping and the HATEOAS-inspired resource representation with embedded hypermedia control elements.

In combination with Section 4.6 the second challenge, HATEOAS principles, is covered. This is achieved through the correct usage of HTTP verbs for different services. For example the HTTP verb "GET" is used for the *Read* service and the HTTP verb "PUT" is used for the *Write* service. Last but not least, Section 4.6 presents features like web browser support for good integration into the web ecosystem. The last identified research challenge, service coverage, is mainly tackled through Section 4.3 and Section 4.5. These sections present concepts on how some of the more difficult services can be mapped to REST paradigms. Section 4.5 even provides a concept of how group-subscriptions can be introduced in a resource-efficient RESTful way. Eventually, the chapter closes with a demonstrator and a detailed evaluation of the presented concepts.

In conclusion, the presented mapping can be considered the first REST mapping of OPC UA, which covers all requirements of REST and OPC UA and can be used as a building block for the design principle of interconnection. The validity of the approach is proven through several implementations as well as in form of contributions to the OPC UA specification itself. Finally, the evaluation results show that the identified research challenges, as well as the research goal, are addressed very well through the concepts of this thesis.

Semantics of OPC UA information models: Section 3.2 presents the identified research challenges to enable the integration of OPC UA into the Semantic Web.

First, the concept behind *InstanceDeclarations* is not covered at all in actual research. Second, the modeling of OPC UA constraints is only covered in parts by some researchers. Third, a complete translation of all OPC UA semantics into the Semantic Web is not presented in one single concept.

The first challenge is addressed through Section 5.7, Section 5.8, and Section 5.9. These sections present concepts to identify *InstanceDeclarations* and concepts to cover and translate the semantics in a generic way to an OWL ontology. The modeling of *InstanceDeclarations* as well as the corresponding modeling constraints are quite complex and are not obvious for data modeling experts without deep OPC UA knowledge. The complete mapping part of Chapter 5 from Section 5.1 to Section 5.11 also contains constraints for the different semantic concepts and thus provides answers for research challenge two. The third challenge, complete translation, is also covered through the mapping part of Chapter 5. However, the mapping also has certain limitations, for example, aspects like arrays and the handling of time series data could be improved further by other researchers. In summary, the mapping provides a huge coverage for the constraints as well as for the semantic concepts. Furthermore, the presented mapping can be used to easily display OPC UA ontologies within typical OWL ontology editors like Protégé (including the correct display of type hierarchies) or can be imported into SPARQL-endpoints and searched with a powerful query engine. In addition, the modeling constraints can also be validated with already existing reasoners of the Semantic Web ecosystem like HermiT [68]. Eventually, the chapter closes with a demonstrator and a detailed evaluation of the presented concepts.

In conclusion, the recursive mapping of OPC UA semantics to OWL of this thesis can be used to automatically translate the semantics of OPC UA information models (also including *Companion Specifications*) to OWL ontologies and thus provides a technological building block for the second design principle of Industry 4.0 - information transparency. The validity of the approach is proven through the implementation of a Java-based mapping tool and also successfully used in the query prototype of Chapter 6. Finally, the evaluation results show that the identified research challenges, as well as the research goal, are addressed very well in huge parts through the concepts of this thesis.

Querying of OPC UA information models: Within Section 3.3 several research challenges are identified to provide an efficient web-based query API for OPC UA semantics.

First, certain features of the OPC UA query language are not addressed at all like the *Filter* (e.g., find only persons with pets), *Limits* (only the first 100 results), and Part 4 Annex B requirements or covered only in parts like the *View* (e.g., maintenance or operation view) requirement. Second, features like the *NodeTypeDescription* (e.g., return the name and the address of the student) are only covered in parts but no complete mapping exists yet.

The two challenges are addressed throughout Chapter 6. The solutions of this chapter make use of the concepts and transformation rules of Chapter 5. Within Chapter 6 two solution paths are discussed and presented. The first one is the translation of OPC UA queries into SPARQL queries while the second one focuses on native SPARQL queries. The first research challenge, the OPC UA *Filter* parameter, is discussed in Section 6.4.1. Besides bitwise operations, all *FilterOperands* can be mapped to SPARQL. The *Limit* parameter is discussed in Section 6.4.4 and can only mapped in parts because the OPC UA concepts do not have a counterpart in SPARQL. *Views* are covered very well in Section 6.2 of this thesis. The different example queries of OPC UA Part 4 Annex B can be executed successfully with both approaches (OPC UA query and native SPARQL queries) and are discussed in greater detail within Section 6.6.1. The second research challenge, a complete mapping for the *NodeTypeDescription* parameter, is addressed very well in Section 6.4.2. Eventually, the chapter closes with a detailed evaluation of the presented concepts and a discussion on some of the identified problems around the standardized OPC UA query language and how these problems could be solved through native SPARQL queries.

In conclusion, this thesis presents the first valid concept (including a prototypical implementation) to query OPC UA graphs with the OPC UA query language. In addition, some issues of the OPC UA query language are presented and fixed with a novel concept to query OPC UA graphs directly with SPARQL. Such a feature could be used to identify, for example, all machines with a fault and detect if there is a configuration difference between the working machines (a possible building block for technical assistance). The approach is validated through the successful execution of all queries of OPC UA Part 4 Annex B (complex examples) with both concepts. Finally, the evaluation results show that the identified research challenges, as well as the research goal, are

addressed very well through the concepts of this thesis and only some small issues like bitwise filter operators remain unsolved.

7.2 Outlook

One idea behind the Industrial Internet of Things and also the World Wide Web is a large number of connected devices, which provide links to each other. In general, also the World Wide Web itself can be interpreted as a graph, where the websites act as nodes and the links as edges between the nodes. The main difference to today's OPC UA information models is the fact that *ExpandedNodeIds* are not used at all within most OPC UA information models. As long as this is not changed most of the available OPC UA servers are not interconnected on the data layer. In the future, it can be expected that this changes due to the fact that more and more *Companion Specifications* emerge, which describe a whole machine or process consisting of several OPC UA devices. This also increases the demand for cross-server browsing. Another interesting future topic is the introduction of a JSON-LD serialization into OPC UA. This would allow to easily combine RESTful node representations (Chapter 4) with the semantics of Chapter 5. Nevertheless, even if all these changes are applied to the current state of OPC UA it will take several years until industry products are adapted and used within factories. To speed up this process solutions have to be considered how already existing machines can be enabled with all these new features [45, 70, 140].

Parallel to this thesis version 1.04 of the OPC UA specification was released (including some contributions of this thesis already). However, the development of the OPC UA specification did not stop after version 1.04. For example, several further amendments emerged during the preparation of this thesis like the *Interfaces* and *AddIns* amendment or the *Dictionary* amendment. The *Interfaces* amendment introduces a completely new paradigm into OPC UA. Based on this concept each *Object* and *ObjectType* can now define several additional *InterfaceTypes*, including the inheritance of the semantics and constraints of these *InterfaceTypes*. In contrast, the *Dictionary* amendment defines concepts of how OPC UA can be used to include semantic tags from other standardization bodies like ECLASS [43] without the need to define corresponding *Companion Specifications*. Both of these amendments have a huge impact on how constraints and semantics are expressed in OPC UA and can only be covered in the OWL mapping through additional mapping rules. Furthermore, the presented query concept of this thesis could be further enhanced through concepts like natural language to SPARQL transformations. In the future, it might be even possible to combine speech recognition with query functionalities of OPC UA graphs. This would allow several new use cases for factory operators. Finally, up to now OPC UA is mainly used in operations use cases. Besides operation semantics, a lot of semantics is defined in the engineering toolchain

[91, 133]. These semantics could also be used to further improve manufacturing in the operation phase. One vision of academia and industry goes exactly in this direction and can be expressed with the term Digital Twin [119, 151, 152]. One aspect of a Digital Twin is to provide access to all information of an asset during the whole lifecycle and thus also includes information about product design (e.g., materials for recycling), simulation models, information about the supply chains, as well as even regulatory information (e.g., the CE sign or in the future the CO_2 footprint) to name only a few. Also in this case the integration of OPC UA data seems beneficial and therefore should be investigated in further research.

ANNEX A

This annex is structured as follows:
Section A.1 provides technical details for web access to OPC UA information models.
Section A.2 presents technical details for semantics of OPC UA information models.
Section A.3 presents some important parts of the OPC UA specification for this thesis.

A.1 Technical details for web access to OPC UA information models

This chapter is structured as follows:
Section A.1.1 provides a definition and generation rules for OPC UA URIs.
Section A.1.2 presents the mapping to HTTP verbs.
Section A.1.3 focuses on the header and query parameters.
Section A.1.4 highlights the concept details of some basic OPC UA services.
Section A.1.5 introduces two JSON schema representations.

A.1.1 URI Definition

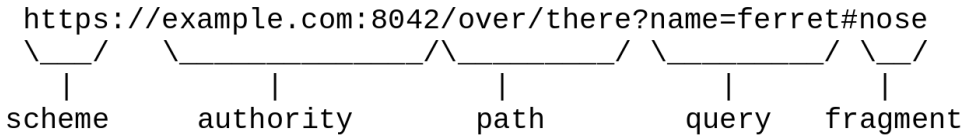


Figure A.1 – URI definition according to RFC 3986 [23].

Figure A.1 gives an overview of the generic URI definition of RFC 3986 [23]. Within Figure A.2 the definition of RFC 3986 is specialized for OPC UA. The **pathPrefix** is a vendor-specific prefix without any restrictions (e.g. “/Siemens/opcua/rest/”) (optional). The **apiVersion** is optional

and only acts as an example of how a version number could be introduced into an URI. The **apiVersion** could, for example, start with a "/V" followed by a number. The **uriVersion** always starts with a "/" followed by a number which represents the f the **urisVersion** *Property* of the OPC UA server (optional). The OPC Foundation shall never connect a *Node* with a hierarchical forward reference to the *Root-Node* and a *BrowseName* which starts with a number in the string part and *NamespaceIndex* "0". This ensures that the optional **uriVersion** can be distinguished from a valid *BrowseName*. The **opcPath** is the entry point into the OPC UA information model.

In the following sections, the serialization schema for *NodeIds* within URIs is further detailed (Section A.1.1.1) as well as the relation of the path structure to OPC UA Part 4 Annex A (Section A.1.1.2). Finally, Section A.1.1.3 defines the Augmented Backus-Naur-Form (ABNF) for the OPC UA URI definition and also introduces construction rules for the **opcPath** and **query** segment of Figure A.2.

A.1.1.1 URI NodeId identifier types

The encoding of *NodeIds* within URIs is detailed in Table A.1. OPC UA defines four different *NodeId* types: Numeric, String, GUID, and OPAQUE. While OPCAQUE *NodeIds* are mapped to a Base64 URL safe encoding [80] all other *NodeId* types are mapped to a string representation. The URL safe Base64 encoding can be generated in two steps: (1) Apply Base64 encoding; (2) Replace "+" with "-", "/" with "_", and "=" with "%3d".

IdentifierType Value	id	Description	Encoding
NUMERIC_0	i	Numeric	String
STRING_1	s	String	String
GUID_2	g	Globally Unique Identifier	String
OPAQUE_3	b	Namespace specific format	Base64UrlSafeEncoded

Table A.1 – Encoding of *NodeIds*.

Table A.2 presents some example encoding for different *NodeIds*.

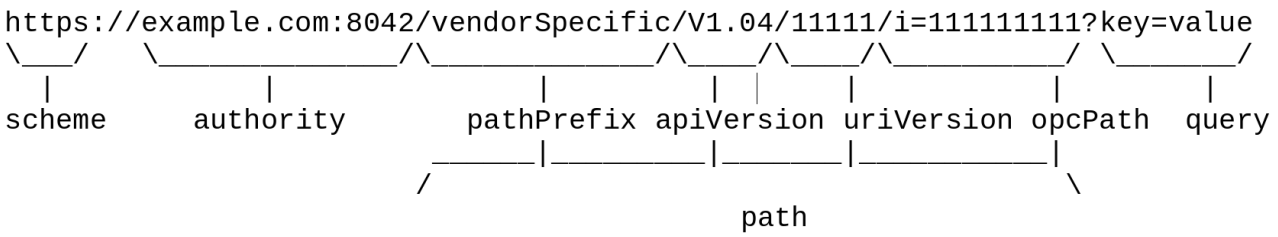


Figure A.2 – OPC UA URI definition for the REST mapping.

ServerIndex	node-id-type	NamespaceIndex	Identifier	Example
0	i	0	100	i=100
1	s	0	test	1s=test
1	i	1	100	1i=1:100
0	s	1	test	s=1:test
0	g	0	C496578A-0DFE-4B8F-870A-745238C6AEAE	g=C496578A-0DFE4B8F-870A-4745238C6AEAE
0	b	2	C496578A	b=2:C496578A

Table A.2 – Example *NodeIds*.

A.1.1.2 OPC UA RelativePath to REST RelativePath

OPC UA Part 4 Annex A defines a textual format for *RelativePaths* that can be used, for example, in the documentation. Some of the special characters used in OPC UA Part 4 Annex A are not URL safe and have to be replaced with URL safe characters (see Table A.3).

OPC UA	REST	Description
<	(	Start of a *Reference*
>	)	End of a *Reference*
#	@	Exclude subtypes
!	!	Follow inverse *References*
/	/	Follow any hierarchical *ReferenceType* in a forward direction
&	$	Escape character
.	.	Follow any subtype of the *Aggregates ReferenceType* in a forward direction
:	:	Separator for *NamespaceIndex*
=	=	Separator for *NodeId*
,	,	Separator for arrays

Table A.3 – Mapping of OPC UA Part 4 Annex A characters to URL safe characters.

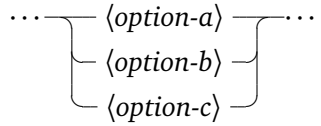
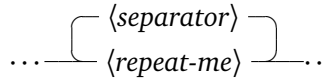
A.1.1.3 URI syntax diagram

The terms of this section are defined according to RFC 3986 [23], RFC 7230 [50], and RFC 7540 [51] using Augmented Backus-Naur-Form (ABNF RFC 5234 [36]) (see Grammar A.1) in combination with syntax diagrams.

The syntax diagrams of this section are constructed as follows:

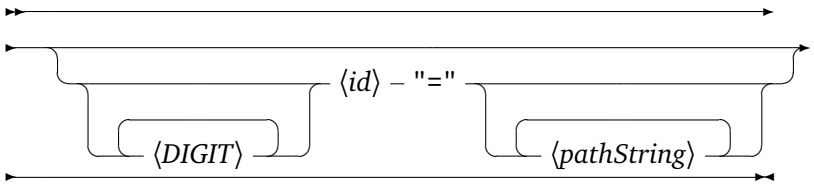
⟨*ALPHA*⟩ = A-Z / a-z
⟨*DIGIT*⟩ = 0-9
⟨*HEXDIG*⟩ = ⟨*DIGIT*⟩ / "A" / "B" / "C" / "D" / "E" / "F"
⟨*unreserved*⟩ = ⟨*ALPHA*⟩ / ⟨*DIGIT*⟩ / "-" / "." / "_" / "~"
⟨*gen-delims*⟩ = ":" / "/" / "?" / "#" / "[" / "]" / "@"
⟨*sub-delims*⟩ = "!" / "$" / "&" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "="
⟨*pct-encoded*⟩ = "%" ⟨*HEXDIG*⟩ ⟨*HEXDIG*⟩
⟨*opcua-reserved*⟩ = "(" / ")" / "@" / "!" / ":" / "=" / "/" / "$" / "," / "-"
⟨*sub-delims-query*⟩ = "!" / "$" / "'" / "(" / ")" / "*" / "+" / ";" / ":" / "@" / "/" / "?"
⟨*id*⟩ = "i" / "s" / "g" / "b"
⟨*pathString*⟩ = ⟨*unreserved*⟩ / ⟨*pct-encoded*⟩ / "$" / "'" / "*" / "+" / ";" / "&" / "=" / ","
⟨*key*⟩ = ⟨*unreserved*⟩ / ⟨*pct-encoded*⟩ / ⟨*sub-delims-query*⟩ / "," / "&"
⟨*value*⟩ = ⟨*unreserved*⟩ / ⟨*pct-encoded*⟩ / ⟨*sub-delims-query*⟩ / "="

Grammar A.1 – Definition of basic terms.

Construction	Meaning
	Start of syntax diagram
	End of syntax diagram
	Continued on next line
	Continued from previous line
	Alternatives: choose any one
	One or more items, with separators

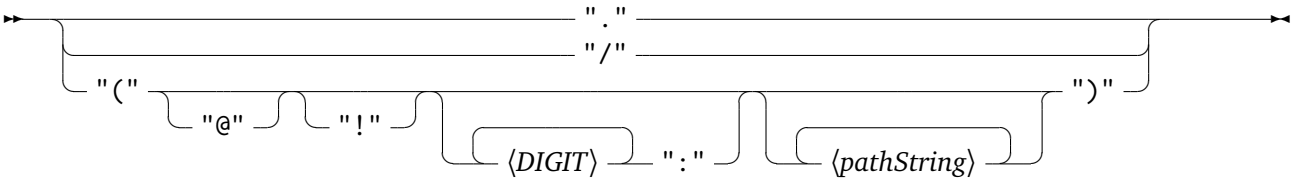
Grammar A.2 introduces the construction rules for the $< rootPath >$ element and introduces two possible branches: (1) If the $< rootPath >$ is empty the path shall start on the OPC UA *Root-Node*; (2) If a *NodeId* is specified the path shall start on the defined *NodeId*.

Grammar A.3 defines the construction rules for the $< reference >$ element and introduces three possible branches (see also Table A.3): (1) Any subtype of the *Aggregates ReferenceType* shall be followed in a forward direction; (2) Any hierarchical *ReferenceType* shall be followed in a forward direction; (3) The *ReferenceType* is specified in detail (including information about the



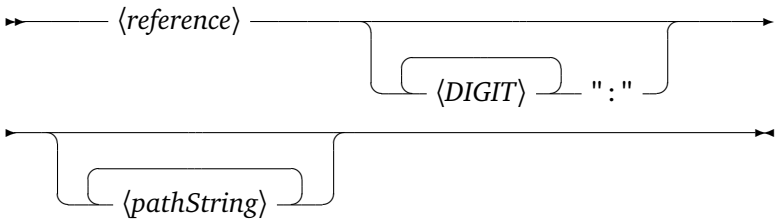
Grammar A.2 – Definition of <rootPath>.

direction and if subtypes shall be followed). Similar to OPC UA Part 4 Annex A the *BrowseName* of the *ReferenceType* is used to specify the *ReferenceType*.



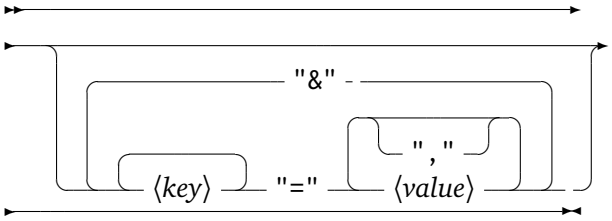
Grammar A.3 – Definition of <reference>.

Grammar A.4 outlines the construction rules for the $< browseElement >$ element, which consists of a $< reference >$ element followed by the *BrowseName* of the target *Node*.



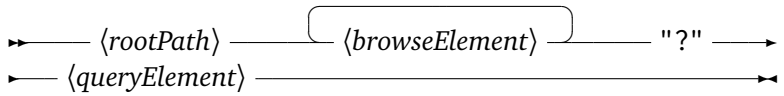
Grammar A.4 – Definition of <browseElement>.

Grammar A.5 presents the construction rules for the $< queryElement >$ element. The $< queryElement >$ element can contain several key-value pairs. Furthermore, it is possible to encode arrays in the value segment (separated through ",").



Grammar A.5 – Definition of <queryElement>.

Grammar A.6 depicts the construction rules for the complete $< opcPath >$ and $< query >$ element of Figure A.2. The $< opcPath >$ is constructed with the $< rootPath >$ and the $< browseElement >$, while the $< query >$ element is constructed through the $< queryElement >$. The $< opcPath >$ and $< query >$ element are separated through a "?".



Grammar A.6 – Definition of <opcPath> and <query>.

A.1.2 Mapping to HTTP verbs

The services which are marked with (+) in Table A.4 are introduced in addition to the already existing services. However, these services can be considered as an orchestration of OPC UA services and so, the implementation effort is very low. Each HTTP verb (first column) is mapped to one ore more OPC UA services (second column) with the correct semantics (e.g., the HTTP verb GET with idempotent and safe semantics to the OPC UA services *Read*, *Browse*, *BrowseNext*, etc.). Furthermore, for each OPC UA service the possible responses are analyzed and expressed through (several) newly introduced MIME-Types (third column). For example, the *Read* service can be used to return the *Value-Attribute* of a *Variable-Node*. In this case the content-type depends on the *DataType* of the *Variable-Node* (e.g., Boolean, String, etc.) and because of that several different MIME-Types can be returned.

HTTP Verb	**OPC UA Service**	**Representation MIME-Type**
GET	Read	app/opcua.Boolean+json app/pdf ...
GET	HistoryRead	app/opcua.HistoryReadResult+json
PUT	Write	app/opcua.Boolean+json app/pdf ...
PATCH	HistoryUpdate	app/opcua.HistoryUpdateResult+json app/json-patch+json
GET	Browse	app/opcua.NodeRepresentation+json
GET	BrowseNext	app/opcua.NodeRepresentation+json
GET	TranslateBrowse PathsToNodeIds	app/opcua.BrowsePathResult+json
GET	(+) ResolvePath	app/opcua.NodeRepresentation+json
POST	Call	app/opcua.CallRequest+json app/opcua.CallResult+json
POST	AddNode	app/opcua.CallRequest+json app/opcua.CallResult+json
DELETE	DeleteNode	app/opcua.StatusCode+json
PATCH	(+) Modify References	app/json-patch+json app/opcua.ModifyRefsResponse+json
POST	Query	app/opcua.CallRequest+json app/opcua.CallResult+json
GET	QueryNext	app/opcua.QueryNextResponse+json

Table A.4 – HTTP verbs and resource representations (app = application).

A.1.3 Header and Query Mapping

In some cases, HTTP already defines semantically identical headers marked with a "Yes" in the fourth column. A "No" means that this header is newly introduced for OPC UA. Notice, that the HTTP mapping of the OPC Foundation does not use all the headers defined in Table A.5 marked in the fifth column named "Std." (e.g., **Accept-Language** is not used - see also Section 4.2.3). If this column contains a "Yes" the header is part of the standardized HTTP mapping of the OPC Foundation and in case of a "No" the header is not defined in the standardized mapping. Some of

the most important HTTP headers are explained in greater detail, starting with the **Accept** header. The **Accept** header can be used to specify the serialization format (e.g, OPC UA Binary, XML, or JSON). The **Accept-Encoding** header can be used to request an additional encoding, for example, gzip. The **Content-Type** header is well-known on the web ecosystem and provides web clients with information about the transmitted content (e.g., a PDF document or an HTML webpage). In addition to the **Content-Type**, the **Content-Length** header specifies the size of the content. The **Content-Encoding** header is part of the response and specifies the applied additional encoding, for example, gzip. A server shall only apply an additional encoding if it is requested by the client but is not forced to do so. The **Accept-Language** header is used to specify the LocaleIds of the client. In RFC 2616 [48] the Accept-Language header can be weighted with a so-called quantity value. For example, the header value "de, en;q=0.9" requests a German representation but also accepts an English one if no German is present. In OPC UA, the client also is able to specify more than one *LocaleId*, where the most preferred one is the first entry in the array with descending order. This can easily be mapped to quantity values of RFC 2616 and vice versa. The **DataValue** headers are introduced to further enhance web browser support (the also Section 4.6.1). This ensures that the *Value-Attribute* from *Variables* can be delivered natively without any OPC UA-specific information and so can be interpreted by clients which do not understand OPC UA but understand how to display the *Value-Attribute* (e.g. pdf, JSON, HTML, ...). This allows to use OPC UA as a web-friendly transport protocol and thus enables new applications on top of OPC UA. An example use case might be the delivery of web pages, or pdf files (see also use case scenarios of Amazon S3 service). A webpage could also, for example, embedded a generic OPC UA client written in JavaScript.

Each header defined in Table A.5 can also be transferred as a query argument. This is especially useful in combination with redirects (e.g., from a Single-Sign-On service) and for more simple web applications that do not allow to set headers. If both parameters are present (query and header) the header value shall be discarded.

Each key-value pair in the query path must encode not allowed characters as specified in Annex A.1.1. If further links (e.g., href property) are part of the returned resource representation the query arguments provided by the client shall be also part of href URLs. This allows simple navigation with the web browser. A special precaution has to be taken for the **Authorization** query parameter. This query parameter shall only be present in the URL if the URL points to the local server (in addition transport encryption has to be used). This ensures that the security token is never leaked through external URLs (*ExpandedNodeId*). All values shall be encoded as string or string array. Of course, HTTPS has to be used if the **Authorization** parameter is present. However, because it is possible that the **Authorization** query parameter is stored in the web browser history such a token shall always have a limited lifetime.

Parameter name	DataType	HTTP header / Query key	New	Std.
authenticationToken	Session AuthenticationToken	Authorization	No	Yes
timestamp	UtcTime	Date	No	Yes
requestHandle	IntegerId	UaRequestHandle	Yes	No
returnDiagnostics	UInt32	UaReturnDiagnostic	Yes	No
auditEntryId	String	UaAuditEntryId	Yes	No
timeoutHint	UInt32	UaTimeoutHint	Yes	No
additionalHeader	Extensible Parameter AdditionalHeader	UaAdditionalHeader	Yes	No
dataEncoding	QualifiedName[]	Accept	No	No
	String	Content-Type	No	Yes
	UInteger	Content-Length	No	Yes
	String[]	Accept-Encoding	No	No
localeIds	LocaleId[]	Accept-Language	No	No
namespaceUris	String[]	UaNamespaceUris	Yes	No
serverUris	String[]	UaServerUris	Yes	No
DataValue				
statusCode	StatusCode	UaDataValueStatusCode	Yes	No
sourceTimestamp	UtcTime	UaDataValueSourceTimestamp	Yes	No
sourcePicoSeconds	UInteger	UaDataValueSourcePicoSeconds	Yes	No
serverTimestamp	UtcTime	UaDataValueServerTimestamp	Yes	No
serverPicoSeconds	UInteger	UaDataValueServerPicoSeconds	Yes	No
serviceResult	StatusCode	UaServiceResult	Yes	No
serviceDiagnostics	DiagnosticInfo	UaServiceDiagnostics	Yes	No
	String	Content-Encoding	No	No
serverStartTime	UtcTime	UaServerStartTime	Yes	No
diagnosticInfo	DiagnosticInfo	UaDiagnosticInfo	Yes	No

Table A.5 – HTTP header and query mapping.

A.1.4 Service Mapping

The goal of this section is to exemplify some of the most important and interesting aspects of the REST mapping. First, the *Read* service is detailed in Section A.1.4.1. In the following the *Browse* service (Section A.1.4.2) is explained and afterward the newly introduced *Next service (Section A.1.4.3) is highlighted. The last sections focus on the *Call* service (Section A.1.4.4) and the newly introduced ModifyReferences service (Section A.1.4.5).

HTTP Method	URI Template
GET	$<URLPREFIX>/\{ReadValueId.nodeId\}/\{AttributeName\}$

Table A.6 – URI template for the *Read* Service.

A.1.4.1 Read Service

The OPC UA *Read* service can be used to fetch the *Attributes* of an OPC UA *Node* like the *Value-Attribute* of the *Variable-NodeClass*. In Table A.6 the URI template for the *Read* service is shown (see also Annex A.1.1). The **AttributeName** is based on the symbolic names of OPC UA-*Attributes* (e.g., *BrowseName*, *DisplayName*, etc.). To avoid naming conflicts with the ResolvePath service of Section 4.6.5 the symbolic names of OPC UA-*Attributes* shall never be used as *BrowseNames* within the OPC UA-*Namespace*. Furthermore, three additional **AttributeNames** are defined: **RequestSchema**, **ResponseSchema**, and **Events**. The **RequestSchema** and **ResponseSchema** are already introduced in Section 4.4.5 and can be used to request schema descriptions for default forms and *Method-Nodes*. The **Events** parameter is used in combination with the *HistoryRead* service and is not further detailed in this work.

Table A.7 details the mapping of *Read* service request parameters to HTTP query arguments. The **Default Value** column defines the value for the parameter if the query argument is omitted. The default values are chosen in such a way that the workload and traffic is reduced for standard web clients. For example, if a standard web client (like a web browser) accesses the *Value-Attribute* of a *Variable-Node* additional application-specific headers are typically ignored and also not displayed to the user. Based on that assumption it is reasonable to only return, for example, timestamps if explicitly requested by the client. The *dataEncoding* parameter of the *ReadValueId* shall be extracted from the Accept-Header. If the Accept-Header does not match any of the known encodings from the server, the default encoding shall be used (see also Section 4.4.5).

Table A.8 details the mapping of *Read* service response parameters to HTTP headers and to the HTTP body. As already explained in Section 4.4.2 the HTTP version of the *Read* service is designed

OPC UA Arguments	Query Arguments	Default Value
requestHeader	see Section 4.4.2	
maxAge	maxAge	0
timestampsToReturn	timestampsToReturn	Neither
ReadValueId		
indexRange	indexRange	null
dataEncoding	see Section 4.4.2	

Table A.7 – *Read* service request arguments.

in a way that it is possible to directly display the content of the *Value-Attribute* in a web client like a web browser.

Listing A.1 presents an example HTTP request and the corresponding HTTP response. Line 1 contains the HTTP verb (in this case "GET"), the *NodeId* (in this case a numeric *NodeId* with the value "2005" in the OPC UA *Namespace*), the symbolic name of the *Attribute* (in this case with the value "DisplayName"), and an authentication token as query argument (in this case with the value "bearerToken"). Lines 2-5 are some headers provided by the client. Line 7 provides the HTTP response starting with the HTTP status code (in this case 200). Lines 8-12 are some of the headers, while Line 14 is the content of the *Value-Attribute* serialized with OPC UA JSON, as requested by the client through the Accept-Header (Line 4).

A.1.4.2 Browse Service

The OPC UA *Browse* service can be used to fetch the *References* of an OPC UA *Node*. In Table A.9 the URI template for the *Browse* service is shown (see also Annex A.1.1).

Table A.10 details the mapping of *Browse* service response parameters to HTTP headers and to the HTTP body. The *resultMask* parameter is extended with four additional parameters (not defined in OPC UA), which are further detailed: Bit 28 **BasicAttributes**, Bit 29 **ExtendedAttributes**, Bit 30 **Forms**, and Bit 31 **DefaultForms**. If the **BasicAttributes** bit is set all *Attributes* with exception of *DisplayName* and *Description* of the source *Node* is part of the representation. The **ExtendedAttributes** selects all *Attributes*. Regardless of the settings of **BasicAttributes** or **ExtendedAttributes** the *Value-Attribute* is included in the representation as link only. While the **BasicAttributes** setting is probably a good setting for automatic clients, the **ExtendedAttributes** setting might be the best solution for human-controlled clients. If **Forms** are selected then the representation contains forms, as described in Section 4.4.5 for each *Method-Node* of an *Object-Node*. The **DefaultForms**

OPC UA Arguments	HTTP Header Name	HTTP Body Type
responseHeader	see Section 4.4.2	
DataValue		
statusCode	see Table A.5	
sourceTimestamp	see Table A.5	
sourcePicoSeconds	see Table A.5	
serverTimestamp	see Table A.5	
serverPicoSeconds	see Table A.5	
diagnosticInfos	see Table A.5	
value		BaseDataType

Table A.8 – *Read* service response arguments.

```
1 GET /i=2005/DisplayName?opcuaAuthenticationToken=bearerToken HTTP/1.1
2   Date: 2002-10-10T00:00:00+05:00
3   UaAuditEntryId: <myId>
4   Accept: application/opcua+json
5   Accept-Language: de,en-US;q=0.7,en;q=0.3
6
7 HTTP/1.1 200 OK
8   UaServiceResult: 0
9   UaDataValueStatusCode: 0
10  Content-Type: application/opcua.LocalizedText+json
11  Content-Length: nnnn
12  Date: 2002-10-10T00:00:00+05:00
13
14  {"Text":"Server","Locale":"en"}
```

Listing A.1 – Example *Read* request and response (simplified).

HTTP Method	URI Template
GET	$<URLPREFIX>/\{BrowseDescription.nodeId\}$

Table A.9 – URI template for the *Browse* service.

enables all default forms for the given *NodeClass* (see also Section 4.4.5). Table A.11 shows what kind of default forms are presented based on the *NodeClass*.

Table A.12 details the mapping of *Browse* service response parameters to HTTP headers and to the HTTP body. The **NodeRepresentation** is explained in greater detail in Section 4.4.5.

OPC UA Arguments	Query Arguments	Default Value
requestHeader	see Section 4.4.2	
ViewDescription		
viewId	viewDescriptionViewId	null
timestamp	viewDescriptionTimestamp	null
viewVersion	viewDescriptionViewVersion	0
requestedMaxReferencesPerNode	requestedMaxReferencesPerNode	0
BrowseDescription		
browseDirection	browseDirection	Both
referenceTypeId	referenceTypeId	null
includeSubtypes	includeSubtypes	true
nodeClassMask	nodeClassMask	0
resultMask	resultMask	0xFF

Table A.10 – *Browse* service request arguments.

Default Form	Variable	VariableType	Object	ObjectType	ReferenceType	DataType	Method	View
NodeManagement								
AddNode	X	X	X	X	X	X	X	X
ModifyReferences	X	X	X	X	X	X	X	X
DeleteNode	X	X	X	X	X	X	X	X
View								
Browse	X	X	X	X	X	X	X	X
BrowseNext	X	X	X	X	X	X	X	X
TranslateBrowsePaths	X	X	X	X	X	X	X	X
RegisterNode	X	X	X	X	X	X	X	X
UnregisterNode	X	X	X	X	X	X	X	X
ResolvePath	X	X	X	X	X	X	X	X
Query								
QueryFirst		X		X				
QueryNext		X		X				
Attribute								
Read	X	X	X	X	X	X	X	X
HistoryRead	X	X	X	X				
HistoryNext	X	X	X	X				
Write	X	X	X	X	X	X	X	X
HistoryUpdate	X	X	X	X				
Session								
Cancel	X	X	X	X				

Table A.11 – Overview of default forms.

OPC UA Arguments	HTTP Header Name	HTTP Body Type
responseHeader	see Section 4.4.2	
diagnosticInfos	see Section 4.4.2	
results		NodeRepresentation

Table A.12 – *Browse* service response arguments.

Listing A.2 displays an example HTTP request and the corresponding HTTP response. Line 1 contains the HTTP verb (in this case "GET") and the *NodeId* (in this case a numeric *NodeId* with the value "100" and *NamespaceIndex* "1"). Lines 2-5 are some headers provided by the client. Line 5 specifies the *NamespaceURI* for *NamespaceIndex* "1" exactly as defined by *SessionlessInvoke* service of Section 4.2.2. In contrast, Lines 7-10 is equal to the request specified with Lines 1-5. The difference is, that in this case the *UrisVersion* is used (see Line 7 with the value of "1"). Lines 12-18 provide the response of the server. An example of a simplified NodeRepresentation can be found in Section 4.4.5.

A.1.4.3 *Next Service

The OPC UA **Next* service can be used to fetch the next results if not all results can be returned at once. In Table A.13 the URI template for the *Next service is shown (see also Annex A.1.1). The *Next service is based on a REST paradigm called pagination. This means that the URI to the next part of the result can be embedded directly into the NodeRepresentation of Section 4.4.5. Furthermore, with this pattern, all *Next* services of OPC UA can be handled with the same basic concept (e.g., *BrowseNext*, *QueryNext*, and also the *Next* function which is embedded into *HistoryRead*). For a complete stateless approach, the *ContinuationPoint* can include all necessary information to generate the corresponding representation. This is especially useful for static representations but also dynamic representations can be addressed by adding some kind of version

```
1 GET /i=1:100 HTTP/1.1
2   Date: 2002-10-10T00:00:00+05:00
3   UaAuditEntryId: <myId>
4   Accept: application/opcua+json
5   UaNamespaceUris: http://myTestNamespace.org/test/
6
7 GET /1/i=1:100 HTTP/1.1
8   Date: 2002-10-10T00:00:00+05:00
9   UaAuditEntryId: <myId>
10  Accept: application/opcua+json
11
12 HTTP/1.1 200 OK
13  UaServiceResult: 0
14  Content-Type: application/opcua.NodeRepresentation+json
15  Content-Length: nnnn
16  Date: 2002-10-10T00:00:00+05:00
17
18  <...>
```

Listing A.2 – Example *Browse* request and response (simplified).

HTTP Method	URI Template
GET	$<URLPREFIX>/\{NodeId\}\{?continuationPoint,releaseContinuationPoint\}$

Table A.13 – URI template for the *Next service.

OPC UA Arguments	Query Arguments	Default Value
requestHeader	see Section 4.4.2	
releaseContinuationPoint	releaseContinuationPoint	false
continuationPoint	continuationPoint	
...	...	...

Table A.14 – *Next service request arguments.

```
{
  "continuationPoint": {
    "continuationPointId": "345dfgr",
    "href": "<host>/1/i=1:100?continuationPoint=345dfgr"
  },
  ...
}
```

Listing A.3 – Representation of a *ContinuationPoint* in JSON

```
GET /1/i=1:100?continuationPoint=345dfgr  HTTP/1.1
  Date: 2002-10-10T00:00:00+05:00
  UaAuditEntryId: <myId>
  Accept: application/opcua+json

HTTP/1.1 200 OK
  UaServiceResult: 0
  Content-Type: application/opcua.BrowseNextRepresentation+json
  Content-Length: nnnn
  Date: 2002-10-10T00:00:00+05:00

  <...>
```

Listing A.4 – Example *Next request and response (simplified).

information into the *ContinuationPoint*. If the version changes the *Bad_ContinuationPointInvalid* result code could be returned.

Table A.14 details the mapping of *Next service response parameters to HTTP headers and to the HTTP body. In comparison to previously introduced services, additional parameters depending on the underlying *Next* service (e.g., *BrowseNext*, *QueryNext*, ...) can be transferred to the server. This is, for example, useful if the server does not want to keep any state (e.g., the client-specific *ResultMask* of the **Browse** service). Of course, it would also be possible to encode also all the additional necessary information in the *ContinuationToken* itself.

Listing A.3 shows an example representation of the *ContinuationPoint* in JSON. A *ContinuationPoint* is always embedded in another resource representation, for example, in the NodeRepresentation of Section 4.4.5. The JSON object contains the **continuationPointId**, which can be used in combination with the URI template in Table A.13 and the **href**, which can be directly inserted into a web browser.

In Listing A.4 an example request and response is depicted. Line 1 contains the HTTP verb (in this case "GET"), the *UrisVersion* (in this case "1"), the *NodeId* (in this case a numeric *NodeId* with the value "100" and *NamespaceIndex* "1"), and the continuationPoint as a query parameter with the value "345dfgr". Lines 2-4 are some headers provided by the client. Line 7 provides the HTTP response starting with the status code (in this case 200). Lines 7-10 are some of the headers. Based on Line 8 it can be inferred that in the background the *BrowseNext* service was executed due to the returned resource representation type. This service returns a BrowseNextRepresentation instead of a NodeRepresentation because otherwise, the client would receive several times a duplicate of the attributes and forms section of a NodeRepresentation. Of course, in the case of JSON such information could be removed easily due to the nature of JSON. In other serialization formats, this might be harder and adds additional overhead (due to some optional handling). Because of that, a special representation is introduced which is similar to the NodeRepresentation without the attributes and forms section.

A.1.4.4 Call Service

The OPC UA *Call* service can be used to invoke OPC UA *Methods*. In Table A.15 the URI template for the *Call* service is shown (without query arguments, see also Annex A.1.1). The *Call* service also makes use of the HTTP verb POST.

HTTP Method	**URI Template**
POST	$< URLPREFIX > /CallMethodRequest.objectId$

Table A.15 – URI template for the *Call* service.

Table A.16 details the mapping of *Call* service request parameters to HTTP query arguments and to the HTTP body. OPC UA Part 4 specifies the *inputArguments* parameter as an array. However, in this JSON REST mapping, the *inputArguments* are defined as JSON object. This allows the usage of JSON schema [168]. JSON schema offers similar capabilities to JSON than XML schema offers for XML. Based on JSON schema descriptions it is not only possible to allow client-side validation of JSON documents, furthermore, also web forms can be automatically generated based on such schema descriptions. This is especially useful if a web client, like a web browser, shall be the consumer of such a representation. An example JSON schema description for the *Method*-based batch version of the *Read* service can be found in Annex A.1.5.

Table A.17 details the mapping of *Call* service response parameters to HTTP headers and to the HTTP body.

Listing A.5 shows the request HTTP body for the *Method*-based batch version of the *Read* service. The JSON schema for the given JSON document can be found in Annex A.1.5. Lines 2-6 contains the *methodId* parameter. The Lines 7 to 26 contain the *inputArgument* parameter for the *Read* service like *maxAge*, *timestampsToReturn*, and an array of *nodesToRead*. The JSON schema of Annex A.1.5 also introduces, for example, schemas for the different JSON representations of *NodeIds* based on the type (e.g., numeric).

In Listing A.6 an example request and response is depicted. Line 1 contains the HTTP verb (in this case "POST") and the *NodeId* (in this case a numeric *NodeId* with the value "100" and *NamespaceIndex* "1"). Lines 2-6 are some headers provided by the client. Line 8 contains the body parameter specified in Listing A.5. Lines 10 to 17 expose an identical request compared to Lines 1-8. The only difference is that in this case the *namespaceUri* parameter is transferred in form of a query parameter including URL encoding (see also Annex A.1.1). Line 19 provides the HTTP response starting with the status code (in this case 200). Lines 20-22 are some of the headers, while Lines 24-29 contains the payload. In contrast, Line 31 shows how a RESTful OPC UA server can react if the service execution takes some time. In this case, the server returns the HTTP status code 201 instead of 200. Based on well-known web semantics a web client knows that if this code is returned also a Location header is provided with further information about the created resource. In this case, a TaskHandle object is created (see also Section 4.6.3), which allows a client to cancel the long-running service execution or get notified if the service is finished.

OPC UA Arguments	Query Arguments	HTTP Body Type
requestHeader	see Section 4.4.2	
CallRequest		application/opcua.CallRequest+json
methodId		
inputArguments		

Table A.16 – *Call* service request arguments.

OPC UA Arguments	HTTP Header Name	HTTP Body Type
responseHeader	see Section 4.4.2	
results		application/opcua.CallMethodResult
statusCode		
inputArgumentsResults[]		
inputArgumentDiagnosticInfos[]		
outputArguments[]		
diagnosticInfos	see Section 4.4.2	

Table A.17 – *Call* service response arguments.

```
{
  "methodId": {
    "namespace": 1,
    "idType": 1,
    "id": "ReadBatch"
  },
  "inputArguments": {
    "maxAge": 0,
    "timestampsToReturn": 0,
    "nodesToRead": [
      {
        "nodeId": {
         "id": 2255
        },
        "attributeId": 3
      },
      {
        "nodeId": {
          "namespace": 1,
          "idType": 1,
          "id": "Boolean"
        },
        "attributeId": 13
      }
    ]
  }
}
```

Listing A.5 – Example payload for the *Method*-based batch *Read* service.

```
POST /i=1:100 HTTP/1.1
  Date: 2002-10-10T00:00:00+05:00
  UaAuditEntryId: <myId>
  Accept: application/opcua+json
  Content-Type: application/opcua.CallRequest+json
  UaNamespaceUris: http://myTestNamespace.org/test/

  <CallRequest>

POST /i=1:100?opcuaNamespaceUris=%5Bhttp%2F%2example.org%2Ftest%2F%5D ↘
   HTTP/1.1
  Date: 2002-10-10T00:00:00+05:00
  UaAuditEntryId: <myId>
  Accept: application/opcua+json
  Content-Type: application/opcua.CallRequest+json
  Content-Length: nnnn

  <CallRequest>

HTTP/1.1 200 OK
  Content-Type: application/opcua.CallMethodResult+json
  Content-Length: nnnn
  Date: 2002-10-10T00:00:00+05:00

  {"statusCode": 0,"inputArgumentsResults": {
      "maxAge": 0,"timestampsToReturn": 0,"nodesToRead": 0},
     "outputArguments": {
      "results": [
        {"value": {"Name": "NamespaceArray"},"statusCode": 0},
        {"value": "StringValue","statusCode": 0}]}}

HTTP/1.1 201 Created
  Date: 2002-10-10T00:00:00+05:00
  Location: /1112251125/i=2:101
```

Listing A.6 – Example *Call* request and response (simplified).

A.1.4.5 ModifyReferences Service

The ModifyReferences service is newly created within this thesis and combines most of the features from the *AddReference* and *DeleteReference* service (*NodeManagement Service Set*). One missing feature is the bidirectional deletion of *References*, which is supported by the *DeleteReferences* service. However, the other features are covered through the usage of JSON Patch [29]. In the case of OPC UA, JSON Patch is restricted to the operations "add" and "remove". Native OPC UA clients can always use the batch versions of the services introduced in Section 4.3.2 if the bidirectional delete is necessary for the use case. The concept around ModifyReferences is based on the NodeRepresentation of Section 4.4.5, which is returned for *Browse* requests as already explained in Section A.1.4.2. The ModifyReferences service is also the main reason why the NodeRepresentation is compliant to JSON Pointer [30]. Table A.18 exemplifies the URI template for the ModifyReferences service (without query arguments, see also Annex A.1.1).

Table A.19 details the mapping of ModifyReferences service request parameters to HTTP query arguments and to the HTTP body, while Table A.20 details the mapping of ModifyReferences service response parameters to HTTP headers and to the HTTP body. The content format of the JSON payload is standardized through JSON Patch. However, OPC UA defines different names for *Structure* elements of the *AddReferencesItem* (used by the *AddReferences* service) and for the *Structure* elements of *ReferenceDescription*, which is the basis for the NodeRepresentation and used by the *Browse* service. For the usage of JSON Patch it is, therefore, necessary to unify the names between both *Structures*. Table A.21 provides such a harmonization. The only parameter which is not mapped at all is the *sourceNodeId* of the *AddReferenceItem* structure. This is not necessary because the source *NodeId* is already defined through the URL of the service execution (see also Table A.18). Furthermore, the *targetServerUri* has no counterpart in the *ReferenceDescription*. The reason for that is because the *ReferenceDescription* uses the *ServerArray* to reference the *ServerURI* through an *ExpandedNodeId*. However, if the referenced server is not yet present in the *ServerArray* the client has to provide the *ServerURI* through this parameter. The server then adds the new *ServerURI* to the *ServerArray* and replaces all parts of the representation through the correct *ServerIndex*.

Listing A.7 shows an example request HTTP body for the ModifyReferences service. The *AddReference* service is mapped to the JSON patch **add** operation (Line 3 and Line 21). The **path** is based on JSON pointer and is also specified in JSON patch (Line 4, 22, and 43). Lines 2-19

HTTP Method	**URI Template**
PATCH	$<URLPREFIX>/\{sourceNodeId\}$

Table A.18 – URI template for the ModifyReferences service.

OPC UA Arguments	Query Arguments	HTTP Body Type
requestHeader	see Section 4.4.2	
ModifyReferences		application/json-patch+json

Table A.19 – ModifyReferences service request arguments.

OPC UA Arguments	HTTP Header Name	HTTP Body Type
responseHeader	see Section 4.4.2	
results[]		StatusCodeArray
diagnosticInfos	see Section 4.4.2	

Table A.20 – ModifyReferences service response arguments.

JSON Patch Key (Type)	AddReferencesItem	ReferenceDescription
	sourceNodeId	
referenceTypeId (ExpandedNodeId)	referenceTypeId	referenceTypeId
isForward (Boolean)	isForward	isForward
serverUri (String)	targetServerUri	
nodeId (ExpandedNodeId)	targetNodeId	nodeId
nodeClass (NodeClass)	targetNodeClass	nodeClass

Table A.21 – Harmonization of the *AddRefererecesItem* and *ReferenceDescription Structure*.

add a *Reference* between two already existing *Nodes*, while Lines 22-40 demonstrate how external nodes can be referenced. In this case, the server adds the missing *ServerURI* to the *ServerArray* and automatically updates the *ExpandedNodeIds* based on the correct *ServerIndex*. Lines 41-44 depict a *DeleteReference* operation. As already explained, it is not possible to remove *References* bidirectional because the scope of JSON Patch is clearly limited to the given NodeRepresentation. Furthermore, Listing A.7 also depicts that *AddReferences* and *DeleteReferences* can be combined in one service call. JSON Patch also defines sequential constraints on the execution, which allows creating a modify operation based on the combination of **remove** and **add** operations. Notice that, OPC UA does not define any order guarantees for the execution of different items in a single *AddReferences* or *DeleteReferences* service call. Because of that, a special service orchestration is necessary to ensure the correct service behavior as expected by a standardized JSON Patch implementation.

In Listing A.8 an example request and response is depicted. Line 1 contains the HTTP verb (in this case "PATCH") and the *NodeId* (in this case a numeric *NodeId* with the value "101" and *NamespaceIndex* "1"). Lines 2-6 are some headers provided by the client. Line 4 specifies as Content-

```
[
  {
    "op": "add",
    "path": "/references/(!i=10:10)i=1:100",
    "value":{
      "referenceDescription":{
        "referenceTypeId":{
          "namespace": 10,
          "id": 10
        },
        "isForward": false,
        "nodeId":{
          "namespace": 1,
          "id": 100
        },
        "nodeClass": 1
      }
    }
  },
  {
    "op": "add",
    "path": "/references/(i=10:11)1s=2:Read",
    "value":{
      "referenceDescription":{
        "referenceTypeId":{
          "namespace": 10,
          "id": 11
        },
        "isForward": true,
        "serverUri": "example.org/opcua",
        "nodeId":{
          "namespace": 2,
          "idType": 1,
          "id": "Read",
          "serverUri": 1
        },
        "nodeClass": 1
      }
    }
  },
  {
    "op": "remove",
    "path": "/references/(i=10:10)i=1:101"
  }
]
```

Listing A.7 – Example payload for the ModifyReferences service.

Type the registered MIME-Type for the standardized JSON Patch operation. Line 8 contains the body parameter specified in Listing A.7. Line 10 provides the HTTP response starting with the status code (in this case 200). Lines 11-13 are some of the headers, while Line 15 contains the payload. The payload consists of a *StatusCode*-Array. For each operation in the request, one *StatusCode* is provided in the response (the array index of the response and request is identical).

```
1 PATCH /1/i=1:101 HTTP/1.1
2   Date: 2002-10-10T00:00:00+05:00
3   UaAuditEntryId: <myId>
4   Content-Type: application/json-patch+json
5   Content-Length: nnnn
6   Accept: application/opcua+json
7
8   <...>
9
10 HTTP/1.1 200 OK
11   Content-Type: application/opcua.ModifyReferencesResponse+json
12   Content-Length: nnnn
13   Date: 2002-10-10T00:00:00+05:00
14
15   <...>
```

Listing A.8 – Example ModifyReferences request and response (simplified)

A.1.5 Example JSON Schema

This section provides an example for the *Read* service request schema (Listing A.9) and the corresponding response schema (Listing A.10) based on JSON schema [168] version 1.4.

```
{
  "type": "object",
  "properties": {
    "methodId": {
      "type": "object",
      "description": "NodeId of the Method to invoke.",
      "properties": {
        "namespace": {
          "type": "number",
          "enum": [1],
          "default": 1
        },
        "idType": {
          "type": "number",
          "enum": [1],
          "default": 1
        },
        "id": {
          "type": "string",
          "enum": ["ReadBatch"],
          "default": "ReadBatch"
        }
      },
      "required": ["namespace", "idType", "id"],
      "additionalProperties": false,
      "options": { "hidden": true }
    },
    "inputArguments": {
      "type": "object",
      "description": "The input arguments for the service call.",
      "properties": {
        "maxAge": {
          "type": "number",
          "description": "Maximum age of the value to be read in milliseconds. The [...]",
          "minimum": -3.40282e1038,
          "maximum": 3.40282e1038,
          "default": 0
        },
        "timestampsToReturn": {
          "type": "number",
          "description": "An enumeration that specifies the Timestamps to be [...]",
          "enum": [0, 1, 2, 3],
          "default": 3
        },
        "nodesToRead": {
          "type": "array",
          "description": "List of Nodes and their Attributes to read. For each [...]",
          "minItems": 1,
          "maxItems": 1000,
          "items": {
```

```
51            "type": "object",
52            "description": "Identifier for an item to read or to monitor.",
53            "properties": {
54              "nodeId": {
55                "type": "object",
56                "description": "NodeId of a Node.",
57                "oneOf": [
58                  {"$ref": "#/definitions/NodeId-i"},
59                  {"$ref": "#/definitions/NodeId-s"},
60                  {"$ref": "#/definitions/NodeId-g"},
61                  {"$ref": "#/definitions/NodeId-b"}
62                ]
63              },
64              "attributeId": {
65                "type": "number",
66                "description" : "NodeId_1; NodeClass_2; BrowseName_3; [...]",
67                "enum": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
68                18, 19, 20, 21, 22]
69              },
70              "indexRange": {
71                "type": "string",
72                "description": "This parameter is used to identify a [...]"
73              }
74            },
75            "required": ["nodeId","attributeId"],
76            "additionalProperties": false
77          }
78        }
79      },
80      "required": ["nodesToRead"],
81      "additionalProperties": false
82    }
83  },
84  "required": ["methodId","inputArguments"],
85  "definitions": {
86    "NodeId-i": {
87      "properties": {
88        "namespace": {
89          "type": "number",
90          "description": "The index for a namespace URI. The field is omitted [...]",
91          "minimum": 1,
92          "maximum": 65535,
93          "multipleOf": 1
94        },
95        "id": {
96          "type": "number",
97          "description": "UInt32 Identifier encoded as a JSON number for a Node [...]",
98          "minimum": 0,
99          "maximum": 4294967295,
100         "multipleOf": 1
101       }
102     },
103     "required": ["id"],
104     "additionalProperties": false
105   },
106   "NodeId-s": {
```

```
107       "properties": {
108         "namespace": {
109           "type": "number",
110           "description": "The index for a namespace URI. The field is omitted [...]",
111           "minimum": 1,
112           "maximum": 65535,
113           "multipleOf": 1
114         },
115         "idType": {
116           "type": "number",
117           "enum": [1],
118           "default": 1,
119           "options": { "hidden": true }
120         },
121         "id": {
122           "type": "string",
123           "description": "A String Identifier encoded as a JSON string for a [...]"
124         }
125       },
126       "required": ["idType", "id"],
127       "additionalProperties": false
128     },
129     "NodeId—g": {
130       "properties": {
131         "namespace": {
132           "type": "number",
133           "description": "The index for a namespace URI. The field is omitted [...]",
134           "minimum": 1,
135           "maximum": 65535,
136           "multipleOf": 1
137         },
138         "idType": {
139           "type": "number",
140           "enum": [2],
141           "default": 2,
142           "options": { "hidden": true }
143         },
144         "id": {
145           "type": "string",
146           "description": "A Guid Identifier encoded as described in Part [...]"
147         }
148       },
149       "required": ["idType", "id"],
150       "additionalProperties": false
151     },
152     "NodeId—b": {
153       "properties": {
154         "namespace": {
155           "type": "number",
156           "description": "The index for a namespace URI. The field is [...]",
157           "minimum": 1,
158           "maximum": 65535,
159           "multipleOf": 1
160         },
161         "idType": {
162           "type": "number",
```

```
163           "enum": [3],
164           "default": 3,
165           "options": { "hidden": true }
166         },
167         "id": {
168           "type": "string",
169           "description": "A ByteString Identifier encoded as described [...]"
170         }
171       },
172       "required": ["idType", "id"],
173       "additionalProperties": false
174     }
175   }
176 }
```

Listing A.9 – REST call request schema.

```
1 {
2   "type": "object",
3   "properties": {
4     "statusCode": {
5       "type": "number",
6       "description": "StatusCode of the Method executed in the [...]"
7     },
8     "inputArgumentResults": {
9       "type": "object",
10      "description": "List of diagnostic information [...]",
11      "properties": {
12        "maxAge": {
13          "type": "number",
14          "description": "StatusCode of maxAge"
15        },
16        "timestampsToReturn": {
17          "type": "number",
18          "description": "StatusCode of timestampsToReturn"
19        },
20        "nodesToRead": {
21          "type": "number",
22          "description": "StatusCode of nodesToRead"
23        }
24      }
25    },
26    "inputArgumentDiagnosticInfos": {
27      "type": "object",
28      "description": "List of diagnostic information corresponding [...]",
29      "properties": {
30        "maxAge": {
31          "type": ["null","object"],
32          "description": "DiagnosticInfo of maxAge",
33          "oneOf": [
34            {"$ref": "#/definitions/DiagnosticInfoNull"},
35            {"$ref": "#/definitions/DiagnosticInfo"}
36          ]
37        },
38        "timestampsToReturn": {
```

```
39          "type": ["null","object"],
40          "description": "DiagnosticInfo of timestampsToReturn",
41          "oneOf": [
42            {"$ref": "#/definitions/DiagnosticInfoNull"},
43            {"$ref": "#/definitions/DiagnosticInfo"}
44          ]
45        },
46        "nodesToRead": {
47          "type": ["null","object"],
48          "description": "DiagnosticInfo of nodesToRead",
49          "oneOf": [
50            {"$ref": "#/definitions/DiagnosticInfoNull"},
51            {"$ref": "#/definitions/DiagnosticInfo"}
52          ]
53        }
54      }
55    },
56    "outputArguments": {
57      "type": "object",
58      "description": "List of output argument values.[...]",
59      "properties": {
60        "results": {
61          "type": "array",
62          "description": "List of Attribute values. [...]",
63          "items":{
64            "type": "object",
65            "properties": {
66              "value": {
67                "type": ["string","number","object","boolean","null"],
68                "description": "The data value. If [...]"
69              },
70              "statusCode": {
71                "type": "number",
72                "description": "The StatusCode that defines [...]",
73                "minimum": 0,
74                "maximum": 4294967295,
75                "multipleOf": 1
76              },
77              "sourceTimestamp": {
78                "type": "string",
79                "description": "The source timestamp for the value."
80              },
81              "sourcePicoSeconds": {
82                "type": "number",
83                "description": "Specifies the number [...]",
84                "minimum": 0,
85                "maximum": 18446744073709551615,
86                "multipleOf": 1
87              },
88              "serverTimestamp":{
89                "type": "string",
90                "description": "The server timestamp for the value."
91              },
92              "serverPicoSeconds": {
93                "type": "number",
94                "description": "Specifies the number [...]",
```

```
95                "minimum": 0,
96                "maximum": 18446744073709551615,
97                "multipleOf": 1
98              }
99            }
100         }
101       },
102       "diagnosticInfos":{
103         "type": "array",
104         "description": "List of diagnostic [...]",
105         "items": {
106           "type": ["null","object"],
107           "oneOf": [
108             {"$ref": "#/definitions/DiagnosticInfoNull"},
109             {"$ref": "#/definitions/DiagnosticInfo"}
110           ]
111         }
112       }
113     }
114   },
115   "diagnosticInfos":{
116     "type": "object",
117     "description": "Diagnostic information for the statusCode [...]",
118     "oneOf": [
119       {"$ref": "#/definitions/DiagnosticInfoNull"},
120       {"$ref": "#/definitions/DiagnosticInfo"}
121     ]
122   }
123 },
124 "definitions": {
125   "DiagnosticInfo": {
126     "type": "object",
127     "properties": {
128       "symbolicId": {
129         "type": "number",
130         "description": "A symbolic name for the status code.",
131         "minimum": -2147483648,
132         "maximum": 2147483647,
133         "multipleOf": 1
134       },
135       "namespaceUri": {
136         "type": "number",
137         "description": "A namespace that qualifies the symbolic id.",
138         "minimum": -2147483648,
139         "maximum": 2147483647,
140         "multipleOf": 1
141       },
142       "locale": {
143         "type": "number",
144         "description": "The locale used for the localized text.",
145         "minimum": -2147483648,
146         "maximum": 2147483647,
147         "multipleOf": 1
148       },
149       "localizedText": {
150         "type": "number",
```

```
151           "description": "A human readable summary of the status code.",
152           "minimum": -2147483648,
153           "maximum": 2147483647,
154           "multipleOf": 1
155         },
156         "additionalInfo": {
157           "type": "string",
158           "description": "Detailed application specific diagnostic information."
159         },
160         "innerStatusCode": {
161           "type": "number",
162           "description": "A status code provided by an underlying system.",
163           "minimum": 0,
164           "maximum": 4294967295,
165           "multipleOf": 1
166         },
167         "innerDiagnosticInfo": {
168           "type": ["null","object"],
169           "description": "Diagnostic info associated with the inner status code.",
170           "oneOf": [
171             {"$ref": "#/definitions/DiagnosticInfoNull"},
172             {"$ref": "#/definitions/DiagnosticInfo"}
173           ]
174         }
175       },
176       "additionalProperties": false
177     },
178     "DiagnosticInfoNull": {
179       "type": "null"
180     }
181   }
182 }
```

Listing A.10 – REST call response schema.

A.2 Technical details for semantics of OPC UA information models

This chapter is structured as follows:
Section A.2.1 provides a definition and generation rules for OPC UA namespaces.
Section A.2.2 focuses on versioning concepts for OPC UA namespaces.
Section A.2.3 highlights technical details of the XML DataType mapping.

A.2.1 Namespaces

In Annex A.1.1 a concept is introduced to generate URIs for OPC UA *Nodes*, which can be easily accessed through a web browser. This concept is the basis to generate URIs, which identify semantic

concepts in OPC UA as well as domain-specific vocabularies introduced through, for example, *Companion Specifications*. In addition to the generation rules of Annex A.1.1, the generation is further refined and extended to enhance the user experience with different vocabularies. The goal is to define URIs in such a way that they can be easily used by humans as well as by web-based autonomous applications. A good example of such an approach can be found on Schema.org. Founded by Google, Microsoft, Yahoo, and Yandex, Schema.org is an extensible vocabulary, which is mainly used to standardize the structure and meaning of, for example, data on websites (see also [135]). The semantics behind Schema.org can be serialized in different formats like JSON-LD [81], RDFa [126], and Microdata [71]. Based on such vocabularies Google [156, 55] is able to automatically identify what kind of data set is exposed by a given website and what is the meaning behind the data without human intervention. For example, someone searches for a recipe of an Apple Pie and wants the results filtered based on reviews. With classic web, this is particularly hard for a search engine. While it would be easy for a human search-engine to identify the meaning of most websites by looking at the content and context, a machine cannot extract the semantics that easily. To be more concrete, the analysis if this particular website describes a book about Apple Pies, a recipe, or even a coffee with the name "Apple Pies" is an easy task for a human but nearly impossible for a machine without semantic annotations. This is exactly where the Semantic Web offers a solution in providing a syntax on how semantics can be exposed in a machine-readable way. Of course, a syntax for a language without vocabulary is not very useful. Typically, the vocabulary is provided by domain experts. An example of such a vocabulary can be found in the friend of a friend ontology [52]. However, besides the definition of terms, concepts like OWL also allow to further describe relationships between terms, like the term woman and the term man are both connected to the concept behind the term person. On top of this logic constructs, tools like reasoners can be used to simplify the usage of such data sets and identify connections which are not obvious for humans (e.g., what persons have in common, which are infected by a given virus like CoVid-19).

OPC UA Part 5 defines two standard *NamespaceIndices*. *NamespaceIndex* zero is the OPC UA *Namespace* and *NamespaceIndex* one is the *Namespace* of the local server. Furthermore, OPC UA specifies that *NamespaceIndex* one has to be identical to *ServerIndex* one. To improve the way how semantics can be retrieved, the following rules shall be applied (see also Annex A.1.1 for a possible URL generation schema):

1. Each *NamespaceURI* has to be a valid and unique URL.

2. The string-part of the *BrowseName* combined with the *NamespaceURI* must result in a valid URL.

3. If *BrowseNames* are used to generate URLs the uniqueness of the resulting URLs has to be assured.

The analysis of a certain number of *Companion Specifications* has shown that all previously mentioned rules are already fulfilled by most of the information models. However, it should be noticed, that none of these rules is mandatory for OPC UA information models. If for example the *NamespaceURI* or the string-part of the *BrowseName* contains forbidden characters an URL-encoding could be applied to generate valid URLs. In contrast, the validity of rule three depends mainly on the mapping itself because in general *BrowseNames* are often not unique within OPC UA.

If all the above rules can be applied, each *Namespace* and the corresponding semantic concepts can be exposed in a similar way to Schema.org with the following rules.

1. Each *Namespace* shall be exposed by an OPC UA server under the corresponding *NamespaceURL*.

2. If the OPC UA server is reachable by a *NamespaceURL*, the corresponding *Namespace* shall be mapped to *NamespaceIndex* one. The only exception is the server that offers the core specification of OPC UA, which is always reachable under *NamespaceIndex* zero.

3. If the URL identifying an entity in the OPC UA information model is constructed with *BrowseNames* instead of *NodeIds*, also the *NodeId*-based URL should be valid and refer to the same location.

Table A.22 – Namespace prefixes

Prefix	**Namespace**
opcua, opc	`http://opcfoundation.org/UA/`
cnc	`http://opcfoundation.org/UA/CNC/`
ta	`http://opcfoundation.org/UA/TA/`
ia	`http://opcfoundation.org/UA/IA/`
dt	`http://opcfoundation.org/UA/DT/`
rs, loc	`http://example.org/UA/`
query	`http://opcfoundation.org/UA/Examples/QueryPart4/`
xsd	`http://www.w3.org/2001/XMLSchema#`
owl	`http://www.w3.org/2002/07/owl#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs	`http://www.w3.org/2000/01/rdf-schema#`
opcuashapes	`http://opcfoundation.org/UA/shapes/`
sh	`http://www.w3.org/ns/shacl#`
spin	`http://spinrdf.org/spin#`

4. If the *BrowseName*-based URL introduces a naming conflict, the URL shall be generated with the *NodeId*.

5. Naming conflicts between *Namespaces* and generated URLs must be covered through additional concepts (e.g., if *Namespace* zero defines a *BrowseName* with the string-part "CNC", which is mapped to "`http://opcfoundation.org/UA/CNC`", the resulting URL would introduce conflicts with the VDW *Companion Specification* for machine tools).

Based on the above rules different *Namespaces* can be automatically connected without the need for additional resolving concepts (e.g., a *DiscoveryServer*). Furthermore, *NamespaceIndex* one could be accepted without a concrete *UrisVersion* (see also Section 4.2.2) based on this mapping. For example, the numeric *NodeId* of *NamespaceIndex* “1” and the value 1000, where *NamespaceIndex* one refers to the URL “`http://myLocalServer/OPCUA/`” can now be resolved based on the following URL: "`http://myLocalServer/OPCUA/i=1:1000`". Throughout this thesis, the prefixes of Table A.22 are used for *Namespaces*.

A.2.2 Namespace Versioning

Typically in OPC UA a *Namespace* is versioned through the *NamespaceMetadataType* (see OPC UA Part 5), while the *NamespaceURI* is unchanged throughout all versions. However, based on several definitions in OPC UA (e.g., *ResponseHeader*, *ExpandedNodeId*, *SessionlessInvoke*, ...) an OPC UA server is not able to import two different versions of a *Namespace* with the identical *NamespaceURI*. Finally, this leads to the following implicit rules of the OPC UA versioning concept.

1. The *Namespace* can only be extended but it is not possible to deprecate concepts.

2. It is not allowed to introduce breaking changes like altering the subtype hierarchy.

These restrictions also apply to each *Companion Specification*, which uses the same concept as the OPC Foundation for versioning. In contrast, the Semantic Web reflects the version often as part of the *NamespaceURIs* (e.g., "`http://www.w3.org/2002/07/owl`" or "`http://www.w3.org/1999/02/22-rdf-syntax-ns`"). If this concept is transferred to OPC UA, the *NamespaceURIs* could be automatically extended with the *PublicationDate* of the given *Namespace* (e.g., "`http://opcfoundation.org/UA/NamespaceVersion/2013-12-02`"), while the standard *NamespaceURIs* (e.g. "`http://opcfoundation.org/UA/`") would always point to the latest version available to the given server. Nevertheless, if a new *Namespace* has to be generated because, for example, the subtype hierarchy has to be altered a lot of additional questions arise. One question would be:

What kind of concepts are semantically equal in the two different *Namespaces* and what kind of concepts are new or have altered semantics? For a machine, it is nearly impossible to determine if the semantics of a *Node* has changed only because certain *Attributes* (e.g., *Description-Attribute* to fix a sentence) have changed. The same problem arises also if different information models introduce the same concepts with different names and in their own distinct *Namespace*. During the writing of this thesis, the OPC Foundation tries to tackle this problem with a so-called "Harmonization" working group. This working group has the task to identify common concepts and standardize them in one place, instead of standardizing the same concept in different *Companion Specifications*. However, this working group only covers *Companion Specifications* but cannot monitor user-specific information models. Furthermore, OPC UA mappings of already existing other standards might also proof rather difficult to be addressed with this concept. Finally, as the vocabulary increases further and further, the identification and harmonization of common concepts become more and more difficult. Of course, the same problem is present in ontologies of the Semantic Web. However, in the Semantic Web several concepts are introduced to address this problem (e.g., the equivalent class axiom of OWL). Based on these concepts a reasoner is able to automatically infer that two concepts of two different ontologies are equal. Even due to the fact that this is a very promising concept the authors of [66] identified some open questions around this topic. In the end it depends on the given problem what kind of concept could be used to solve the problem.

A.2.3 XML DataType mapping

Table A.23 depicts the mapping of OPC UA-*DataTypes* (first column) to an OWL compatible format (second column). OWL data types represent a subset of XML data types [125]. The mapping of most of the data types is similar to the already existing mapping of OPC UA-*DataTypes* to XML data types (see also OPC UA Part 6). However, some of the OPC UA-*DataTypes* must be mapped in a different way to allow better handling in the OWL ecosystem. The **NodeId**, **ExpandedNodeId**, and **QualifiedName** are mapped to an URI data type as already explained in Section A.2.1. This is necessary because OPC UA defines its own concept of how *Namespaces* are encoded in XML and how they shall be embedded into these *DataTypes*. The proprietary nature of this concept makes it unusable in the OWL XML representation. The **ExtensionObject**, **DataValue**, **Variant**, **DiagnosticInfo**, and **Decimal** are modeled as a string. These *DataTypes* are mapped to XML complex types in OPC UA. As already mentioned previously, the OWL XML representation only supports a subset of XML data types and some of the data types of the OPC UA XML representation are not supported in the OWL XML representation like the complex type of XML. The solution approach in such cases is to map the data type to a string data type and expose the structure of the objects through OWL properties (the two columns on the right side). Based on this solution the value can be always accessed in a single transaction context in form of one large string.

Table A.23 – OPC UA DataType to OWL DataType mapping.

OPC UA DataType	OWL DataType	OPC UA Structure	OWL Structure
Boolean	xs:boolean	-	-
Sbyte	xs:byte	-	-
Int16	xs:short	-	-
UInt16	xs:unsignedShort	-	-
Int32	xs:int	-	-
UInt32	xs:unsignedInt	-	-
Int64	xs:long	-	-
UInt64	xs:unsingedLong	-	-
Float	xs:float	-	-
Double	xs:double	-	-
String	xs:string	-	-
DateTime	xs:dateTime	-	-
Guid	xs:string	-	-
ByteString	xs:base64Binary	-	-
XmlElement	xs:string	-	-
NodeId	xs:anyUri	NamespaceIndex IdentifierType Identifier	see Section A.2.1
ExpandedNodeId	xs:anyUri	ServerIndex NamespaceIndex IdentifierType Identifier	see Section A.2.1
StatusCode	xs:unsignedInt	-	-
QualifiedName	xs:anyUri	NamespaceIndex Name	see Section A.2.1
LocalizedText	rdf:PlainLiteral	Locale Text	-
ExtensionObject	xs:string	TypeId Encoding Body	dt:typeId dt:encoding dt:body
DataValue	xs:string	Variant Status SourceTimestamp SourcePicoSeconds ServerTimestamp ServerPicoSeconds	dt:variant dt:status dt:sourceTimestamp dt:sourcePicoSeconds dt:serverTimestamp dt:serverPicoSeconds
Variant	xs:string	Type Body Dimensions	dt:type dt:body dt:dimension
DiagnosticInfo	xs:string	SymbolicId NamespaceUri Locale LocalizedText AdditionalInfo InnerStatusCode InnerDiagnosticInfo	dt:symbolicId dt:namespaceUri dt:locale dt:localizedText dt:additionalInfo dt:innerStatusCode dt:innerDiagnosticInfo
Decimal	xs:string	Scale Value	dt:scale dt:value
Enumerations	xs:int	-	-
Arrays	xs:string	-	-
Structures	xs:string	-	-
Structures (with optional fields)	xs:string	-	-
Unions	xs:string	-	-
Messages	xs:string	-	-

Nevertheless, in most cases, it might prove very useful to make the substructures of a data type directly accessible in RDF (e.g., the **Scale** value as XML unsingedShort of the **Decimal-DataType**). This also applies to, for example, the **Structure**-*DataType* (see also Section 5.3.2) and to **Arrays** (see also [6] for an OWL-based **Array** example).

A.3 Details of the OPC UA Specification

This chapter is structured as follows:
Section A.3.1 summarizes the mandatory and optional *Attributes* of the eight different *NodeClasses*.
Section A.3.2 focuses on the most important operators and operands of the *Query* service. In addition, the example OPC UA information model of OPC UA Part 4 Annex B is presented.

A.3.1 OPC UA Attributes

Table A.24 depicts an overview of all mandatory and optional OPC UA *Attributes* for the eight different *NodeClasses*.

Attributes	Variable	VariableType	Object	ObjectType	ReferenceType	DataType	Method	View
AccessLevel	M							
AccessLevelEx	O							
ArrayDimensions	O	O						
AccessRestrictions	O	O	O	O	O	O	O	O
BrowseName	M	M	M	M	M	M	M	M
ContainsNoLoops								M
DataType	M	M						
DataTypeDefinition						O		
Description	O	O	O	O	O	O	O	O
DisplayName	M	M	M	M	M	M	M	M
EventNotifier			M					M
Executable							M	
Historizing	M							
InverseName					O			
IsAbstract		M		M	M	M		
MinimumSamplingInterval	O							
NodeClass	M	M	M	M	M	M	M	M
NodeId	M	M	M	M	M	M	M	M
RolePermissions	O	O	O	O	O	O	O	O
Symmetric					M			
UserAccessLevel	M							
UserExecutable							M	
UserRolePermissions	O	O	O	O	O	O	O	O
UserWriteMask	O	O	O	O	O	O	O	O
Value	M	O						
ValueRank	M	M						
WriteMask	O	O	O	O	O	O	O	O

Table A.24 – Overview of *Attributes* (M = Mandatory, O = Optional) [77].

A.3.2 OPC UA Query

This section details some of the aspects of the *Query* service of OPC UA Part 4. Section A.3.2.1 starts with the introduction of selected operators and operands, which are necessary to understand this thesis. The following Section A.3.2.2 explains the OPC UA specific conversion and precedence rules. Finally, Section A.3.2.3 depicts the example information model of OPC UA Part 4 Annex B, which is used throughout this thesis to exemplify the *Query* service.

A.3.2.1 Query Service Operators and Operands

The *RelatedTo* filter operator contains up to six operands and is used to model the relations between different *Nodes* (see Table A.25). The first column represents the operand number, while the second column differentiates between the possible inputs for the given operand. For example, Operand[0] accepts a *Node* as a value but also another *RelatedTo* operator is allowed. In the end, this allows the chaining of *RelatedTo* operators to express more complex graph patterns. Finally, the third column gives a brief explanation of the operand and how the corresponding value is used in detail.

Operand	value	Definition
0	Node	The *Instances* of the given *Type* will be used as source *Nodes* of the *RelatedTo* operator. Only *VariableTypes* or *ObjectTypes* are allowed.
	RelatedTo	The filtered *Instances* of the other *RelatedTo* operator will be used as source *Nodes* of the *RelatedTo* operator.
1	Node	The *Instances* of the given *Type* will be used as target *Nodes* of the *RelatedTo* operator. Only *VariableTypes* or *ObjectTypes* are allowed.
	RelatedTo	The filtered *Instances* of the other *RelatedTo* operator will be used as target *Nodes* of the *RelatedTo* operator.
2	Node	The *Reference* which shall connect Operand[0] with Operand[1]. Only *ReferenceTypes* are allowed.
3	0	An undefined number of hops shall be followed in a forward direction. Each *Node* shall be of the *Type* specified by Operand[1].
	1	The *Instances* for Operand[0] shall be directly related to the *Instances* of Operand[1].
	>1	The exact number of hops defined by Operand[3] shall be followed to reach the *Instances* of Operand[1] from the *Instances* of Operand[0]. The *Type* of the intermediate *Node(s)* is undefined.
4	True	The *Instances* of Operand[0] and Operand[1] should include also subtypes.
	False	The *Instances* of Operand[0] and Operand[1] should not include subtypes.
5	True	Operand[0] and Operand[1] can also be connected through a subtype of Operand[2].
	False	Operand[0] and Operand[1] can only be connected through a *Reference* of the Type defined by Operand[2] (no subtypes are allowed).

Table A.25 – OPC UA Part 4 - *RelatedTo* operator definition (simplified) [77].

Table A.26 highlights the most complex *FilterOperand* of the *Query* service, which is called *AttributeOperand*. The first column defines the element names, while the second column specifies the *DataType*. Finally, the third column provides an explanation for each of the elements.

Name	Type	Comment
nodeId	NodeId	The *NodeId* of a *Node* from the type system.
alias	String	A symbolic name which can be reused in other locations in the filter structure (optional).
browsePath	RelativePath	The browse path which should be followed. Defined in-line.
elements[]	RelativePathElement	Array of elements to define the browse path. Each entry represents exactly one hop in the browse path.
referenceTypeId	NodeId	The *NodeId* of the *ReferenceType* to follow.
isInverse	Boolean	If the value is TRUE the inverse *Reference* shall be followed.
includeSubtypes	Boolean	If the value is TRUE also subtypes of the given *ReferenceType* should be followed.
targetName	QualifiedName	The *BrowseName* of the *Node* which shall be the target. It is also allowed to include a *Type NodeId* instead of a *BrowseName*. In this case the target *Node* shall be of the given *Type*. Finally, if this parameter is empty all *Nodes* are connected through the referenceTypeId are valid targets.
attributeId	IntegerId	The *Id* of the *Attribute*.
indexRange	NumericRange	Used to access arrays.

Table A.26 – OPC UA Part 4 - *AttributeOperand* (simplified) [77].

A.3.2.2 Query Service conversion rules

The OPC UA *Query* service does not only define different operands and operators, instead, also data conversion rules have to be defined if different *DataTypes* are used within operators. Table A.27 presents these OPC UA specific conversion rules. Each row shows how a given *DataType* can

be converted to another *DataType* (columns). Furthermore, the "I" stands for an implicit conversion while the "E" marks explicit conversions. If a conversion is not possible the corresponding cell is marked with an "X" (see also Table A.27). It should be noted that some of the implicit conversion rules are quite exclusive for OPC UA like the implicit conversion of a String to a Double. This is also the reason for Table A.28, which provides an order for all implicit conversions. For example, if an operand of *DataType* String shall be compared with an operand of *DataType* Double, the String operand would be implicitly converted to Double (if possible) because the Double *DataType* has higher precedence (see Table A.28). Another interesting fact is, that the StatusCode *DataType* cannot be converted to String but it is possible to convert the StatusCode *DataType* to, for example, the Int64 *DataType*, which can be converted to *String*.

Target Type (To) / Source Type (From)	Boolean	Byte	ByteString	DateTime	Double	ExpandedNodeId	Float	Guid	Int16	Int32	Int64	NodeId	SByte	StatusCode	String
Boolean	-	I	X	X	I	X	I	X	I	I	I	X	I	X	E
Byte	E	-	X	X	I	X	I	X	I	I	I	X	I	X	E
ByteString	X	X	-	X	X	X	X	E	X	X	X	X	X	X	X
DateTime	X	X	X	-	X	X	X	X	X	X	X	X	X	X	E
Double	E	E	X	X	-	X	E	X	E	E	E	X	E	X	E
ExpandedNodeId	X	X	X	X	X	-	X	X	X	X	X	E	X	X	I
Float	E	E	X	X	I	X	-	X	E	E	E	X	E	X	E
Guid	X	X	E	X	X	X	X	-	X	X	X	X	X	X	E
Int16	E	E	X	X	I	X	I	X	-	I	I	X	E	X	E
Int32	E	E	X	X	I	X	I	X	E	-	I	X	E	E	E
Int64	E	E	X	X	I	X	I	X	E	E	-	X	E	E	E
NodeId	X	X	X	X	X	I	X	X	X	X	X	-	X	X	i
SByte	E	E	X	X	I	X	I	X	I	I	I	X	-	X	E
StatusCode	X	X	X	X	X	X	X	X	X	I	I	X	X	-	X
String	I	I	X	E	I	E	I	I	I	I	I	E	I	X	-

Table A.27 – OPC UA Part 4 - Conversion Rules (not complete) [77].

Rank	DataType	Rank	DataType	Rank	DataType
1	Double	7	StatusCode	13	Guid
2	Float	8	Int16	14	String
3	Int64	9	UInt16	15	ExpandedNodeId
4	UInt64	10	SByte	16	NodeId
5	Int32	11	Byte	17	LocalizedText
6	UInt32	12	Boolean	18	QualifiedName

Table A.28 – OPC UA Part 4 - Data Precedence Rules [77].

A.3.2.3 OPC UA Query example information model

OPC UA Part 4 Annex B defines an example information model. Several different Types are introduced (see Figure A.3): The PersonType, including the *Properties* Lastname, FirstName, StreetAddress, City, and ZipCode; The AnimalType, including the *Property* Name and subtypes like CatType, DogType, and PigType; The CatType, including the properties NickName and CatBreed. The DogType, including the *Properties* NickName, DogBreed, and License. The PigType, including the *Property* PigBreed. The ScheduleType, including the *Property* Period and the subtype FeedingScheduleType. The FeedingScheduleType, including the *Properties* Food and Amount. In addition, also several *ReferenceTypes* are introduced: The HasChild-*ReferenceType* to connect a parent to its child; The HasSchedule-*ReferenceType* to connect an animal to its schedule; The HasAnimal-*ReferenceType* to connect a person to its animal including the two subtype-*ReferenceTypes* HasFarmAnimal and HasPet to further refine the connection type. Finally, to structure the *Instances* the AreaType is introduced. Moreover, OPC UA defines several *Instances* for the previously introduced *Types*. Figure A.4 depicts the full information model, including five *Instances* of the PersonType, four *Instances* of the AnimalType, two *Instances* of the AreaType, and two *Instances* of the ScheduleType. Finally, also a *View* is introduced (blue box in Figure A.4).

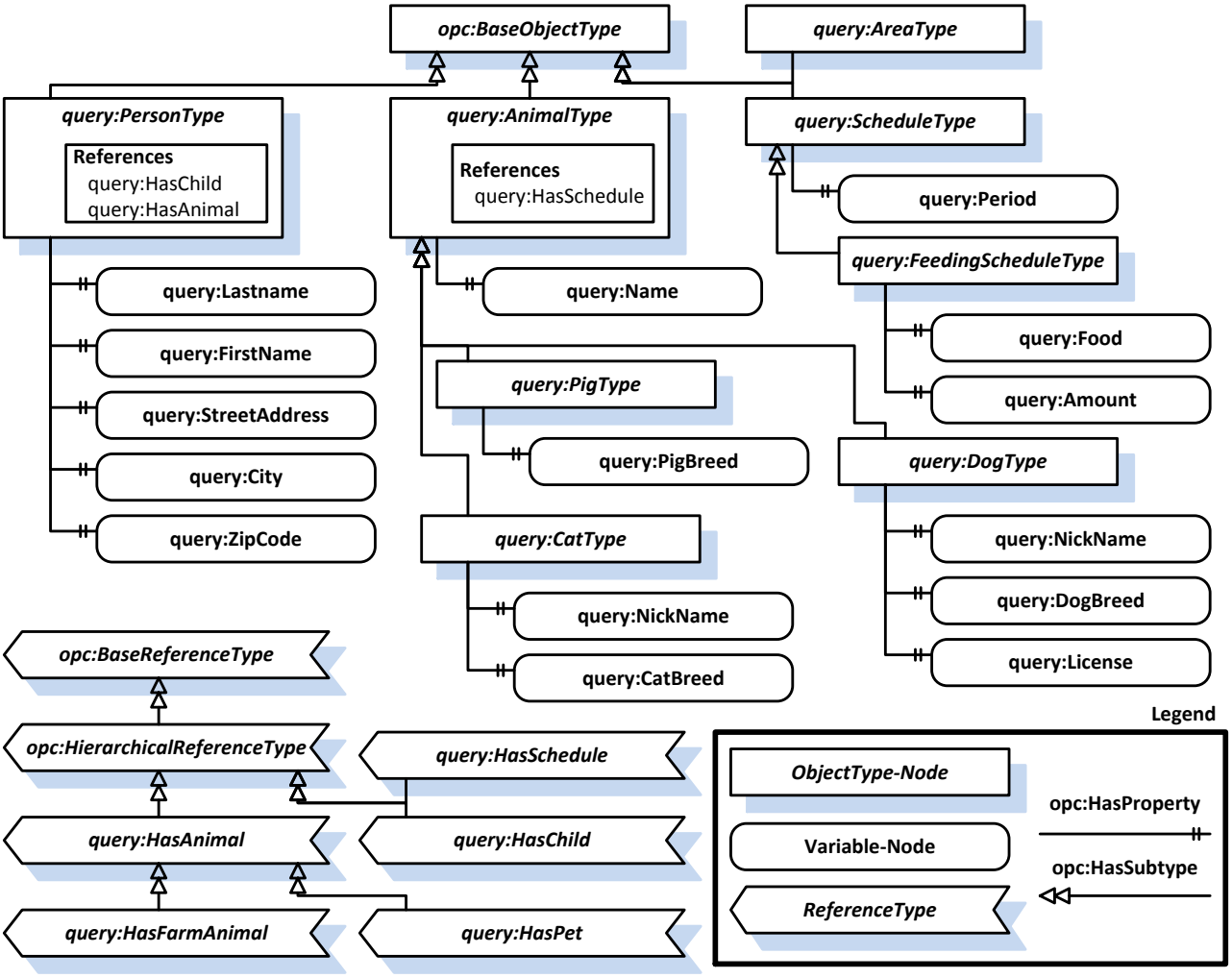


Figure A.3 – OPC UA Part 4 Annex B - Example *Type-Nodes* [77].

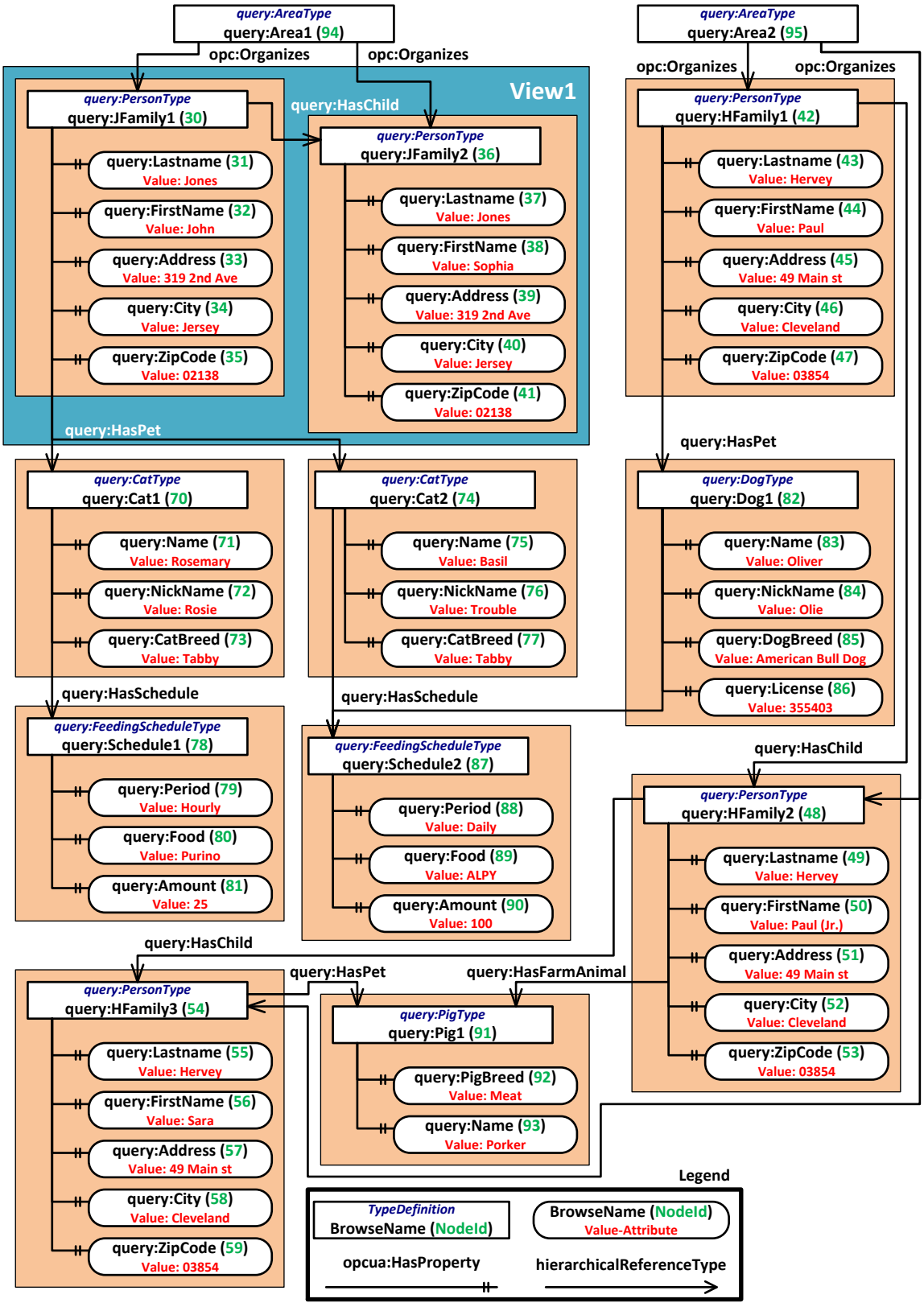


Figure A.4 – OPC UA Part 4 Annex B - Example *Instance-Nodes* [77].

PUBLICATIONS

B

The ideas and results presented in this thesis have partly also been published as:

[136] Rainer Schiekofer, Stephan Grimm, Maja Milicic Brandt, and Michael Weyrich. "A formal mapping between OPC UA and the Semantic Web." In: *IEEE 17th International Conference on Industrial Informatics (INDIN)*. 2019.

[137] Rainer Schiekofer, Andreas Scholz, and Michael Weyrich. "REST based OPC UA for the IIoT." In: *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018.

[138] Rainer Schiekofer and Michael Weyrich. "Introduction of Group-Subscriptions for RESTful OPC UA clients in IIoT environments." In: *IEEE 24th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[139] Rainer Schiekofer and Michael Weyrich. "Querying OPC UA information models with SPARQL." In: *IEEE 24th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

In [136], [137], [138], and [139] I was the main author and main contributor for the presented concepts and implementations.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[1] K. S. Aggour, V. S. Kumar, P. Cuddihy, J. W. Williams, V. Gupta, L. Dial, T. Hanlon, J. Gambone, and J. Vinciquerra. "Federated Multimodal Big Data Storage & Analytics Platform for Additive Manufacturing." In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019.

[2] A. Ait-Mlouk and L. Jiang. "KBot: A Knowledge Graph Based ChatBot for Natural Language Understanding Over Linked Data." In: *IEEE Access* (2020).

[3] M. Alam and A. Napoli. "Interactive Exploration over RDF Data using Formal Concept Analysis." In: *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 2015.

[4] *Amazon Neptune - Fast, reliable graph database built for the cloud*. `https://aws.amazon.com/neptune/`. 2020.

[5] *Amazon S3*. `https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html`. 2018.

[6] A. Andrejev, D. Misev, P. Baumann, and T. Risch. "Spatio-Temporal Gridded Data Processing on the Semantic Web." In: *2015 IEEE International Conference on Data Science and Data Intensive Systems*. 2015.

[7] A. Andrejev, X. He, and T. Risch. "Scientific Data as RDF with Arrays: Tight integration of SciSPARQL Queries into MATLAB." In: *Proceedings of the 2014 International Conference on Posters & Demonstrations Track (ISWC-PD)*. 2014.

[8] G. R. Andrews. "Paradigms for Process Interaction in Distributed Programs." In: *ACM Computing Surveys* 23.1 (1991).

[9] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. "EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning." In: *International Conference on World Wide Web (WWW)*. 2011.

[10] C. Anutariya and R. Dangol. "VizLOD: Schema Extraction And Visualization Of Linked Open Data." In: *15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2018.

[11] *Apache Jena - A free and open source Java framework for building Semantic Web and Linked Data applications.* `https://jena.apache.org/`. 2020.

[12] *Apache Jena Fuseki SPARQL server*. `https://jena.apache.org/documentation/fuseki2/`. 2018.

[13] *Apple News API*. `https://developer.apple.com/`. 2018.

[14] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. "Triplify - Light-Weight Linked Data Publication from Relational Databases." In: *Proceedings of the 18th International Conference on World Wide Web*. 2009.

[15] *AutomationML OPC UA Information Model - Companion Specification Release 1.00*. Standard. 2016.

[16] H. Baars and H.-G. Kemper. "Management Support with Structured and Unstructured Data-An Integrated Business Intelligence Framework." In: *Information Systems Management* (2008).

[17] J. Bakakeu, M. Brossog, J. Zeitler, J. Franke, S. Tolksdorf, H. Klos, and J. Peschke. "Automated Reasoning and Knowledge Inference on OPC UA Information Models." In: *IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*. 2019.

[18] J. Bakakeu, J. Bauer, S. Tolksdorf, H.-H. Klos, J. Peschke, A. Fehrle, W. Eberlein, J. Bürner, M. Brossog, L. Jahn, and J. Franke. "An Artificial Intelligence Approach for Online Energy Optimization of Flexible Manufacturing Systems." In: *Energy Efficiency in Strategy of Sustainable Production IV*. Trans Tech Publications Ltd, 2018.

[19] S. Banerjee and D. Großmann. "Aggregation of Information Models — An OPC UA Based Approach to a Holistic Model of Models." In: *4th International Conference on Industrial Engineering and Applications (ICIEA)*. 2017.

[20] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. "An Execution Environment for C-SPARQL Queries." In: *Proceedings of the 13th International Conference on Extending Database Technology*. 2010.

[21] L. Bassi. "Industry 4.0: hope, hype or revolution?" In: *IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*. 2017.

[22] T. Bauernhansl, M. ten Hompel, and B. Vogel-Heuser. *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung · Technologien · Migration*. Springer Fachmedien Wiesbaden, 2014. ISBN: 978-3-658-04682-8.

[23] T. Berners-Lee, R. T. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. 2005.

[24] T. Berners-Lee, J. Hendler, and O. Lassila. "The Semantic Web." In: *Scientific American* (2001).

[25] C. Bizer and A. Seaborne. "D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs." In: *In Proceedings of the 3rd International Semantic Web Conference (ISWC)*. 2004.

[26] D. Boldt, H. Hasemann, M. Karnstedt, A. Kroeller, and C. von der Weth. "SPARQL for Networks of Embedded Systems." In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 2015.

[27] A. Bolles, M. Grawunder, and J. Jacobi. "Streaming SPARQL - Extending SPARQL to Process Data Streams." In: *The Semantic Web: Research and Applications*. 2008.

[28] S. Brandt, E. G. Kalaycı, R. Kontchakov, V. Ryzhikov, G. Xiao, and M. Zakharyaschev. "Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic." In: *AAAI Conference on Artificial Intelligence*. 2017.

[29] P. Bryan and M. Nottingham. *JavaScript Object Notation (JSON) Patch*. RFC 6902. 2013.

[30] P. Bryan, K. Zyp, and M. Nottingham. *JavaScript Object Notation (JSON) Pointer*. RFC 6901. 2013.

[31] A. Bunte, O. Niggemann, and B. Stein. "Integrating OWL Ontologies for Smart Services into AutomationML and OPC UA." In: *IEEE Emerging Technologies and Factory Automation*. 2018.

[32] A. Cachada, J. Barbosa, P. Leitão, C. A. S. Geraldes, L. Deusdado, J. Costa, C. Teixeira, J. Teixeira, A. H. J. Moreira, P. M. Moreira, and L. Romero. "Maintenance 4.0: Intelligent and Predictive Maintenance System Architecture." In: *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018.

[33] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. "Enabling Ontology-Based Access to Streaming Data Sources." In: *The Semantic Web (ISWC)*. 2010.

[34] G. Cheng, L. Liu, X. Qiang, and Y. Liu. "Industry 4.0 Development and Application of Intelligent Manufacturing." In: *IEEE International Conference on Information System and Artificial Intelligence (ISAI)*. 2016.

[35] P. Chiariotti, P. Castellini, E. Concettoni, M. Fitti, G. L. Duca, E. Minnetti, N. Paone, and C. Cristalli. "Smart measurement systems for Zero-Defect Manufacturing." In: *IEEE 16th International Conference on Industrial Informatics (INDIN)*. 2018.

[36] D. Cocker and P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. RFC 5234. 2008.

[37] D. Damljanovic, M. Agatonovic, and H. Cunningham. "FREyA: An Interactive Way of Querying Linked Data Using Natural Language." In: *Proceedings of 1st Workshop on Question Answering over Linked Data (QALD-1), Collocated with the 8th Extended Semantic Web Conference (ESWC 2011)*. Heraklion, Greece, 2011.

[38] *Dash web application*. https://www.thedash.com/. 2018.

[39] H. Derhamy, J. Rönnholm, J. Delsing, J. Eliasson, and J. van Deventer. "Protocol interoperability of OPC UA in Service Oriented Architectures." In: *IEEE Industrial Informatics*. 2017.

[40] C. Dripke, B. Schneider, M. Dragan, A. Zoitl, and A. Verl. "Concept of Distributed Interpolation for Skill-Based Manufacturing with Real-Time Communication." In: *Tagungsband des 3. Kongresses Montage Handhabung Industrieroboter*. 2018.

[41] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. 2005.

[42] L. Dürkop and J. Jasperneite. "„Plug & Produce" als Anwendungsfall von Industrie 4.0." In: *Handbuch Industrie 4.0 Band 2: Automatisierung*. Springer Berlin Heidelberg, 2017.

[43] *ECLASS - Standard for master data and semantics for digitalization.* https://www.eclass.eu/index.html. 2020.

[44] *FaCT++ reasoner*. http://owl.cs.manchester.ac.uk/tools/fact/. 2018.

[45] A. Faul, N. Jazdi, and M. Weyrich. "Approach to Interconnect Existing Industrial Automation Systems with the Industrial Internet." In: *21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016.

[46] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. 2000.

[47] R. T. Fielding. *REST APIs must be hypertext-driven*. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. 2008.

[48] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. 1999.

[49] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio, and M. Hadley. *URI Template*. RFC 6570. 2012.

[50] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. 2014.

[51] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. 2015.

[52] *FOAF Vocabulary Specification*. http://xmlns.com/foaf/spec/. 2020.

[53] A. Fuggetta, G. P. Picco, and G. Vigna. "Understanding Code Mobility." In: *IEEE Transactions on Software Engineering* 24.5 (1998).

[54] D. Garlan and M. Shaw. *An Introduction to Software Architecture*. Tech. rep. 1994.

[55] *Get your recipes on Google*. https://developers.google.com/search/docs/data-types/recipe. 2020.

[56] G. D. Giacomo, M. Lenzerini, and R. Rosati. "On Higher-Order Description Logics." In: *Description Logics*. 2009.

[57] *GitHub replaces REST with GraphQL*. https://githubengineering.com/the-github-graphql-api/. 2018.

[58] *Global Restrictions on Axioms in OWL 2 DL*. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/#Global_Restrictions_on_Axioms_in_OWL_2_DL. 2012.

[59] T. Goldschmidt and W. Mahnke. "An Internal Domain-Specific Language for Constructing OPC UA Queries and Event Filters." In: *European Conference on Modelling Foundations and Applications*. 2012.

[60] *Google Gmail*. https://developers.google.com/gmail/api/v1/reference/. 2018.

[61] M. Graube, L. Urbas, and J. Hladik. "Integrating Industrial Middleware in Linked Data Collaboration Networks." In: *IEEE Emerging Technologies and Factory Automation*. 2016.

[62] D. Großmann, M. Bregulla, S. Banerjee, D. Schulz, and R. Braun. "OPC UA Server Aggregation — The Foundation for an Internet of Portals." In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 2014.

[63] S. Gruener, J. Pfrommer, and F. Palm. "RESTful Industrial Communication with OPC UA." In: *IEEE Transactions on Industrial Informatics* 12.5 (2016).

[64] C. Gröger, H. Schwarz, and B. Mitschang. "The Deep Data Warehouse: Link-Based Integration and Enrichment of Warehouse Data and Unstructured Content." In: *IEEE 18th International Enterprise Distributed Object Computing Conference*. 2014.

[65] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. "A Comparison of RDF Query Languages." In: *The Semantic Web (ISWC)*. 2004.

[66] H. Halpin, P. J. Hayes, J. P. McCusker, D. L. McGuinness, and H. S. Thompson. "When owl:sameAs Isn't the Same: An Analysis of Identity in Linked Data." In: *The Semantic Web (ISWC)*. Springer Berlin Heidelberg, 2010.

[67] M. Hermann, T. Pentek, and B. Otto. "Design Principles for Industrie 4.0 Scenarios." In: *49th Hawaii International Conference on System Sciences (HICSS)*. 2016.

[68] *HermiT OWL Reasoner*. http://www.hermit-reasoner.com/. 2018.

[69] C. Hildebrandt, A. Scholz, A. Fay, T. Schröder, T. Hadlich, C. Diedrich, M. Dubovy, C. Eck, and R. Wiegand. "Semantic Modeling for Collaboration and Cooperation of Systems in the production domain." In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2017.

[70] X. L. Hoang, A. Fay, P. Marks, and M. Weyrich. "Systematization Approach for the Adaptation of Manufacturing Machines." In: *21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016.

[71] *HTML Microdata*. `https://www.w3.org/TR/microdata/`. 2018.

[72] *HTTP status code in IIS 7 and later*. `https://support.microsoft.com/en-us/help/943891/the-http-status-code-in-iis-7-0-iis-7-5-and-iis-8-0`. 2020.

[73] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. "Answering Natural Language Questions by Subgraph Matching over Knowledge Graphs." In: *IEEE Transactions on Knowledge and Data Engineering* (2018).

[74] T. Hubauer, S. Lamparter, P. Haase, and D. Herzig. "Use Cases of the Industrial Knowledge Graph at Siemens." In: *International Semantic Web Conference (ISWC)*. 2018.

[75] *IANA Media Types*. `https://www.iana.org/assignments/media-types/media-types.xhtml`. 2020.

[76] C. P. Iatrou, L. Ketzel, M. Graube, M. Häfner, and L. Urbas. "Design classification of aggregating systems in intelligent information system architectures." In: *25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2020.

[77] *IEC 62541: OPC Unified Architecture*. Standard. 2015.

[78] *ISA-95 Common Object Model - Companion Specification Release 1.00*. Standard. 2013.

[79] T. Javied, J. Bakakeu, D. Gessinger, and J. Franke. "Strategic energy management in industry 4.0 environment." In: *2018 Annual IEEE International Systems Conference (SysCon)*. 2018.

[80] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. 2006.

[81] *JSON-LD 1.1 - A JSON-based Serialization for Linked Data*. W3C Recommendation. 2020.

[82] B. Katti, C. Plociennik, M. Schweitzer, and M. Ruskowski. "SA-OPC-UA: Introducing Semantics to OPC-UA Application Methods." In: *IEEE International Conference on Automation Science and Engineering*. 2018.

[83] H. Kazi, B. S. Chowdhry, and Z. Memon. "MedChatBot: An UMLS based Chatbot for Medical Students." In: *International Journal of Computer Applications* (2012).

[84] H.-G. Kemper, H. Baars, and H. Lasi. "An Integrated Business Intelligence Framework." In: *Business Intelligence and Performance Management* (2013).

[85] *Kepware: KEPServerEX*. https://www.kepware.com/en-us/products/kepserverex/. 2018.

[86] E. Kharlamov, E. Jiménez-Ruiz, D. Zheleznyakov, D. Bilidas, M. Giese, P. Haase, I. Horrocks, H. Kllapi, M. Koubarakis, Ö. Özçep, M. Rodríguez-Muro, R. Rosati, M. Schmidt, R. Schlatte, A. Soylu, and A. Waaler. "Optique: Towards OBDA Systems for Industry." In: *The Semantic Web: ESWC 2013 Satellite Events*. 2013.

[87] E. Kharlamov, N. Solomakhina, Ö. L. Özçep, D. Zheleznyakov, T. Hubauer, S. Lamparter, M. Roshchin, A. Soylu, and S. Watson. "How Semantic Technologies Can Enhance Data Access at Siemens Energy." In: *The Semantic Web (ISWC)*. 2014.

[88] O. Kilic and A. Dogac. "Achieving Clinical Statement Interoperability Using R-MIM and Archetype-Based Semantic Transformations." In: *IEEE Transactions on Information Technology in Biomedicine* (2009).

[89] N. Kitcharoen, S. Kamolsantisuk, R. Angsomboon, and T. Achalakul. "RapidMiner Framework for Manufacturing Data Analysis on the Cloud." In: *The 2013 10th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2013.

[90] S. Komazec, D. Cerri, and D. Fensel. "Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams." In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 2012.

[91] A. Kuss, T. Dietz, K. Ksensow, and A. Verl. "Manufacturing task description for robotic welding and automatic feature recognition on product CAD models." In: *Procedia CIRP* (2017).

[92] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. "A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data." In: *The Semantic Web (ISWC)*. 2011.

[93] D. Le-Phuoc, J. X. Parreira, and M. Hauswirth. "Linked Stream Data Processing." In: *Reasoning Web: Semantic Technologies for Advanced Query Answering*. 2012.

[94] P. Leitão. "Agent-based distributed manufacturing control: A state-of-the-art survey." In: *Engineering Applications of Artificial Intelligence* (2009).

[95] Y. Li, S. Hu, W. Tao, and B. Li. "Research on the Industrial Network Architecture of Northbound Interface." In: *Chinese Automation Congress*. 2017.

[96] Y Lu, K. C. Morris, and S. P. Frechette. "Current Standards Landscape for Smart Manufacturing Systems." In: *NIST Interagency/Internal Report (NISTIR) - 8107*. National Institute of Standards and Technology, 2016.

[97] W. Mahnke, S.-H. Leitner, and M. Damm. *OPC Unified Architecture*. Springer-Verlag Berlin Heidelberg, 2009.

[98] M. Majumder, L. Wisniewski, and C. Diedrich. "A Comparison of OPC UA & Semantic Web Languages for the purpose of Industrial Automation Applications." In: *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[99] S. Malakuti, J. Bock, M. Weser, P. Venet, P. Zimmermann, M. Wiegand, J. Grothoff, C. Wagner, and A. Bayha. "Challenges in Skill-based Engineering of Industrial Automation Systems*." In: *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018.

[100] L. F. de Medeiros, F. Priyatna, and O. Corcho. "MIRROR: Automatic R2RML Mapping Generation from Relational Databases." In: *Engineering the Web in the Big Data Era*. 2015.

[101] O. Meyer, G. Rauhoeft, D. Schel, and D. Stock. "Industrial Internet of Things: covering standardization gaps for the next generation of reconfigurable production systems." In: *IEEE 16th International Conference on Industrial Informatics (INDIN)*. 2018.

[102] F. Michel., L. Djimenou., C. Faron-Zucker., and J. Montagnat. "Translation of Relational and Non-Relational Databases into RDF with xR2RML." In: *Proceedings of the 11th International Conference on Web Information Systems and Technologies (WEBIST)*. 2015.

[103] F. Michel, J. Montagnat, and C. Faron-Zucker. "A survey of RDB to RDF translation approaches and tools." In: *Research Report I3S* (2014).

[104] M. H. Miraz, M. Ali, P. S. Excell, and R. Picking. "A Review on Internet of Things (IoT), Internet of Everything (IoE) and Internet of Nano Things (IoNT)." In: *2015 Internet Technologies and Applications (ITA)*. 2015.

[105] E. Motta, P. Mulholland, S. Peroni, M. d'Aquin, J. M. Gomez-Perez, V. Mendez, and F. Zablith. "A Novel Approach to Visualizing and Navigating Ontologies." In: *The Semantic Web (ISWC)*. 2011.

[106] M. zur Muehlen and R. Shapiro. "Business Process Analytics." In: *Handbook on Business Process Management* (2009).

[107] A. Nicolae, A. Korodi, and I. Silea. "An Overview of Industry 4.0 Development Directions in the Industrial Internet of Things Context." In: *Romanian Journal of Information Science and Technology (ROMJIST)* (2019).

[108] *NodeOPCUA*. `https://node-opcua.github.io/`. 2020.

[109] M. Nottingham. *URI Design and Ownership*. RFC 7320. 2014.

[110] *OPC UA - Field Level Communiation*. https://opcfoundation.org/wp-content/uploads/2018/11/OPCF-FLC-v2.pdf. 2018.

[111] *OPC UA - PubSub*. https://opcfoundation.org/news/press-releases/opc-foundation-announces-opc-ua-pubsub-release-important-extension-opc-ua-communication-platform/. 2018.

[112] *OPC UA - Roadmap*. https://opcfoundation.org/about/opc-technologies/opc-ua/opcua-roadmap/. 2018.

[113] *OPC UA - TSN*. https://smartindustryforum.org/opc-ua-tsn-a-small-step-for-mankind-but-a-giant-leap-for-industry/. 2018.

[114] *OPC UA Companion Specifications*. https://opcfoundation.org/markets-collaboration/. 2018.

[115] *OWL 2 New Features and Rationale - Punning*. https://www.w3.org/TR/owl2-new-features/#F12:_Punning. 2020.

[116] *OWL 2 Web Ontology Language*. W3C Recommendation. 2012.

[117] *OWL-S: Semantic Markup for Web Services*. https://www.w3.org/Submission/OWL-S/. 2018.

[118] S. K. Panda, T. Schröder, L. Wisniewski, and C. Diedrich. "Plug&Produce Integration of Components into OPC UA based data-space." In: *23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018.

[119] A. Perzylo, S. Profanter, M. Rickert, and A. Knoll. "OPC UA NodeSet Ontologies as a Pillar of Representing Semantic Digital Twins of Manufacturing Resources." In: *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[120] A. Peña and Y. K. Penya. "Distributed semantic repositories in smart grids." In: *9th IEEE International Conference on Industrial Informatics*. 2011.

[121] *PLCopen OPC UA Information Model - IEC 61131-3 - Companion Specification Release 1.00*. Standard. 2010.

[122] S. Profanter, K. Dorofeev, A. Zoitl, and A. Knoll. "OPC UA for Plug & Produce: Automatic Device Discovery using LDS-ME." In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2017.

[123] *Projexsys: HyperUa*. http://projexsys.com/hyperua/. 2018.

[124] *Protégé*. https://protege.stanford.edu/. 2018.

[125] *RDF Semantics - Interpreting Datatypes*. https://www.w3.org/TR/2004/REC-rdf-mt-20040210/#dtype_interp. 2020.

[126] *RDFa 1.1 - Rich Structured Data Markup for Web Documents*. https://www.w3.org/TR/rdfa-primer/. 2015.

[127] *Reference Architectural Model Industrie 4.0*. https://opcconnect.opcfoundation.org/2015/06/opc-ua-in-the-reference-architecture-model-rami-4-0/. 2018.

[128] *REST API Multiple-Request Chaining*. https://github.com/mikestowe/REST-API-Multiple-Request-Chaining. 2016.

[129] A. Rodríguez Díaz, A. Benito-Santos, A. Dorn, Y. Abgaz, E. Wandl-Vogt, and R. Therón. "Intuitive Ontology-Based SPARQL Queries for RDF Data Exploration." In: *IEEE Access* (2019).

[130] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, and T. Salmon Cinotti. "A Semantic Publish-Subscribe Architecture for the Internet of Things." In: *IEEE Internet of Things Journal* (2016).

[131] K. Routh and T. Pal. "A survey on technological, business and societal aspects of Internet of Things by Q3, 2017." In: *IEEE 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*. 2018.

[132] M. Sabou, I. Arsal, and A. M. P. Braşoveanu. "TourMISLOD: a Tourism Linked Data Set." In: *Semantic Web* (2013).

[133] C. Scheifele, A. Verl, and O. Riedel. "Real-time co-simulation for the virtual commissioning of production systems." In: *Procedia CIRP* (2019).

[134] S. Scheifele, J. Friedrich, A. Lechler, and A. Verl. "Flexible, self-configuring control system for a modular production system." In: *Procedia Technology* (2014).

[135] *schema.org - WebPage*. https://schema.org/WebPage. 2020.

[136] R. Schiekofer, S. Grimm, M. M. Brandt, and M. Weyrich. "A formal mapping between OPC UA and the Semantic Web." In: *IEEE 17th International Conference on Industrial Informatics (INDIN)*. 2019.

[137] R. Schiekofer, A. Scholz, and M. Weyrich. "REST based OPC UA for the IIoT." In: *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018.

[138] R. Schiekofer and M. Weyrich. "Introduction of Group-Subscriptions for RESTful OPC UA clients in IIoT environments." In: *IEEE 24th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[139] R. Schiekofer and M. Weyrich. “Querying OPC UA information models with SPARQL.” In: *IEEE 24th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[140] J. Schlechtendahl, M. Keinert, F. Kretschmer, A. Lechler, and A. Verl. “Making Existing Production Systems Industry 4.0-Ready.” In: *Production Engineering* (2014).

[141] M. Schleipen, A. Lüder, O. Sauer, H. Flatt, and J. Jasperneite. “Requirements and concept for Plug-and-Work.” In: *at - Automatisierungstechnik* (2015).

[142] J.-P. Schmidt, T. Müller, and M. Weyrich. “Einsatz einer service-orientierten Architektur zur Orchestrierung eines dezentralen Intralogistiksystems.” In: *Handbuch Industrie 4.0: Produktion, Automatisierung und Logistik*. 2020.

[143] S. Schmied, D. Großmann, S. G. Mathias, and R. Klaus Mueller. “An approach for aggregation and historicization of production entities in the graph.” In: *25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2020.

[144] I. Seilonen, T. Tuovinen, J. Elovaara, I. Tuomi, and T. Oksanen. “Aggregating OPC UA Servers for Monitoring Manufacturing Systems and Mobile Work Machines.” In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016.

[145] B. A. Shawar and E. Atwell. “ALICE Chatbot: Trials and Outputs.” In: *Computación y Sistemas* (2015).

[146] *Softing: dataFeed OPC Suite*. `https://industrial.softing.com/en/products/software-connectivity/opc-suite-servers-and-middleware.html`. 2018.

[147] *SPARQL query language*. `https://www.w3.org/TR/sparql11-overview/`. 2018.

[148] T. Spieldenner, R. Schubotz, and M. Guldner. “ECA2LD: Generating Linked Data from Entity-Component-Attribute runtimes.” In: *Global Internet of Things Summit (GIoTS)*. 2018.

[149] T. G. Stavropoulos, D. Vrakas, D. Vlachava, and N. Bassiliades. “BOnSAI: A Smart Building Ontology for Ambient Intelligence.” In: *Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics*. 2012.

[150] G. Steindl, T. Frühwirth, and W. Kastner. “Ontology-Based OPC UA Data Access via Custom Property Functions.” In: *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[151] B. A. Talkhestani, N. Jazdi, W. Schloegl, and M. Weyrich. “Consistency check to synchronize the Digital Twin of manufacturing automation based on anchor points.” In: *Procedia CIRP* (2018).

[152] B. A. Talkhestani and M. Weyrich. “Digital Twin of manufacturing systems: a case study on increasing the efficiency of reconfiguration.” In: *at - Automatisierungstechnik* (2020).

[153] D. Toti. "AQUEOS: A System for Question Answering over Semantic Data." In: *International Conference on Intelligent Networking and Collaborative Systems*. 2014.

[154] G. Tschinkel, E. Veas, B. Mutlu, and V. Sabol. "Using Semantics for Interactive Visual Analysis of Linked Open Data." In: *ISWC 2014 Posters & Demonstrations Track*. 2014.

[155] *Twitter API*. https://developer.twitter.com/en/docs/api-reference-index. 2018.

[156] *Understand how structured data works*. https://developers.google.com/search/docs/guides/intro-structured-data. 2020.

[157] *Unified Automation - C++ Based OPC UA Client and Server SDK (Bundle)*. https://www.unified-automation.com/products/server-sdk/c-ua-server-sdk.html. 2019.

[158] *Unified Automation: UAExpert*. https://www.unified-automation.com/de/produkte/entwicklerwerkzeuge/uaexpert.html. 2018.

[159] R. Valencia-García, F. García-Sánchez, D. Castellanos-Nieves, and J. T. Fernández-Breis. "OWLPath: An OWL Ontology-Guided Query Editor." In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* (2011).

[160] *VDMA - Members*. http://www.vdma.org/en/mitglieder. 2018.

[161] *VDMA - OPC UA working groups*. https://opcua.vdma.org/en/. 2018.

[162] *VDW OPC UA Information Model for CNC Systems - Companion Specification Release 1.00*. Standard. 2017.

[163] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos. "Software-Defined Industrial Internet of Things in the Context of Industry 4.0." In: *IEEE Sensors Journal* (2016).

[164] P. Wang, C. Pu, H. Wang, J. Wu, Y. Yang, L. Shao, and J. Hou. *OPC UA Message Transmission Method over CoAP*. Internet-Draft draft-wang-core-opcua-transmission-03. Work in Progress. IETF, 2018.

[165] M. Weyrich and C. Ebert. "Reference Architectures for the Internet of Things." In: *IEEE Software* (2016).

[166] M. Weyrich, J.-P. Schmidt, and C. Ebert. "Machine-to-Machine Communication." In: *IEEE Software* (2014).

[167] M. Wienke, O. Niggemann, S. Faltinski, and J. Jasperneite. "mINA-DL: A Novel Description Language Enabling Dynamic Reconfiguration in Industrial Automation." In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2011.

[168] A. Wright and H. Andrews. *JSON Schema: A Media Type for Describing JSON Documents*. Internet-Draft draft-handrews-json-schema-00. Work in Progress. IETF, 2017.

[169] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. "Ontology-Based Data Access: A Survey." In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence IJCAI-18*. 2018.

[170] L. D. Xu, W. He, and S. Li. "Internet of Things in Industries: A Survey." In: *IEEE Transactions on Industrial Informatics* (2014).

[171] M. Zapp, M. Hoffmeister, and A. Verl. "Methodology to apply semantic wikis as lean knowledge management systems on the shop floor." In: *Procedia CIRP* (2013).

[172] P. Zimmermann, E. Axmann, B. Brandenbourger, K. Dorofeev, A. Mankowski, and P. Zanini. "Skill-based Engineering and Control on Field-Device-Level with OPC UA." In: *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.