Querying OPC UA information models with SPARQL

Rainer Schiekofer[1,2] and Michael Weyrich[2]
[1]Siemens AG [2]University of Stuttgart

***Abstract*—OPC UA is one of the most important communication protocols for IIoT applications in the automation domain. One important requirement for Industry 4.0 scenarios is standardized semantics, which is introduced into OPC UA through Companion Specifications. Nevertheless, OPC UA still lacks a comfortable interface for accessing these huge graphs, especially on the edge- and cloud-layer. Of course, there is a query language specified for OPC UA. However, to the best of our knowledge, there is no publicly available implementation.**

In this paper, we will present how SPARQL can be used to query OPC UA graphs natively and also will present rules to transform OPC UA queries into SPARQL queries. Furthermore, we will highlight some issues of OPC UA Query and how they can be bypassed by native SPARQL. Finally, we were able to correctly execute all nine example queries of OPC UA Part 4 Annex B (complex examples) with both approaches.

***Index Terms*—OPC UA, Query, SPARQL, Information Model, OPC UA Query, OWL, Semantic Web, RDF, RDF(S), Mapping**

I. Introduction

In the area of factory automation, OPC Unified Architecture (OPC UA) [1] is one of the most important standards for device communication and promised 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 in 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 publish-subscribe pattern (finished) [2]; the introduction of domain-specific semantics which mainly are developed by the VDMA (ongoing) [3]; real-time capabilities for OPC UA (ongoing) [4]; the so-called Field Level Communication group (FLC) which aims to unify the field buses (ongoing) [5]; the support of dictionaries like eCl@ss (ongoing) [6].

If the above steering direction of the OPC Foundation is considered, 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 find the necessary data points on the aggregating layers like edge and cloud and bind them to the 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 we 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 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* [7].

As motivated above, some form of query capability is crucial for the further success of OPC UA as Industry 4.0 enabler. Rather than inventing a new query language from scratch or directly implementing OPC UA *Query*, we decided to investigate other query languages with already existing implementations and after that map OPC UA *Query* to the selected query language. Some of our selection criteria were: Compatibility to the data model of OPC UA; available implementations; usability and user base; available tutorials. Finally, we selected SPARQL [8] out of Gremlin [9], GraphQL [10], and Cypher [11] as the most promising query language. Mainly due to the high compatibility of the data model based on the OPC UA to OWL mapping of [12].

Furthermore, a few other researchers also tried to connect OPC UA to the Semantic Web so far, which at the end would allow expressing OPC UA in the form of RDF [13] triples. For example, the authors of [14] describe an approach to integrate OPC UA into a Linked Data environment. However, the authors seem to define a new ontology for OPC UA and also make no use of most of the built-in OPC UA concepts like the type concept. Another interesting approach with a focus on the reversed mapping direction from OWL [15] to OPC UA is discussed in [16]. The problem of [16] is that this approach does not cover all OPC UA concepts like *ReferenceTypes*. Because of that, this mapping also cannot be used to generate helpful triples for an OPC UA-specific query application, mainly due to the fact of the missing concept transformations, which are heavily used in most of the *Companion Specifications*.

As the primary contribution of this paper, we present how OPC UA information models can be queried with SPARQL directly and how the OPC UA-specific query language can be mapped to SPARQL. Furthermore, we will discuss some major issues of OPC UA *Query* which can be solved through the usage of native SPARQL queries.

II. Background

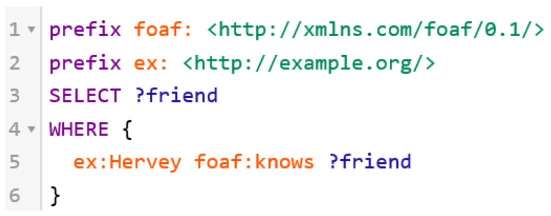
The following sections introduce the necessary background to understand the further parts of this work, starting with the basics of OPC UA (Section II-A), followed by a detailed explanation of OPC UA *Query* (Section II-B). Section II-C gives a brief overview of the SPARQL query language, while Section II-D addresses the underlying data model for the SPARQL query language, which is modeled in OWL.

A. OPC UA

Open Platform Communication Unified Architecture (OPC UA) [1] is one of the most promising industrial communication standards for Industry 4.0 scenarios. OPC UA aims to solve the two most important problems of typical IIoT scenarios, which are interoperability on the transport and semantic layer.

Interoperability on the transport layer can be fulfilled through the standardization of the transport layer based on protocols like OPC TCP and HTTP(S) in combination with different serialization formats like OPC JSON, OPC Binary, or OPC XML. Since V1.04 OPC UA also contains a cloud-ready interface based on the well-known publish-subscribe pattern, using transport protocols like MQTT and AMQP. Furthermore, the OPC Foundation started a working group to add real-time capabilities to OPC UA based on TSN. However, only standardizing the transport protocols and serialization formats is not enough. The final step for interoperability on the transport layer is the standardization of interaction patterns with the service. In OPC UA these patterns are called *Services* and used to access the graph-based data model of OPC UA. Several *Services* were defined to introspect and manipulate the graph-based information model (e.g., the *Read* service, for reading data and the *Write* service for writing data).

Interoperability on the semantic layer is achieved by the graph-based data model of OPC UA combined with so-called *Companion Specifications*. In previous years *Companion Specifications* were mostly mappings from other already existing standards to OPC UA like AutomationML, PLCOpen, ISA-95, etc. [17]–[19]. All these standards are generic and solve the problem of semantic interoperability on a rather abstract layer. Eventually, these standards are the first step towards semantic interoperability, but some pieces are still missing. For example, only standardizing the notion of a ”Thing” and a concept how ”Skills” of these ”Things” must be exposed, does not solve the issue of concrete Industry 4.0 applications like automatic skill-matching. Such applications depend on standardized semantics of concrete skills, like clamping or drilling. This final step towards semantic interoperability is exactly what is addressed by the VDMA [20]. VDMA represents more than 3200 companies within the manufacturing domain and can, therefore, be considered the largest industry association in Europe. One goal of the VDMA is to standardize domain-specific semantics for a huge part of the automation domain (e.g., robotics, machine vision, cnc machines, powertrain etc.). Their semantics will be standardized within OPC UA *Companion Specifications* [21], leading to a whole new level of semantic interoperability in the automation domain.



```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix ex: <http://example.org/>
SELECT ?friend
WHERE {
  ex:Hervey foaf:knows ?friend
}
```

Fig. 1. Example SPARQL Query.

Name	Name
Request	**Response**
view	queryDataSets[]
nodeTypes[]	nodeId
typeDefinitionNode	instanceTypeDefinitionNode
includeSubtypes	values[]
dataToReturn[]	continuationPoint
relativePath	parsingResults[]
attributeId	statusCode
indexRange	dataStatusCodes[]
filter	dataDiagnosticInfos[]
maxDataSetsToReturn	diagnosticInfos[]
maxReferencesToReturn	filterResult

TABLE I
Query First Service Parameters - without headers [1].

B. OPC UA Query

The OPC UA *Query ServiceSet* is probably the only OPC UA *ServiceSet* which has no publicly available implementation up to now. Even the fact that a whole annex was introduced in OPC UA Part 4, which added a lot of examples for this *Service*, has not fulfilled its goal. We will now take a closer look at the *QueryFirst Service* of OPC UA Part 4 and explain some of the parameters in greater detail (see also Table I).

The **view** parameter is used to select the *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, which are **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*. The **attributeId** and **indexRange** are applied to the target *Node* of this path, if a target *Node* exists. The **filter** parameter is probably 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* with each other to build very expressive filters. Additionally, OPC UA also defines so-

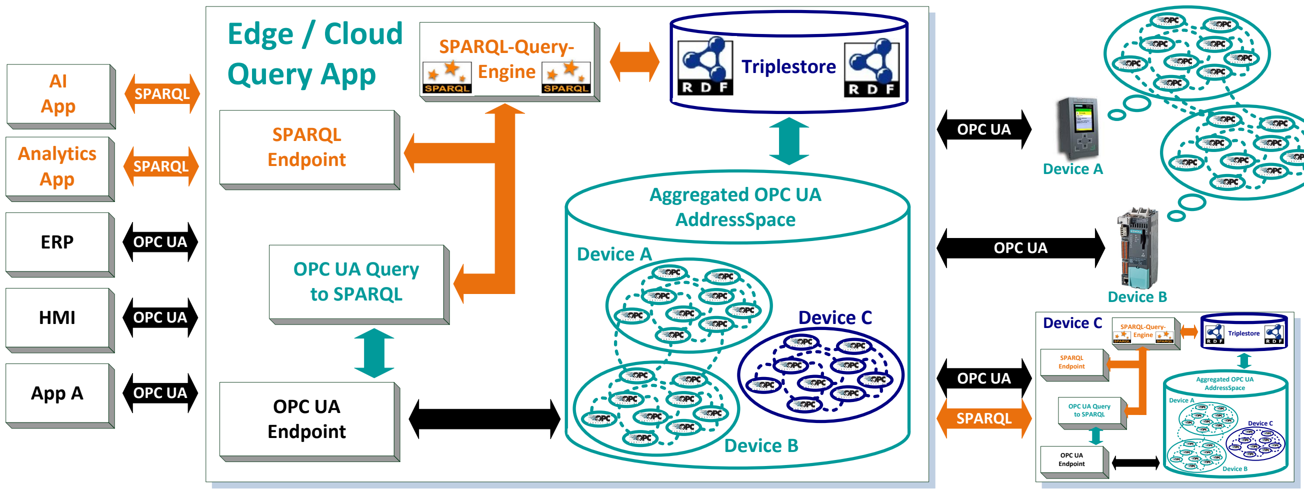


Fig. 2. Possible query architecture for the cloud-/edge-layer.

called 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 *Instance-Node* 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.

C. SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) [8] is designed to query and manipulate Resource Description Framework (RDF) [13] based data sources. SPARQL, as well as, RDF are developed by the W3C. In a nutshell, RDF allows making statements about resources with subject-predicate-object expressions. An example of such a triple is ”Hervey knows Jones”, where ”Hervey” is the subject, ”knows” is the predicate, and ”Jones” is the object. Several of these triples can be used to form an ontology to represent knowledge. An example SPARQL query is depicted in Figure 1. This query would return all objects which are connected to ”Hervey” with a ”knows” predicate. Line 1-2 are namespaces which can be assigned to subjects, objects, and predicates. Line 3 states that all values for the ”?friend” variable shall be returned. Finally, line 5 assigns all objects which are connected to ”Hervey” with a ”knows” predicate to the variable ”?friend”. In our case, the result would be ”Jones”. Of course, SPARQL offers a lot more features like aggregation, subqueries, federated queries, and plenty of expressions to further restrict the results. For example, the COALESCE statement returns the first expression that evaluates without error; The EXISTS filter operator returns true if the pattern matches and false if not; The BOUND statement which returns true if the variable is bound to a value and false if this is not the case. Finally, assignments can be formulated through the BIND statement. The full SPARQL feature set can be found under [8].

D. OWL and OPC UA information models

The Web Ontology Language (OWL) [15] is a W3C standard that provides ontological constructs for knowledge representation. OWL is based on other W3C standards such as RDF [13] and RDF(S) [22]. The semantics of OWL is grounded in Description Logics [23], which are decidable fragments of the first-order-logic.

The basic entities in OWL are the following: OWL individuals denote objects (e.g., Hervey and Jones); OWL classes denote classes of objects (e.g., Person, Animal, Schedule); OWL object properties relate objects to objects (e.g., relating a child to its parent with hasChild); OWL data properties assign data values to objects (e.g., relating a name to a person); and OWL annotation properties to record ontology meta-information, such as the author and creation date. Moreover, OWL provides constructs for building complex class expressions, such as existential restrictions (a class of Persons who have at least one child). OWL axioms, such as class subclass axioms (Cat subClassOf Animal) are used to axiomatize the ontology. Finally, individual axioms are used for statements about individuals (type assertions, property assertions, same individuals, different individuals). OWL Reasoners [24], [25] can automatically perform reasoning tasks such as checking consistency and inferring implicit relationships.

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

TABLE II
FilterOperator to SPARQL mapping.

Based on the high similarity between OWL and OPC UA it should be possible to convert OPC UA information models into OWL ontologies. The authors of [12] proposed such a mapping, which can be used for analytics and validation purpose as well. In a nutshell, the transformation rules of [12] can be reduced and simplified (for this work) in the following way: All *Type-Nodes* including *InstanceDeclarations* except *ReferenceTypes* are mapped to OWL classes; *ReferenceType-Nodes* are mapped to OWL object properties; *Attributes* are mapped to OWL data properties and annotation properties; The *BrowseName-Attribute* of most *InstanceDeclarations* is mapped to OWL object properties; *Instances* are mapped to OWL individuals; The *HasTypeDefinition-ReferenceType* is mapped to OWL type assertions; The *HasSubtype-ReferenceType* is mapped to subClassOf and subPropertyOf axioms, depending on the source concept.

III. Approach

Below, we will give an architectural overview of our query application (Section III-A). Subsequently, we explain how OPC UA information models can be queried with SPARQL directly (Section III-B), followed by Section III-C which provides the transformation rules from OPC UA *Query* to SPARQL.

A. Architecture

In this Section, we will discuss an architecture to query OPC UA information models. 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 our study we identified two main requirements to offer a simple and powerful query interface: (1) Standardized semantics, which is introduced through *Companion Specifications* in OPC UA (see also Section II-A). 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 2 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 depicted, which want to query the information model. Our prototype (the middle part of Figure 2) offers two different query languages for clients: SPARQL and OPC UA *Query*. Internal we use only the SPARQL query language and therefore have to translate OPC UA queries 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 of [12] (see also II-D). We categorize OPC UA information models 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 2 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.

B. SPARQL and OPC UA

In this Section, we will explain some of the design decisions of our native SPARQL interface. 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 object properties or as 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 for OPC UA data models by a huge amount. For our prototype, we are using OWL ontologies including inferred knowledge based on OWL-Reasoners (see also Section II-D for a sketch of the transformation rules).

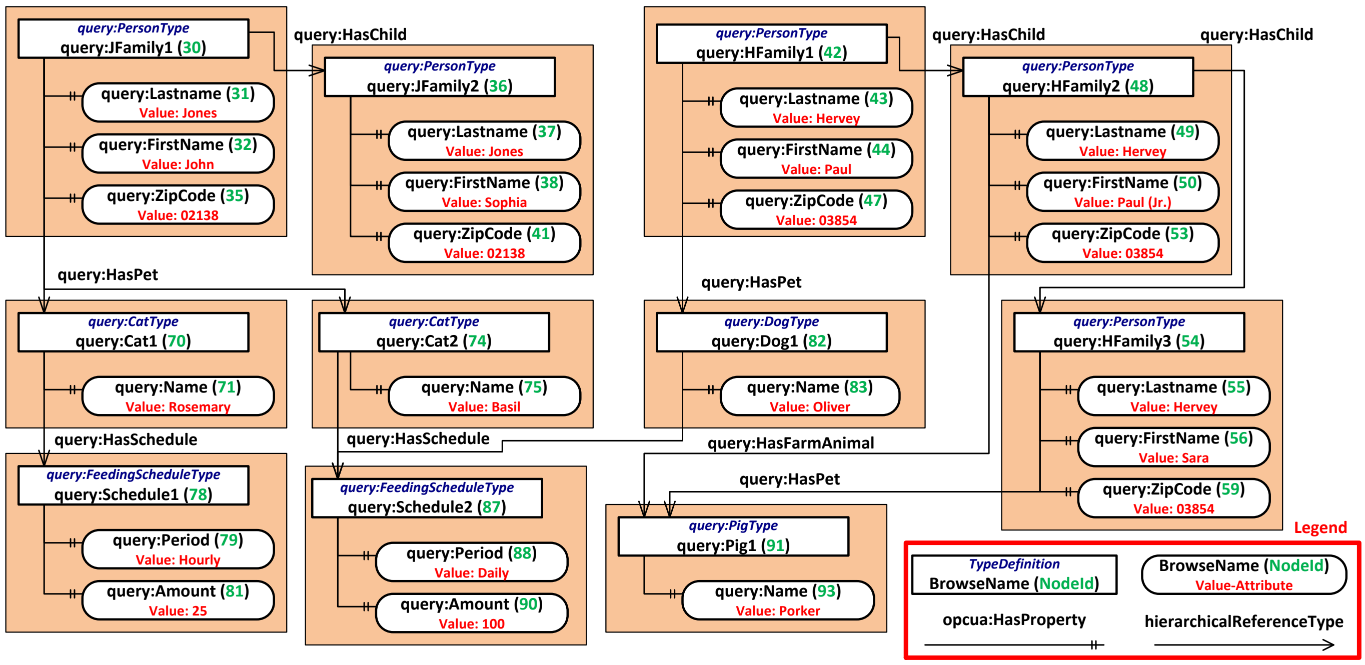


Fig. 3. Example Instance Nodes OPC UA Part 4 Figure B.4 (not complete) [1].

Views in OPC UA are introduced to offer different perspectives on a machine (e.g., for maintenance or monitoring purpose). If a certain *View* is selected only the *Nodes* which are contained in the *View* will 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*.

OPC UA introduces a concept which is called: *Programming against the TypeDefinitionNode*. In a nutshell, this concept is used to identify *Instances* which are based on an *InstanceDeclaration*. Normally such *Instances* are identified by their *BrowseName*. However, OPC UA also allows to define multiple *Nodes* with the same *BrowseName* in the context of the same *Instance-Node*. In this case the *TranslateBrowsePathsToNodeIds-Service* of OPC UA returns the *Node* which is based on the *InstanceDeclaration* as the first entry in the list. However, SPARQL does not offer a similar concept. Because of that, an additional *BrowseNames* object property is introduced into the ontology for most *InstanceDeclarations* (see also Section II-D). This object property not only allows to support the *Programming against the TypeDefinitionNode* concept, furthermore, it replaces three SPARQL filter statements through only one statement.

C. OPC UA Query to SPARQL

In the previous Section, we gave insights 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, we will now explain in greater detail how OPC UA *Query* can be translated to SPARQL. Table II 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. This is not true for SPARQL. Because of that, additional algorithms (not further discussed in this paper) are necessary to cover all implicit OPC UA *Query* conversion rules. For a similar reason, the cast operator cannot be fully supported, because the data type model of OPC UA is extensible, while in contrast the OPC UA to OWL mapping is limited to certain XSD-Schema types, which are supported by OWL tools. The *BitwiseAnd* and *BitwiseOr* filter operators also have no direct counterpart in SPARQL. The *RelatedTo* filter operator contains up to six operands, which sometimes lead to large SPARQL representations (e.g., if operand three is "0") and because of that are only included partly in form of examples in Section IV-B. The mapping of the other missing parts (e.g., *AttributeOperand* and *RelativePath*) are only included partly in form of examples in Section IV-B also because of space-limitations.

In conclusion, besides the few restrictions on some of the operators explained above, we were able to cover most of the features 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*.

IV. Prototype

Below, we will discuss our query prototype based on two examples of OPC UA Part 4 Annex B [1]. In Section IV-A we will shortly introduce the necessary parts of the example information model, followed by two example queries to show the differences between the OPC UA *Query* based approach (Section IV-B) and the native SPARQL approach (Section

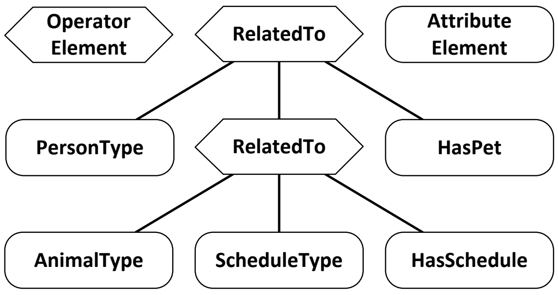


Fig. 4. Example B.2.4 - Filter [1].

IV-C). Finally, we will end with a discussion regarding the two different concepts (Section IV-D).

A. *Example Information Model*

OPC UA Part 4 Annex B [1] defines an example information model. Several different Types are introduced: The PersonType, including *Properties* like Lastname, FirstName, and ZipCode; The AnimalType, including *Properties* like Name and subtypes like CatType, DogType, and PigType; The ScheduleType, including *Properties* like Period and the subtype FeedingScheduleType. 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. The parts of the *Instance*-hierarchy of the information model, which are necessary to understand the subsequent queries are shown in Figure 3, including five *Instances* of the PersonType, four *Instances* of the AnimalType, and two *Instances* of the ScheduleType.

B. *OPC UA Query*

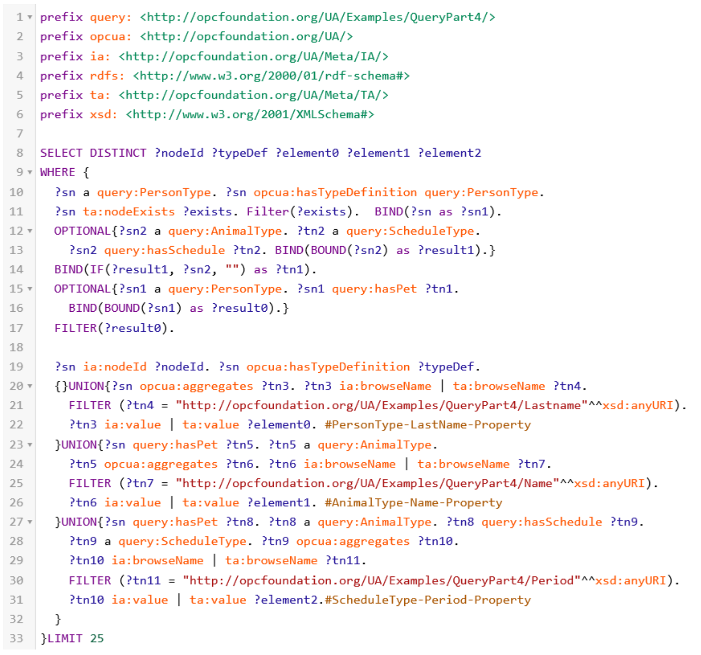
Now we will focus on Example B.2.4 from OPC UA Part 4 utilizing the transformation rules of Section III-C.

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 *ReferenceType*. In addition, the AnimalType *Instance* must be connected to a ScheduleType *Instance* with a HasSchedule *ReferenceType* (see also Figure 4).

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 III).

TABLE III
EXAMPLE B.2.4 - NODETYPEDESCRIPTION (NODETYPES[]) [1].

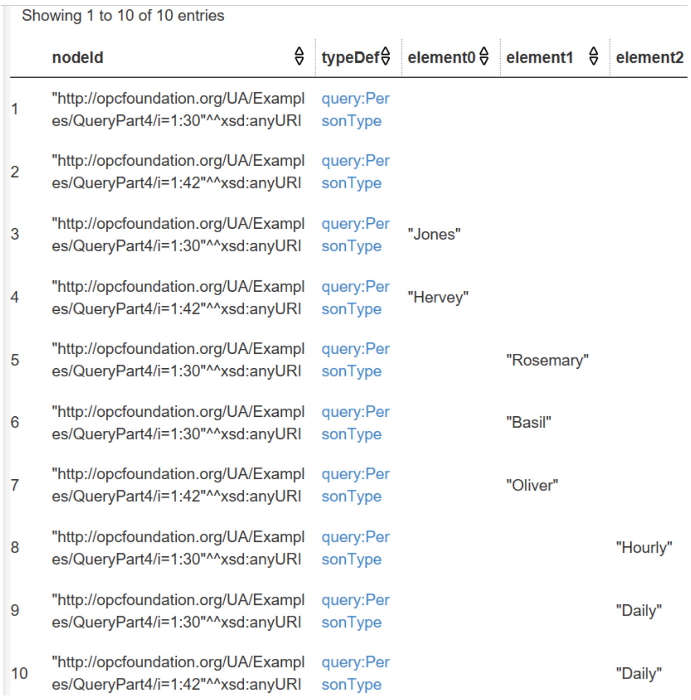
Type-DefinitionNode	**Include Subtypes**	**QueryDataDescription**	
		Relative Path	**Att.**
PersonType	FALSE	".12:Lastname"	value
		"<12:HasPet>12:AnimalType .12:Name"	value
		"<12:HasPet>12:AnimalType <12:HasSchedule>12:ScheduleType.12:Period"	value



```
 1 prefix query: <http://opcfoundation.org/UA/Examples/QueryPart4/>
 2 prefix opcua: <http://opcfoundation.org/UA/>
 3 prefix ia: <http://opcfoundation.org/UA/Meta/IA/>
 4 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 5 prefix ta: <http://opcfoundation.org/UA/Meta/TA/>
 6 prefix xsd: <http://www.w3.org/2001/XMLSchema#>
 7
 8 SELECT DISTINCT ?nodeId ?typeDef ?element0 ?element1 ?element2
 9 WHERE {
10   ?sn a query:PersonType. ?sn opcua:hasTypeDefinition query:PersonType.
11   ?sn ta:nodeExists ?exists. Filter(?exists).  BIND(?sn as ?sn1).
12   OPTIONAL{?sn2 a query:AnimalType. ?tn2 a query:ScheduleType.
13     ?sn2 query:hasSchedule ?tn2. BIND(BOUND(?sn2) as ?result1).}
14   BIND(IF(?result1, ?sn2, "") as ?tn1).
15   OPTIONAL{?sn1 a query:PersonType. ?sn1 query:hasPet ?tn1.
16     BIND(BOUND(?sn1) as ?result0).}
17   FILTER(?result0).
18
19   ?sn ia:nodeId ?nodeId. ?sn opcua:hasTypeDefinition ?typeDef.
20   {}UNION{?sn opcua:aggregates ?tn3. ?tn3 ia:browseName | ta:browseName ?tn4.
21     FILTER (?tn4 = "http://opcfoundation.org/UA/Examples/QueryPart4/Lastname"^^xsd:anyURI).
22     ?tn3 ia:value | ta:value ?element0. #PersonType-LastName-Property
23   }UNION{?sn query:hasPet ?tn5. ?tn5 a query:AnimalType.
24     ?tn5 opcua:aggregates ?tn6. ?tn6 ia:browseName | ta:browseName ?tn7.
25     FILTER (?tn7 = "http://opcfoundation.org/UA/Examples/QueryPart4/Name"^^xsd:anyURI).
26     ?tn6 ia:value | ta:value ?element1. #AnimalType-Name-Property
27   }UNION{?sn query:hasPet ?tn8. ?tn8 a query:AnimalType. ?tn8 query:hasSchedule ?tn9.
28     ?tn9 a query:ScheduleType. ?tn9 opcua:aggregates ?tn10.
29     ?tn10 ia:browseName | ta:browseName ?tn11.
30     FILTER (?tn11 = "http://opcfoundation.org/UA/Examples/QueryPart4/Period"^^xsd:anyURI).
31     ?tn10 ia:value | ta:value ?element2.#ScheduleType-Period-Property
32   }
33 }LIMIT 25
```

Fig. 5. 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 3. We will now take a closer look at the SPARQL representation of Example B.2.4 (see also Figure 5). Notice that, the COALESCE statements of Table II are omitted for readability purpose because no implicit casts are necessary. For our prototype, we used Apache Fuseki [26] from the Jena Semantic Web framework. Line 1 and 2 of Figure 5 defines OPC UA Namespaces, where Namespace "12"



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"

Fig. 6. Example B.2.4 - Results (Apache Fuseki).

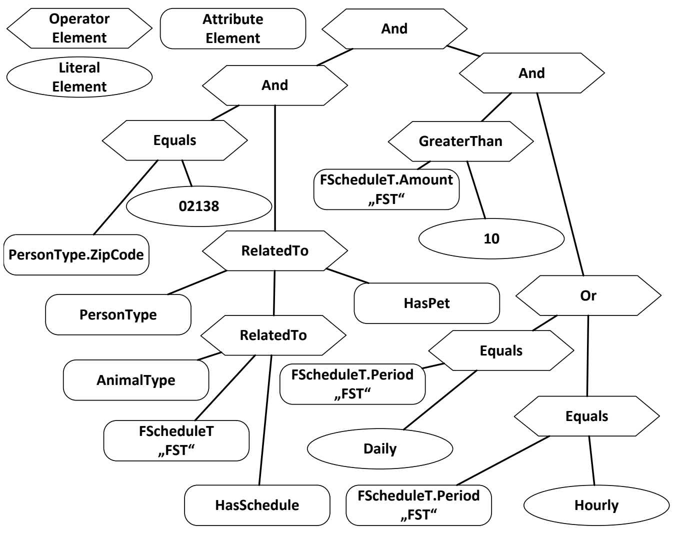


Fig. 7. Example B.2.6 - Filter [1].

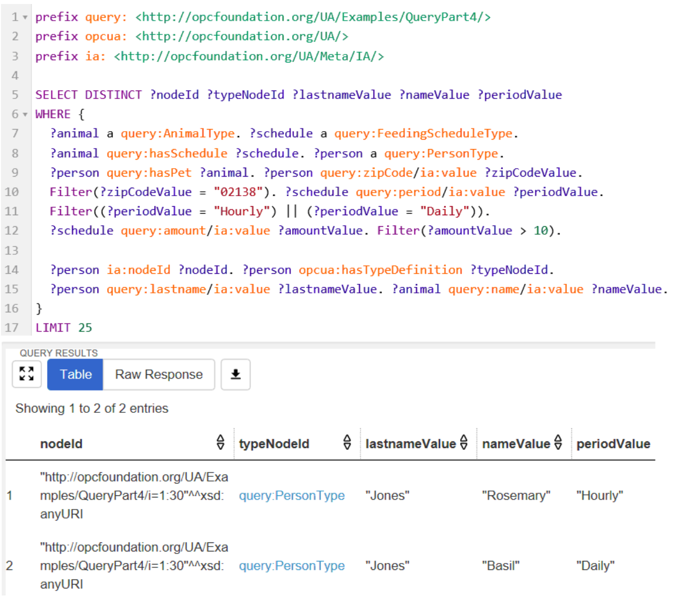
of Annex B is mapped to ”query”. Lines 3-6 define prefixes which are used in the OPC UA to OWL mapping. Lines 10-17 depicts how the filter is translated. Line 8 and the lines 19-32 are formulating the *QueryDataDescription* (**dataToReturn**). Note that, the entries of the *QueryDataDescription* (e.g., ”element0”, ”element1”, and ”element2”) are encapsulated in UNION statements according to our transformation rules. Finally, the results exactly match the results which shall be returned for the given query according to OPC UA Part 4 Annex B (see also Figure 6).

C. *Native SPARQL Query*

For the native SPARQL Query, we chose Example B.2.6 of OPC UA Part 4 Annex B. This example has the most impressive filter of all examples (see also Figure 7). 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 7).

The corresponding *NodeTypeDescription* of example B.2.6 is nearly equal to the *NodeTypeDescription* of example B.2.4, which was explained in greater detail in Section IV-B.

Figure 8 shows how this query is formulated in SPARQL natively. Line 1-3 define the used *Namespaces* similar to Section IV-B. 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 8). The results (see the lower part of Figure 8) 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


```
1  prefix query: <http://opcfoundation.org/UA/Examples/QueryPart4/>
2  prefix opcua: <http://opcfoundation.org/UA/>
3  prefix ia: <http://opcfoundation.org/UA/Meta/IA/>
4
5  SELECT DISTINCT ?nodeId ?typeNodeId ?lastnameValue ?nameValue ?periodValue
6  WHERE {
7    ?animal a query:AnimalType. ?schedule a query:FeedingScheduleType.
8    ?animal query:hasSchedule ?schedule. ?person a query:PersonType.
9    ?person query:hasPet ?animal. ?person query:zipCode/ia:value ?zipCodeValue.
10   Filter(?zipCodeValue = "02138"). ?schedule query:period/ia:value ?periodValue.
11   Filter((?periodValue = "Hourly") || (?periodValue = "Daily")).
12   ?schedule query:amount/ia:value ?amountValue. Filter(?amountValue > 10).
13
14   ?person ia:nodeId ?nodeId. ?person opcua:hasTypeDefinition ?typeNodeId.
15   ?person query:lastname/ia:value ?lastnameValue. ?animal query:name/ia:value ?nameValue.
16 }
17 LIMIT 25
```

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"

Fig. 8. Example B.2.6 - Native SPARQL Query with results (Apache Fuseki).

query would fail, while in contrast, OPC UA *Query* 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 8, which will be further discussed in the next Section.

D. *Discussion*

In conclusion, we showed that with our 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, we also highlighted how a native SPARQL query executed against an OPC UA information model can look like. We will now highlight some crucial differences between the formulation of OPC UA *Queries* and native SPARQL queries, starting with the query of Section IV-B.

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 were 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 IV-C) has a very complex filter statement. However, in OPC UA the **filter** statement and the **dataToReturn** statement is 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 7) 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 fulfils all **filter** statements and therefore 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 8 would only return Basil, because in SPARQL it is possible to interconnect the **dataToReturn** statement with the **filter** statement. Furthermore, in Figure 8 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.

In conclusion, we showed that formulating native OPC UA *Queries* is not as easy as it looks like. 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 will have a hard time with OPC UA *Query*. In addition, we identified 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. In contrast, we showed how OPC UA information models can be queried with SPARQL natively. Based on this 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 [7])), as well as, the complexity of formulating queries because, for example, the filter statement can be directly interconnected with the result statement. Finally, we were able to execute all nine example queries of OPC UA Part 4 Annex B (complex examples) with the correct results for both approaches.

V. Summary and Outlook

In this paper, we showed how SPARQL can be used to directly query OPC UA data models. Moreover, we also introduced a mapping how the OPC UA-specific query language can be mapped to SPARQL. Subsequently, we showed the validity of both approaches through the successful execution of all example queries in OPC UA Part 4 Annex B (complex examples). Finally, we closed with a discussion about some OPC UA *Query* specific issues and how they can be bypassed by the native SPARQL query approach.

Currently, we are investigating concepts to efficiently connect dynamic data from the underlying OPC UA devices to the triplestore. A first demonstrator shows that this is possible, but compared to static data there is a performance decrease. However, the first results are very promising but further performance evaluation is necessary. Another open topic is the classification of OPC UA *Attributes* in static (e.g., *NodeClass*) and dynamic (e.g., *Value*) including an efficient notification concept for changes in the static *Attributes* similar to the *ModelChangeEvent* of OPC UA Part 3.

References

[1] "Iec 62541: Opc unified architecture," Standard, 2015.
[2] "Opc ua - pubsub," https://opcfoundation.org/news/press-releases/opc-foundation-announces-opc-ua-pubsub-release-important-extension-opc-ua-communication-platform/, 2018.
[3] "Vdma - opc ua working groups," https://opcua.vdma.org/en/, 2018.
[4] "Opc ua - tsn," https://smartindustryforum.org/opc-ua-tsn-a-small-step-for-mankind-but-a-giant-leap-for-industry/, 2018.
[5] "Opc ua - field level communiation," https://opcfoundation.org/wp-content/uploads/2018/11/OPCF-FLC-v2.pdf, 2018.
[6] "Opc ua - roadmap," https://opcfoundation.org/about/opc-technologies/opc-ua/opcua-roadmap/, 2018.
[7] 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.
[8] "Sparql query language," https://www.w3.org/TR/sparql11-overview/, 2018.
[9] "Gremlin query language," http://tinkerpop.apache.org/gremlin.html, 2018.
[10] "Graphql query language," https://graphql.org/, 2018.
[11] "Cypher query language," https://neo4j.com/developer/cypher-query-language/, 2018.
[12] R. Schiekofer, S. Grimm, M. M. Brandt, and M. Weyrich, "A formal mapping between opc ua and the semantic web," in *IEEE Industrial Informatics (in press)*, 2019.
[13] "Rdf," https://www.w3.org/TR/rdf11-new/, 2018.
[14] M. Graube, L. Urbas, and J. Hladik, "Integrating industrial middleware in linked data collaboration networks," in *IEEE Emerging Technologies and Factory Automation*, 2016.
[15] "Owl 2," https://www.w3.org/TR/owl2-primer/, 2018.
[16] 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.
[17] "Automationml opc ua information model - companion specification release 1.00," Standard, 2016.
[18] "Plcopen opc ua information model - iec 61131-3 - companion specification release 1.00," Standard, 2010.
[19] "Isa-95 common object model - companion specification release 1.00," Standard, 2013.
[20] "Vdma - members," http://www.vdma.org/en/mitglieder, 2018.
[21] "Opc ua companion specifications," https://opcfoundation.org/markets-collaboration/, 2018.
[22] "Rdf schema," https://www.w3.org/TR/rdf-schema/, 2018.
[23] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, 2003.
[24] "Hermit owl reasoner," http://www.hermit-reasoner.com/, 2018.
[25] "Fact++ reasoner," http://owl.cs.manchester.ac.uk/tools/fact/, 2018.
[26] "Apache jena fuseki sparql server," https://jena.apache.org/documentation/fuseki2/, 2018.