Data interoperability across IoT domains

Károly Farkas, Zoltán Pödör, Gergely Mezei, Ferenc Somogyi

***Abstract*—**Nowadays the proliferation of IoT (Internet of Things) devices results in heterogeneous and proprietary sensor data formats which makes challenging the processing and interpretation of sensor data across IoT domains. Thus, to achieve syntactic interoperability (the ability to exchange uniformly structured data) and semantic interoperability (the ability to interpret the meaning of data unambiguously) is still an issue under research. In this paper, we introduce and discuss our purpose developed new script language called Language for Sensor Data Description (L4SDD), and the basic principles of our generic, ontology-based approach to achieve cross-domain syntactic and semantic interoperability. Moreover, we illustrate our solutions via a real-life smart parking case study.

***Keywords*—** IoT, Interoperability, Data normalization, Ontology.

I. Introduction

WE have been a witness today to the proliferation of IoT (Internet of Things) devices, solutions and use case scenarios in a myriad of domains ranging from smart home to production digitalization/industry 4.0. However, these are usually proprietary systems and solutions struggling with issues and challenges when interoperability is required. A traditional field where interoperability, or the ability of systems to exchange information, plays an important role is data network communication. Nevertheless, for using IoT data across domains and scenarios a broader, cross-domain definition of interoperability is now required [1].

A substantial step into this direction is the Virginia Modeling Analysis and Simulation Center's Levels of Conceptual Interoperability Model (LCIM) [2], which defines three categories of interoperability, such as technical, syntactic and semantic interoperability [3]:

- Technical interoperability is the fundamental ability of a network to exchange raw information of any kind.
- Syntactic interoperability is the ability to exchange structured data between two or more machines. Here, data normalization is carried out. For instance, standard data formats such as XML and JSON provide syntax that allows systems to recognize the type of data being transmitted or received.
- Semantic interoperability enables systems to interpret meaning from structured data in a contextual manner (often through the use of metadata).

To facilitate broad interoperability amidst these circumstances, the Industrial Internet Consortium (IIC) recently published the "Industrial Internet Connectivity Framework," or IICF [3]. The IICF redefines the traditional OSI model by combining the *Presentation* and *Session* layers (Layers 5 and 6) in a so-called *Framework* layer (Fig. 1) to provide all of the necessary mechanisms to facilitate how data is unambiguously structured and parsed by the endpoints.

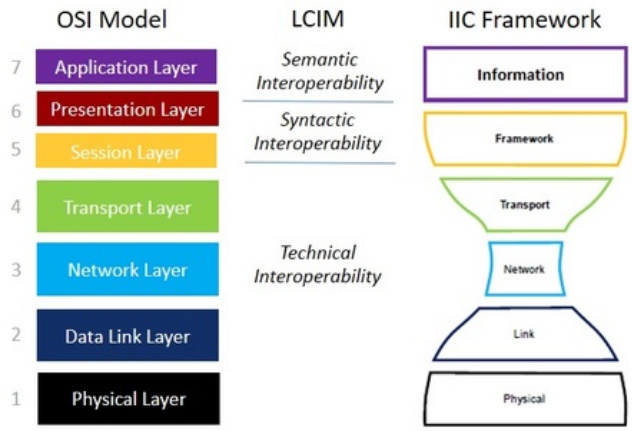

Fig. 1 Categories of interoperability

In this paper, we focus our attention on syntactic and semantic interoperability, because it can be safely assumed that technical interoperability is ensured by today's systems. We introduce and discuss our approach to achieve syntactic and semantic interoperability, which are at the heart of the framework we have been developing at the moment. This framework implements on one hand a new script language for dynamic sensor data description and data format conversion, on the other hand a generic, ontology-based semantics. The framework can be used at different parts of an IoT system, thus at the edge (running on the edge gateway) or at the IoT platform (running in the cloud) according to the needs of the implemented IoT scenario.

The rest of this paper is structured as follows. In Section II, we discuss syntactic interoperability, shortly overview some related approaches, and briefly introduce our script language.

The work presented in this paper has been carried out in the frame of project no. 2017-1.3.1-VKE-2017-00042, which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2017-1.3. funding scheme.

K. Farkas is with the Department of Networked Systems and Services, Budapest University of Technology and Economics and with NETvisor Ltd., Budapest, Hungary (corresponding author; phone: +36 1 3712719; fax: +36 1 2041664; e-mail: karoly.farkas@netvisor.hu).

Z. Pödör is with the Institute of Informatics and Economics, University of Sopron, Sopron, Hungary (e-mail: podor@inf.uni-sopron.hu).

G. Mezei and F. Somogyi are with the Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary (e-mail: [gmezei | somogyi.ferenc]@aut.bme.hu).

In Section III, we discuss semantic interoperability, give a brief overview of some related standards and introduce our ontology-based approach to achieve semantic interoperability. In Section IV, we shortly demonstrate our interoperability solutions via a smart parking case study. Finally, in Section V we conclude the paper pointing out to some future directions.

II. Syntactic Interoperability

A. Data Normalization

Nowadays it is getting more and more common that an application obtains its data from heterogeneous IoT devices. It is rather challenging to support different formats of data stemming from these devices. Thus, the data should be normalized or converted to a common format, which can be read by all the elements of the IoT system or even by different IoT systems leading to syntactic interoperability.

To achieve syntactic interoperability, first we tried to create a static description, namely a data format definition capable of describing all variants of all sensor messages. It is worth to mention that most of these sensors are capable of sending the data only in their native format, i.e., they cannot convert the data to XML or JSON. Although the static format description seemed to be a good solution at first glance, we had to realize that it fails when we want to apply it on previously known industrial case studies borrowed from real-life systems. The source of the problem was that we had many cases, where the format was dependent on a particular fragment of the data, or the data had (pre)processing instructions for itself. For example, we had to support five different encoding based on the first two bytes of the data processed. We have realized that describing complex dependencies between the fragments and using dozens of alternative paths would result in a highly verbose, overcomplicated static format definition.

Therefore, we have decided to create a dynamic scheme that is easier to customize and more expressive. This dynamic scheme, in this context, refers to our purpose developed script language called L4SDD, the Language for Sensor Data Description. L4SDD not only defines an output data format, but also specifies how the data is to be calculated from the sensor input. The converted data can later be directly stored in databases and processed by data analysis techniques.

B. Overview of Related Approaches

In order to achieve syntactic interoperability in cross-industry projects, we need a common data format understandable, readable and writable by all participants. In order to reach this goal, some of the existing approaches focus on defining a universal format applicable in all scenarios and domains. In our case, this is not enough, since even if we succeed in defining such a format, the sensors cannot convert the data to it. Therefore, we needed a solution to transform the original data given in various formats to the common form.

The Data Distribution Service (DDS) [4] is a popular data-centric publish-subscribe protocol defined by OMG. It is created to handle communication between the participants, but it would need to create adapters for IoT devices. DDS offers a standard to describe the data format only, the conversion to this format is not considered.

The OPC-Unified Architecture (OPC-UA) [5] is a popular machine-to-machine protocol for industrial automation. Its basic idea is promising, however at the current stage it is rather a pre-release standard than a working, platform independent solution. Most of the issues come from heterogeneous, incomplete implementations.

The Sensor Markup Language (SenML) [6] developed by IETF is created to describe sensor measurements and devices, which could fit into our scenario, but SenML allows to use XML, JSON, Concise Binary Object Representation (CBOR) and Efficient XML Interchange (EXI) formats only. This is not suitable in our case because of the limitation of the sensors.

The Data Format Description Language (DFDL) [7] is perhaps the nearest to provide a solution to our challenges. It is a modeling language for describing general text and binary data in a standard way. The schemas defined in DFDL allow any text or binary data to be read from its native format and written into a destination language. The standard has several implementations available and it can be integrated with several system technologies. Even by understanding its promising capabilities, we could not use DFDL. The most important reason for this is that DFDL implementations have a concrete platform to apply the conversion on and we needed more flexibility. Moreover, we wanted to optimize the conversion and to ensure its safety. By defining a new script language with limited, but efficient features and creating an environment around it (e.g., compiler, execution framework), we could achieve these goals easier.

C. The Language for Sensor Data Description

By creating L4SDD, a new, purpose developed script language, our primary aim was to devise a dynamic data description solution. Since L4SDD has been a newly created language, there was no compiler available for it, thus, we have built one based on Xtext [8].

The compiler compiles L4SDD script definitions to source code (currently to JavaScript, but it can be an arbitrary language). The source code is then registered by the framework, and it is executed every time the associated sensor sends data. More precisely the steps of data conversion definition for a new sensor are the following: (i) a new L4SDD script is defined that describes how the input data of the sensor are to be read; (ii) the script is compiled to source code and the source code is registered as a data processor algorithm; (iii) if the sensor sends data, the framework iterates through the registered data processors, selects the appropriate ones and execute them; (iv) the result of the script execution is formatted data, which are self-describing (or at least which describe their own structure). The framework stores these data in its database. The key to store all kinds of data is the self-describing data structure.

L4SDD scripts consist of several sections: (i) an *Output* definition that describes the format of the output data; (ii) a

Filter definition that is executed at Step #3, when the framework tries to find scripts applicable for the certain data; (iii) a *Mapping* definition that is the conversion itself, namely how the output data are produced from the input data; (iv) the script may also contain a *Params* definition, where additional parameters can be passed (e.g., the current location), which can affect the result of *Mapping*. These data are not sent by the sensor to the framework, but the framework should add this information as an additional input parameter for the script when it is executed.

The *Output* and *Params* sections can use the same language elements and syntax (they are static format descriptions), while the *Filter* and *Mapping* parts also share their language (they are transformation logic descriptors). The four parts together allows us to specify the interpretation of all kinds of sensor data to our universal, self-describing data format used in later steps of data processing.

III. Semantic Interoperability

A. The Meaning of the Data

These days the data coming from IoT and/or smart devices are stored and communicated in many different forms. By solving the challenge of supporting syntactic interoperability, we have reached only halfway. We also need the description of the exact meaning of the data, also called metadata or semantic data. The challenge of semantic interoperability is to ensure the ability to represent, communicate and interpret the meaning of data automatically and in a consistent manner even across domain borders.

On top of data normalization, which is assured by syntactic interoperability, semantic interoperability facilitates IoT systems to share and exchange data with unambiguous meaning and interpretation. Thus, it enables the different elements of IoT systems, such as sensors, actuators, gateways, IoT platforms, applications, to interpret meaning unambiguously from structured data in a contextual manner. Semantic interoperability provides a common understanding of the transferred data by using standard and well-defined vocabulary, data format and structure. The semantic meaning of the data can be transferred together with the raw data in a self-describing extension package, whose format and structure are independent of the used information system.

Beyond offering the opportunity of effective and automated interworking among IoT systems, semantic interoperability also assures the possibility of data coupling and data broker services across IoT domains and applications.

B. Overview of Related Approaches

Several approaches have been arisen to deal with the semantic interoperability challenge. Most of them, e.g., the SenML [6], the EPCIS standard [9] by GS1, or the Haystack project [10], use a common vocabulary and a tag-based solution to define the meaning of the data. In this case, the transferred measurement data usually have an additional, tag-based description in a special format (e.g., JSON or XML). These tags are supposed to define the meaning of the data for the system, but they do not have a structure, i.e., they do not use an ontology to describe the relationships between the tags.

Some other approaches define a communication model for data sharing which inherently deals also, to some extent, with the meaning of the data. For instance, the DDS specification [4] describes a Data Centric Publish-Subscribe (DCPS) model for distributed application communication and integration as mentioned before. It makes possible for entities to publish data to other, subscribing entities. The topics identify the meaning of the data elements.

The ontology-based approaches, e.g., the Smart Appliances REFerence (SAREF) ontology [11] by ETSI or the Base Ontology (BO) [12] by oneM2M, use a vocabulary with a well-defined structure. In these ontologies, there are classes, which are a set of individuals (objects and subjects) to describe the domain of interest. Moreover, properties represent the relationships between the specified domain and domain ranges. The ontology can also assign restrictions to each class. Together the classes and properties define the structure of the entities.

C. Our oneM2M Base Ontology Based Approach

In our framework, our aim is to use an ontology-based solution, because the structure over the vocabulary (tags, or metadata) can provide a standardized classification of the actual domain entities via the corresponding classes. Each class represents a category of objects, which can be well identified. Every class has relationships (properties), attributes and restrictions. In an ontology, the classes have a hierarchical structure, which can be as deep as needed by using subclasses and relationships between them.

To achieve interoperability across domain borders is a real challenge because devising a general ontology which fits in every domain is an extremely hard if not impossible task. Our approach is to use a generic skeleton, which can serve as a ‘glue’ among the domain-specific ontologies. Then defining a mapping between these ‘external’ ontologies and the skeleton cross-domain interoperability can be achieved.

We have selected the oneM2M Base Ontology (BO) [10] as our generic skeleton. OneM2M defines an architectural approach for IoT cross-domain interoperability, where different applications share common service functions and network infrastructure based on a horizontal, common service layer. Moreover, oneM2M defines a semantic interoperability framework, which provides semantic information about resource contents and functionalities. This framework specifies the BO.

The oneM2M BO has been developed to provide semantic cooperation between oneM2M and external systems. The external systems are described by an external ontology. The BO does not model domain-specific aspects, it is a general and minimum ontology for sensor-based IoT systems. Rather, it defines mapping rules for external ontologies based on the classes.

In the BO, classes are directly defined by the class name and

the hierarchy, or defined by the properties of the individuals in the class. There are two main property types: (i) object properties describing the relationship between two classes; (ii) data properties describing the relationship between an object (class) and a concrete data value. Fig. 2 shows the classes and properties of the BO (the nodes denote the classes and the arrows denote the object properties) [10]. We omit the detailed description of these classes and properties due to space limitations (for more information consult [10]).

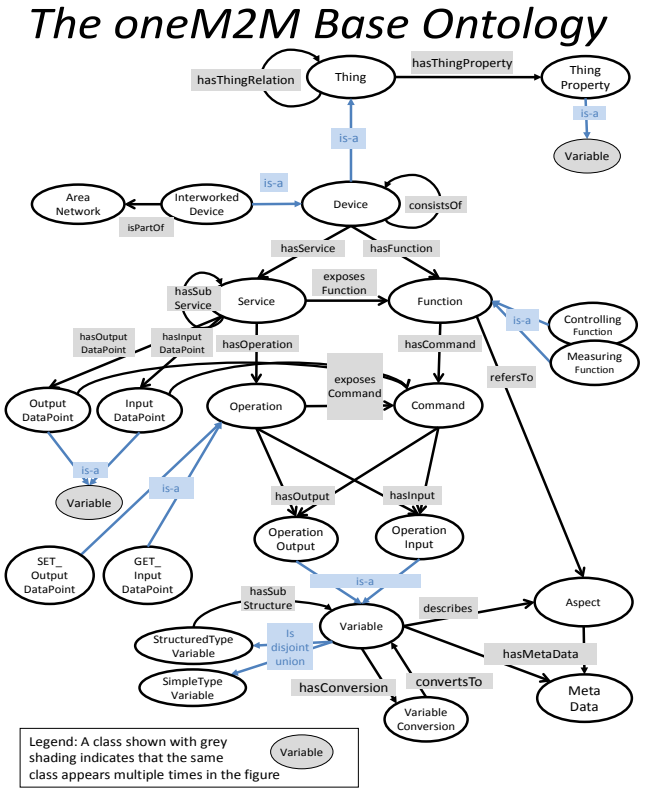

Fig. 2 OneM2M Base Ontology

The mapping of the external ontologies to the Base Ontology follows the next two rules: (i) A is a subclass of B (property subClass); and (ii) A is equivalent of B (property equivalentClass). The inheritance from upper properties and classes are implied according to the mapped hierarchy as illustrated by the following case study.

IV. Case Study

A. Smart Parking

In the following, we illustrate our approach on a simple, smart parking scenario. In this scenario, we use the Libelium Smart Parking sensor solution [13]. We apply the sensor to detect free and available parking slots in an outdoor parking area. The detection method relies on the concept of measuring the magnetic field (in a three-axis coordinate system), which is substantially modified by a car standing above the sensor and containing a lot of metal. Based on the measured values and a given threshold, the sensor indicates whether the parking slot is empty or not.

The sensor can send different kinds of data, which always consists of a Basic frame and an additive frame. The Basic frame contains the actual status of the parking slot (free, or not free), the battery state and the ID of the additive frame type, which can be the following: Info; Keep-alive; Daily update; Start frame1; Start frame2; Error. Table I summarizes the frame types and their content.

Table I Frames used by Libelium Smart Parking sensors

Frame type	Description	Fields
Basic	Base frame	• Slot status • Battery state • Additional frame type ID
Info	Status of parking slot is changed	• Slot status • Sensor temperature • Magnetic field values
Keep-alive	No slot status change, beacon to show operation	• Timestamp • Temperature • Magnetic field values
Daily update	Daily summary of the sensor's operation	Counters wrt. the operation over the last 24 hours: • How many times the sensor was used • How many times the sensor transmitted data • How many times the sensor was reset
Start frame1	The first frame after the sensor starts to work	• Temperature • Reference axis values • Battery voltage value
Start frame2	Second frame after the Start frame1	• Beginning of the night mode • Duration of the night mode • Sleep time in night mode • Duration between two Keep-alive frames sent in night mode • Sleep time in normal mode • Duration between two Keep-alive frame sent in normal mode • Threshold
Error	Used when the sensor cannot send a normal frame	• Error data • Temperature • Magnetic field values • Battery level

B. L4SDD Example

For the sake of understandability, we omit some practical details in the code. Our goal is to convert the data sent by the Libelium Smart Parking sensors to a common data format. The raw sensor data frame snippet we use in our example can be seen in Fig. 3.

The data in the frame are described in JSON, because our system appends extra information (metadata) to the raw data sent by the sensor. The raw data can be found in the highlighted *data* field. Our approach supports multiple input formats (e.g., binary, CSV). Note that the order of the data fields is not bound, our approach is flexible also in this regard. Most of the usual data sent by the sensor (e.g., ID, frequency, timestamp) can be converted without further processing. However, the raw data are sent in a hexadecimal format that needs to be processed. Moreover, it can happen that different sensors (or the same sensor at different intervals) send the data in different structures. We can solve this problem by writing multiple scripts, or by processing the data dynamically, based on the markers in it (not used in the presented example).

```
{ "ack":false, "port":4, "cls":0, "codr":"4/5","freq":"868.1",
"tstamp": "2018-01-30T00:18:17.286593Z", "snr":"8.5",
"deveui":"00-00-00-00-00-1a-fb-1d", "data":"ArEDwQAAAAsAAAA=",
... }
```

Fig. 3 Parking sensor input data example

The first step of data conversion is the format description, namely, how we describe the input data in our universal schema. This is accomplished in the *Output*. The schema description for the parking sensor example can be seen in Fig. 4. The type definitions in the description (e.g., *byte*, *int*) are specific to L4SDD, and they may be mapped differently to the target language that we generate the code for.

```
OUTPUT {
  deviceEui : byte[32];
  port : int;
  timestamp : Date;
  raw:
  {
    type : int;
    occ : int;
    measure : float;
    // ...
  };
  freq : float;
  snr : int;
  // ...
}

FILTER {
var decodedData =
ToBase64(INPUT.data);
Assert (decodedData[0] &
0xF == 2) { // ... }
}
```

```
MAPPING {
var item = JSON(MSG);

OUTPUT.deviceEui = item.deveui;
OUTPUT.port = item.port;
OUTPUT.tstamp = item.tstamp;

// Process raw data
var raw = ToBase64(item.data);
OUTPUT.raw.type =
ParseInt(Substring(…));

OUTPUT.raw.occ =
(ParseInt(Substring(…) === 1);
OUTPUT.raw.measure =
raw[2] * 256 + raw[3];
// ...

OUTPUT.freq = item.freq;
OUTPUT.snr = item.lsnr;
// ...
}
```

Fig. 4 L4SDD script example

The second step is to define the processing logic in the script. The purpose of the *Filter* section (Fig. 4) is to check if the data can be processed by the script. In the example, we check the first byte of the raw data, which identifies the type of the message sent by the sensor. Then, the *Mapping* section contains the processing logic for the script. In the example, we process the raw data by using the standard operations and functions in the L4SDD language. The rest of the data can be directly copied to the structured output without further processing.

The third step of the data conversion is the code generation for the target language(s). Later, the system will execute this generated code on the data accepted by the *Filter*. Currently, we generate JavaScript code (referred to as L4JS), but the code generation can easily be extended to other languages.

A snippet of the generated code of our example can be seen in Fig. 5. The *Filter* and *Mapping* are both mapped to a JavaScript functions (*isAccepted* and *processData*). The system calls these functions with the specific sensor data. Note, that the structure and syntax of the generated code is very similar to that of the L4SDD script, because our example is rather simple. However, in case of more sophisticated, real-life scenarios, our language also simplifies the syntax of the script. For example, functions can be mapped to more complex instructions in the target language (e.g., switch-cases, data copying), simplifying to write the processing logic in L4SDD.

```
function isAccepted(input) {
    let decodedData = ToBase64(input.data);
    if ((decodedData[0] & 0xF) != 2) {
        return false;
    }
    return true;
                                }
function processData(input, params, message) {
    let item = JSON(message);
    let raw = ToBase64(item.data);
    structuredOutput.deviceEui = item.deveui;
    structuredOutput.port = item.port;
    structuredOutput.tstamp = item.tstamp ;
    structuredOutput.raw.type = ParseInt(…);
    structuredOutput.raw.occ = ((ParseInt(…)) === 1);
    structuredOutput.raw.measure = raw[2] * 256 + raw[3];
    // ...
    structuredOutput.freq = item.freq;
    structuredOutput.snr = item.lsnr;
    // ...
    return structuredOutput;
}
```

Fig. 5 Generated JavasScript (L4JS) code snippet

Finally, Fig. 6 depicts the structured JSON output for the parking sensor example, containing the type and value information. The type information (on the left) contains the structure of the output. We can use this information later, when the converted data is to be stored.

```
{
  "deviceEui": "byte[32]",
  "port": "int",
  "tstamp": "Date",
  "raw": {
    "type": "int",
    "occ": "int",
    "measure": "float", ...
  },
  "freq":"float",
  "snr":"float", ...
}
```

```
{
  "deviceEui": "00-00-00-00-
00-1a-fb-1d",
  "port": 4,
  "tstamp": "2018-01-
30T00:18:17.286593Z",
  "raw": {
    "type": 2,
    "occ": false,
    "measure": 961, ...
  },
  "freq":"868.1",
  "snr":"8.5", ...
}
```

Fig. 6 JSON output for the parking sensor example

C. Ontology Example

We have developed an external ontology called PSO (Parking Sensor Ontology) for this sensor type keeping in mind the compatibility with, and thus the easy mapping to the

oneM2M Base Ontology. We tried to incorporate all the functions provided by the sensor into our PSO. Table II shows the classes and some explanation of our PSO.

Table II Classes of the Parking Sensor Ontology

Class name	Description	Main subclasses
Device	Parking sensor	–
Function	Functionalities of the device to accomplish the tasks of the parking sensor	• Measurement • Status • Control
Command	Represents an action to support a function	• Start • Stop • Reset • Set • Send
Service	Function for the network	–
Property	Measured values	–
Status	Actual sensor status	• OnOff • Error
Position	Actual sensor position	–
Operation	Communication of the Service over the network	–
Input	Input type of an Operation	–
Output	Output type of an Operation	–
Unit	Units of the measured values	–

Our Parking Sensor Ontology and the Base Ontology are compatible. The classes of our PSO can be easily mapped to the relevant classes of the BO. Fig. 7 shows this mapping. We used *isa* relationship between the PSO classes and the key BO classes, which corresponds to the subclass property in oneM2M.

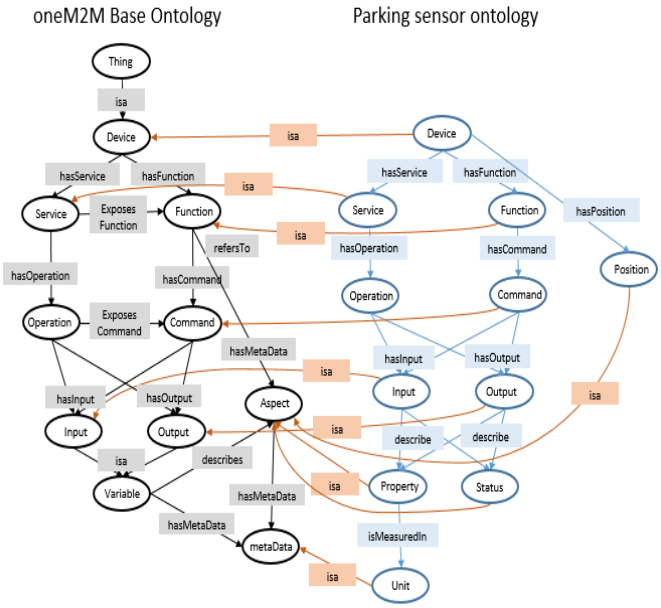

Fig. 7 Mapping of our PSO to oneM2M Base Ontology

V. Conclusion

In this paper, we introduced and discussed our purpose developed, new script language, called L4SDD, for dynamic sensor data description and data format conversion, and our cross-domain, ontology-based semantics to achieve syntactic and semantic interoperability across IoT domains. Moreover, we demonstrated these approaches in a real-life smart parking case study.

As a future work, we plan to implement a context sensitive web based editor for L4SDD to facilitate the creation of the sensor data description scripts for third party users. Moreover, we plan to develop a method which is able to couple the semantic data (metadata) to the sensor data descriptors in an efficient manner. Finally, we intent to integrate our framework into our Hadoop based IoT platform called SensorHUB [14], thus link the sensor data descriptors to the Hadoop schema registry, and insert the data format conversion and semantic handling solutions into the data flow processing procedures of the IoT platform.

References

[1] V. Berrios, R. Halter, M. Harrison, S. Hollenbeck, E. Kendall, D. Migliori, J. Petze, J. C. Stevens, "Cross-industry Semantic Interoperability, Part I," *Embedded Computing Design*, June 29, 2017. [Online]. Available: http://www.embedded-computing.com/semantic-interop/cross-industry-semantic-interoperability-part-one#

[2] T. Andreas, et al, "Applying the Levels of Conceptual Interoperability Model in Support of Integratability, Interoperability, and Composability for System-of-Systems Engineering," Virginia Modeling Analyses & Simulation Center, Old Dominion University [Online]. Available: http://www.iiisci.org/journal/cv$/sci/pdfs/p468106.pdf

[3] R. Joshi, et al, "The Industrial Internet of Things Volume G5: Connectivity Framework," Industrial Internet Consortium, [Online]. Available: https://www.iiconsortium.org/pdf/IIC_PUB_G5_V1.0_PB_20170228.pdf

[4] Data Distribution Service (DDS), [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF

[5] OPC-Unified Architecture (OPC-UA), [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture

[6] C. Jennings, Z. Shelby, J. Arkko, Media Types for Sensor Markup Language (SenML), [Online]. Available: https://tools.ietf.org/pdf/draft-jennings-senml-10.pdf

[7] Data Format Description Language (DFDL), [Online]. Available: https://www.ogf.org/ogf/doku.php/standards/dfdl/dfdl

[8] Xtext framework, [Online]. Available: http://www.eclipse.org/Xtext/

[9] EPC Information Services (EPCIS) Standard, [Online]. Available: (https://www.gs1.org/sites/default/files/docs/epc/EPCIS-Standard-1.2-r-2016-09-29.pdf

[10] Project Haystack, [Online]. Available: https://project-haystack.org/doc

[11] SAREF Ontology, [Online]. Available: http://ontology.tno.nl/saref/

[12] Base Ontology, Technical Specification, TS-0012-V3.7.1, [Online]. Available: http://www.onem2m.org/component/rsfiles/download-file/files?path=Release_3_Draft_TS%255CTS-0012-Base_Ontology-V3_7_1.docx&Itemid=238

[13] Libelium Smart Parking, [Online]. Available: http://www.libelium.com/downloads/documentation/plug_and_sense_smart_parking_technical_guide.pdf

[14] L. Lengyel, P. Ekler, T. Ujj, T. Balogh, H. Charaf, "SensorHUB: An IoT Driver Framework for Supporting Sensor Networks and Data Analysis," *International Journal of Distributed Sensor Networks*, 2015(1):1-12, July 2015