The Dynamic Sensor Data Description and Data Format Conversion Language

Gergely Mezei[1], Ferenc A. Somogyi[1] and Károly Farkas[2,3]
[1]*Department of Automation and Applied Informatics, Budapest Univ. of Technology and Economics, Budapest, Hungary*
[2]*Department of Networked Systems and Services, Budapest Univ. of Technology and Economics, Budapest, Hungary*
[3]*NETvisor Ltd., Budapest, Hungary*

Keywords: Sensors, IoT, Compiler, Syntactic Interoperability, Data Transformation, Data Conversion.

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. To achieve syntactic interoperability (the ability to exchange uniformly structured data) is still an issue under research. In this paper, we introduce our purpose developed new script language called Language for Sensor Data Description (L4SDD) to achieve cross-domain syntactic interoperability. L4SDD defines a unified output data format and specifies how the data is to be converted into this format from the various sensor inputs. Besides the language itself, we also present the main features of the workbench created to edit and maintain the scripts and introduce the IoT framework around the solution. The approach provides a high performant, secure and easy-to-use solution to transform their data to an easily processed, self-describing, universal data structure. Although the paper contains implementation details, the solution can be used in other, similar projects as well. As a practical validation, we also illustrate our solution via a real-life case study.

1 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 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 (Berrios et al., 2017).

The Virginia Modeling Analysis and Simulation Center's Levels of Conceptual Interoperability Model (LCIM) (Andreas et al., 2013) defines three categories of interoperability: technical, syntactic and semantic (Joshi et al., 2017): (i) Technical interoperability is the fundamental ability of a network to exchange raw information. (ii) 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 provide syntax that allows systems to recognize the type of data being transmitted or received. (iii) Semantic interoperability enables systems to interpret meaning from structured data in a contextual manner.

Although there is an urge to solve all three levels of interoperability, but only the technical part is solved in general.

We are currently working on an IoT framework collecting data from sensors, transformation the data to a universal format, storing the data in a database and providing data analytics for the data collected. The framework has many components, one of them is responsible for the syntactic interoperability. In this paper, we focus our attention on this topic and introduce our solution to the well-known issues of processing heterogeneous data uniformly.

To achieve syntactic interoperability, we have defined a new script language for dynamic sensor data description and data format conversion called Language for Sensor Data Description (L4SDD). We have built a compiler and a workbench for the language, thus, now it acts as a complete framework to define data transformations for IoT sensors.

Mezei, G., Somogyi, F. and Farkas, K.
The Dynamic Sensor Data Description and Data Format Conversion Language.
DOI: 10.5220/0006912203380346
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 338-346
ISBN: 978-989-758-320-9

2 THE BACKGROUND

2.1 The Framework

We have been developing a framework called the Heterogeneus IoT Sensor DAta Management and Integration Framework (HISDAMIF). The framework implements functions to register and manage sensors, handle sensor data collection, conversion, interpretation, management, and storage. The features are divided into two logical groups: sensor management and message handling. Both feature groups are supported by modules.

If a new sensor is introduced, the following modules are used: (i) The device is registered via the **Sensor Registration** module, which handles the registration, the configuration, and monitoring of the device. (ii) We create a message conversion algorithm that transforms the sensor data into our universal, self-describing data format. The algorithm is created by using the **Data Transformation** module, which is also responsible for registering the data transformation algorithms in the system. (iii) The **Ontology** module (based on our oneM2M Base Ontology (OneM2M: Base Ontology, 2018) implementation) is used to interpret the collected data unambiguously. (iv) The sensor description consisting of the three aforementioned part is stored by the **Inventory** module that acts as a catalog of sensors and data processing algorithms.

The second scenario is when a previously registered sensor sends data. (i) The data is received by the **Sensor Data Management** (SDM) module, which queries the registered data transformation algorithms (using the Inventory module). SDM selects the algorithms applicable (based on a filtering method as discussed later) and applies them. (ii) As the result of data transformation, the heterogeneous sensor data is converted to our universal format. We store this data in a Hadoop-based database by using the **Data Storage** module. IoT applications can be built on top of this central data store.

In this paper, we focus mainly on the Data Transformation module. We present the key to flexible data transformation, and introduce our new, purpose developed script language capable of describing sensor data independently of its domain, and also capable of defining the transformation logic. The paper also presents the main aspects of the workbench built around the language.

2.2 Syntactic Interoperability

It is rather challenging to support different formats of data stemming from various IoT devices. In this case, the data should be normalized or converted to a common form, which can be read by all elements of the system or even by different systems leading to syntactic interoperability (Figure 1).

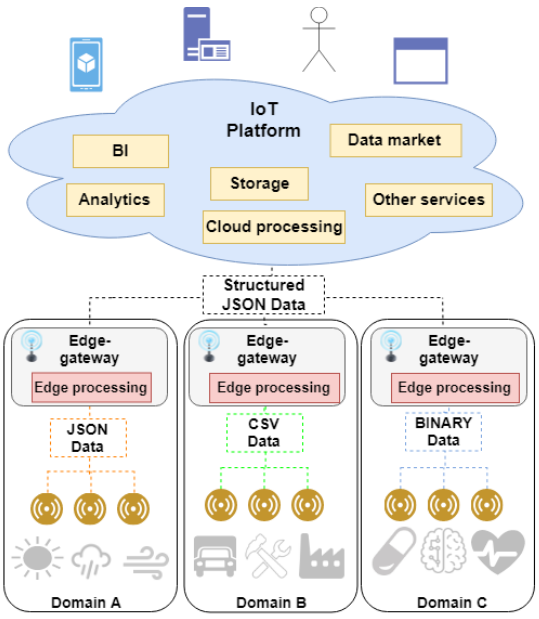


Figure 1: Handling heterogeneous data sources.

To achieve syntactic interoperability, first, we tried to create a static description, namely a data format definition to describe all variants of all sensor messages. It is worth mentioning 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 to 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. Another example is when the first field contained the number of key-value pairs stored in the rest of the message to process. 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 the Language for Sensor Data Description (L4SDD).

L4SDD not only defines an output data format but also specifies how the data is to be calculated from the various sensor inputs. The converted data can later be directly stored in databases and processed by data analysis techniques. The main goal of this paper is to present this language and the workbench around the language in detail. Although the discussion is based on our concrete approach and implementation, the ideas and solutions are generic, thus they are also useful in other environments.

2.3 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 identifying such a format, the capabilities of the sensor devices are strongly limited, and they cannot convert the data to the desired format by themselves. Thus, our goal was twofold: (i) a format that can describe the format of all data independently of its domain, and (ii) a solution to transform the original data to this standard form.

The Data Distribution Service (DDS) (OMG: Data Distribution Service, 2015) 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, but data transformation is not considered.

The OPC-Unified Architecture (OPC-UA) (OPC-Unified Architecture, 2015) 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 various, incomplete implementations.

The Sensor Markup Language (SenML) (Network Working Group: Sensor Markup Language, 2013) 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) (Open Grid Forum: Data Format Description Language, 2014) (McGrath et al., 2009) 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 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. In contrast, the implementation platform of our data conversion framework must be modifiable (e.g., instead of Java, we should be able to switch to a JavaScript platform). 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.

3 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. Before discussing the language, it is useful to introduce, how the data processing algorithm is applied in our framework. When the script is created, it is compiled to source code to the target platform, currently to JavaScript, but it is configurable. The generated data processor function is then registered by the framework. Later, if the framework receives a sensor data message, the registered data processors are queried. All processors have a filter that decides, whether the processor is applicable for the data, or not. If the answer of the filter is positive, the data transformation is applied.

L4SDD scripts consist of several sections: (i) an *Output* definition that describes the format of the output data; (ii) a *Filter* definition that is used, when the framework tries to find scripts applicable to the specific data; (iii) a *Mapping* definition that defines the conversion itself, namely how the output data is 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 parameters are not sent by the sensor to the framework, instead, the framework appends the information as an additional input parameter for the script when it is executed.

The *Output* and *Params* sections use the same language elements and syntax (they are static format descriptions), while the *Filter* and *Mapping* sections also share their syntax (they are transformation logic descriptors). The four sections together allow 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.

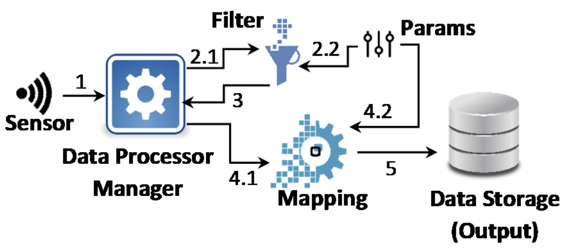


Figure 2: Steps of sensor data processing.

Figure 2 shows the typical steps of processing the sensor data. First, the data is received by the Data Processor Manager (DPM) inside the Sensor Data Management module. DPM forwards the data to the Filter algorithm (Step 2.1), which also receives additional parameters (Step 2.2). It should be noted that the section Params describes only the structure of additional parameter data, the concrete values are calculated by the Data Processor Manager. This later relation is not shown in the figure for the sake of simplicity. When the filter is processed, it sends back its result (Step 3) to DPM. If the result is positive, the data is forwarded to the Mapping algorithm (Step 4.1) accompanied by parameters (Step 4.2). Finally, the result of *Mapping* is sent to the Data Storage module. It is worth noting that the format in Step 5 fits the *Output* definition provided in the script.

The *Output* and *Params* sections describe the data structure produced as the result/accepted as a parametrization at the beginning respectively. The syntax of these sections is relatively simple: we define data tags with names and types. The type can be one of the built-in types (e.g., integer, string or DateTime), or a complex type composed from other types hierarchically. A complex type is similar to a *struct* in the C language. The multiplicity of the tags can be altered by allowing optional multiplicity (the tag can be omitted), and arrays. Fixed-length and variable length arrays are both supported.

There are a few restrictions to follow as well: the multiplicity of the outermost element of the hierarchy must be exactly one, namely, it must not be optional or an array, and it must have a name. This seems to be a limitation, however, we can always create a nesting (outermost) type and use an arbitrary multiplicity within. Another restriction is that we do not support empty hierarchies (hierarchies without any data tag). These restrictions do not limit the expressivity of the solution, however, they do simplify the processing of the data.

3.1 The Data Transformation Logic

The dynamic part, i.e., the data transformation description of L4SDD is a Java-like imperative language with several restrictions (compared to Java), but, on the other hand, supporting custom functions in order to simplify the transformation.

3.1.1 The Type System

The language of L4SDD is strongly typed. It supports many types known from programming languages, such as integer, double, string or DateTime. We will refer to these types like primitive types from now on in order to distinguish them from complex types composed of several fields. The users cannot define new primitive types, but they are allowed to combine existing types to create new complex types. Moreover, we can also create arrays from any known type. These arrays can be of fixed or variable length. All types (primitive and complex) are able to serialize and deserialize themselves to/from string format. The conversion between the type and string is implicitly supported thus simplifying value assignments.

3.1.2 Variables

The scripts have built-in variables accessible from the *Filter*/*Mapping* sections of the script: (i) INPUT: Can be accessed from both *Filter* and *Mapping*. The input data of the script (the data to process) represented as a variable. The type of the variable is byte array. (ii) SOURCE: Can be accessed from both *Filter* and *Mapping*. The source of the input data (i.e., the identifier of the sensor sending the data) as a string. (iii) OUTPUT: Can be accessed from *Mapping* only. The structure of the output variable is specified by the *Output* section. (iv) PARAMS: Can be accessed from both *Filter* and *Mapping*. Parameters passed to the script. The structure of the variable follows the definition of the *Params* section.

Note that in *Filter*, we do not have an explicit output variable, since *Filter* definitions consist of assert statements. The result of *Filter* is true if none of the assertions fails.

Besides these global variables, it is also possible to define and use local variables. The scope of local variables are handled as usual: they end at the end of their defining block. The type of the local variables can be any valid type according to the aforementioned type system.

3.1.3 Language Statements

The language borrows several concepts and solutions from modern programming languages such as Java. These concepts include arithmetic, relational, bit and logical operations, assignment, conditional branches and loops (for, while). The language also supports a foreach statement providing an easy to use way to iterate through a collection. Specifying the type of the iterator variable is not necessary in case of *foreach*, since it can be resolved from the collection.

Compared to classic programming languages, L4SDD is limited to several extents since it does not support (i) user-defined functions in the scripts, (ii) visibility settings of data tags in complex types, (iii) reference types, (iv) custom primitive types (as mentioned before) and custom error types.

The language also contains several special constructs to simplify data processing. Since processing of sensor data often requires working with byte and bit arrays, the feature is supported by various built-in functions. For example, the expression `Bit(x, 3, 4)` returns the third to sixth bit of variable x. For the sake of simplicity, this can also be expressed by array indexers, such as `x[3..6]`.

Besides byte arrays, the input data tend to contain JSON or CSV (comma separated value) data fragments. To support JSON, the language has a function that produces a hierarchy of key-value pairs based on a string representation of JSON-encoded data. Similarly, there is a function that produces a two-dimensional array from CSV data, where the item/row separating characters can be specified. It is also a common case that we encode key-value pairs as CSV data. For example the string `"ack":false, "port":4, "cls":0` describes three pairs (the keys are “ack”, ”port” and “cls”). Moreover, the two CSV styles can also be mixed (some of the values have a key, others have only an index).

Finally, it is a common task to extend an array with a new element and to initialize it just before it is added to the collection. In order to support this, we have an Add function, which is somewhat unconventional, since it also has a statement block after the function call. When calling the function, we can pass a variable that will point to the newly created item. This variable can be an existing one (defined previously somewhere earlier), or a new one (the variable is then freed when the execution of the block is ended). If an unhandled error occurs inside the block, the new element is not added to the array.

3.1.4 Error Handling

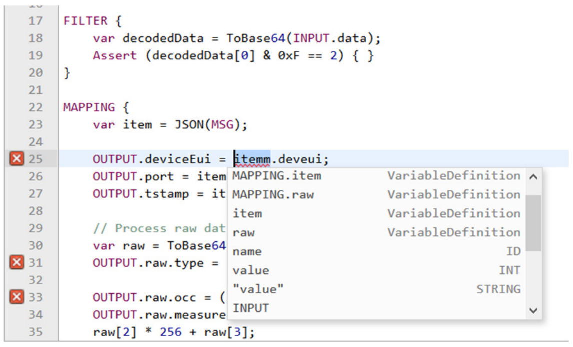
Safety plays an important role in our project, thus having a stable and safe error handling solution is also important. The error handling follows the well-known try-catch pattern. If an error occurs inside a try block, the execution is stopped, and the appropriate catch blocks are executed. However, L4SDD handles each block automatically as a try block: the users can attach error handling (catch) blocks to them matching a specific type of error or specifying that all errors must be caught. It is also possible to specify multiple error types for a specific catch block.

If there is no matching catch block for the error, then it is forwarded to the container block and so on, until the outermost block is reached. The outermost block automatically handles all unhandled errors and creates a log message about them. If there is an unhandled error in the *Filter* section, it returns false, which means the automatic rejection of the message (the script is not meant to process the data). Besides the implicit “try” statement behavior of blocks, assignments also act as try blocks. If the assignment cannot be applied, the value of the destination variable remains untouched, and no errors are thrown. If the user wants to change this behavior, it is possible by explicitly overriding it.

4 THE WORKBENCH

We have built a compiler for L4SDD based on Xtext (Eysholdt and Behrens, 2010) (The Xtext Framework, 2018). In order to make the definition of L4SDD scripts as user-friendly as possible, we provide a feature rich integrated development environment (IDE) based on Xtext. Originally, we were working with the standard, Eclipse-based version of Xtext, but later we decided to switch to a JavaScript-based web editor also based on Xtext. The web-based editor (Figure 3) is reachable from anywhere, and it is easier to integrate into our system. The editor communicates with an Xtext servlet, which contains the compiler, and does the parsing. From the viewpoint of the user, there is no functional difference compared to using a standard IDE. The editor provided by Xtext supports many features that make the editing process of the L4SDD scripts as user-friendly as possible, e.g., syntax highlight, semantic highlight, content assist, validation (semantic analysis).

During the compilation process, we generate the source code for the target language, which is currently JavaScript. It is worth mentioning that we can also



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

MAPPING {
    var item = JSON(MSG);

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

    // Process raw dat
    var raw = ToBase64
    OUTPUT.raw.type =

    OUTPUT.raw.occ = (
    OUTPUT.raw.measure
    raw[2] * 256 + raw[3];
```

```
MAPPING.item        VariableDefinition
MAPPING.raw         VariableDefinition
item                VariableDefinition
raw                 VariableDefinition
name                                ID
value                              INT
"value"                         STRING
INPUT
```

Figure 3: L4SDD Web Editor.

simultaneously support code generation for multiple target languages if needed. The types we defined in L4SDD are all mapped to types in the target language. This is done by a configurable type system. Hierarchical types can also be mapped (usually to classes or structures), if the target language is object-oriented. In the cases of weakly-typed target languages like JavaScript, there is no need for the type conversion. L4SDD also supports the mapping of commonly-used functions to the target language. The functions of L4SDD are mapped to custom functions in the target language with the aid of our function mapping system similarly to type mapping. Note that in this case, the implementation of the functions must exist in the target language, thus, a function library is required for each language supported. The function library is written by hand by the framework provider when the new target language is added. For example, the *Base64* function (the purpose of which is to encode a binary array to Base64) in L4SDD is mapped to the *toBase64* function written in JavaScript as part of the framework. The generated code then calls *toBase64* everywhere *Base64* is called in the script.

5 ANALYZING THE SOLUTION

Our original goal was to create a language that supports the conversion of sensor data given in any format to a common data format that is easily processed in later steps. An additional goal was to create a user-friendly workbench around the language that allows the users to benefit from the advantages of the language in a comfortable way. While these goals are clearly reached by L4SDD and its workbench, the solution has additional advantages as well.

Possibly the most important benefit is that the scripts are not bound to a specific programming library or even to an execution platform. While L4SDD is similar to general purpose programming languages and thus, easy to learn by programmers, the scripts can be compiled into any modern programming language. Currently, we produce JavaScript code from the scripts, but if we would need to use another language, we only need to replace the code generator of our compiler. This means that we can change the target platform without touching the scripts. By taking into account how fast the evolution of programming libraries are especially in web-based areas, this advantage can efficiently simplify the maintenance of the data transformation logic.

As the direct consequence of the previous benefit, it is also possible to generate code for different platforms from the same script. For example, we can generate code for a JavaScript and a Java environment simultaneously, thus, supporting several environments at the same time. The key is the separation of data processing logic and its implementation.

The fact that we have full control over the scripting language makes it possible to add efficient features to the language. Currently, this is mainly limited to bit, byte, JSON and CSV processing functions, but later, we can extend the language based on user feedback. Obviously, this would also be possible in a general programming language by creating utility functions, but in a purpose developed script language, we have more flexibility (e.g., the function Add) and more options to customize and adapt to the user requests.

The usage of a purpose developed script language not only simplifies the code generation, but it also helps in avoiding dangerous, malicious codes. By restricting the available functionality, we can guarantee the safety of the scripts much easier. As mentioned in the language description, safety is also supported by custom error handling.

Safety is supported not only by the language itself but also with the script management lifecycle. The scripts must be editable by the users, but this does not necessarily mean that the generated code must also be shared with them. The users edit the scripts, then the scripts are uploaded to the server. They are compiled there, and the data transformation algorithm (described by the script) is registered and later used by the framework and not by the user. By hiding the generated source code from the users, it is much easier to avoid security issues, since the users do not know how their script is implemented. They do not even know to which programming language we compile their scripts. Moreover, this also allows us to replace the target platform any time, transparently.

By completely controlling the compiling process and hiding the generated source code from the user, we can also add debugging and logging instructions to the code if it is required, for example, while testing. This solution also makes it possible to map usual function calls, create statistics from them and optimize the generated code based on the results.

To sum it up, by following our approach, the users obtain a high performant, secure and easy-to-use solution to transform their data to an easily processed, self-describing, universal data structure.

6 CASE STUDY

In the following, we illustrate our approach on a simple, smart parking scenario. In this scenario, we use the Libelium Smart Parking sensor solution (Libelium Smart Parking, 2018). We apply the sensors to detect available parking slots in a parking area based on the measured values.

Our goal is to convert the data sent by the parking sensors to a common data format. The raw sensor data of our example can be seen in Code 1. For the sake of understandability, we omit some details

```
{"ack":false,freq":"868.1",snr":"8.5",
"deveui":"00-00-00-00-00-1a-fb-1d",
"data":"ArEDwQAAAAsAAAA=", ... }
```

Code 1: Raw sensor data.

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. The sensor does not maintain a strict ordering between the fields, their order may change between messages. Most of the data (e.g., ID, frequency) can be converted without further processing. However, the raw data parts 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).

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* section. The schema description for the parking sensor example can be seen in Code 2. The type definitions in the description (e.g., *byte*, *int*) are specific to L4SDD, and they can be mapped to types in the target language that we generate the code for.

```
OUTPUT {
 deviceEui : byte[32];
 raw:
 {
  type : int;
  occ : int;
  meas : float;
 };
 freq : float;
 snr : int; }
```

Code 2: The Output definition.

The second step is to define the processing logic in the script. The purpose of the *Filter* section (Code 33) is to check if the data can be processed by the script. Here, we check the first byte of the raw data, which identifies the message type sent by the sensor.

```
FILTER {
var decData = Base64(INPUT.data);
Assert (decData [0] & 0xF == 2){…} }
```

Code 3: The Filter definition.

Then, the *Mapping* section (Code 4) contains the processing logic for the script. In the example, we process the raw data by using standard instructions in the L4SDD language. The rest of the data can be directly copied to the output without processing.

```
MAPPING {
 var item = JSON(INPUT);
 OUTPUT.deviceEui = item.deveui;
 var raw = Base64(item.data);
 OUTPUT.raw.type = Int(SStr(…));
 OUTPUT.raw.occ = (Int(SStr(…) === 1);
 OUTPUT.raw.meas = raw[2]*256+raw[3];
 OUTPUT.freq = item.freq;
 OUTPUT.snr = item.lsnr; }
```

Code 4: The Mapping definition.

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, e.g. to Java.

A snippet of the generated code of our example can be seen in Code . *Filter* and *Mapping* are both mapped to a JavaScript function (*isAccepted* and *processData*). The system calls these functions with the specific sensor data.

```
function isAccepted(input) {
    let decData = Base64(input.data);
    if ((decData [0] & 0xF) != 2)
    {   return false;    }
    return true; }
function processData(input, params,
                             message) {
    let item = JSON(message);
    let raw = Base64(item.data);
    strOutput.deviceEui = item.deveui;
    strOutput.raw.type = Int(…);
    strOutput.raw.occ = ((Int(…))===1);
    strOutput.raw.meas =
                  raw[2]* 256 + raw[3];
    strOutput.freq = item.freq;
    strOutput.snr = item.lsnr;
    return structuredOutput; }
```

Code 5: The generated source code.

Finally, Code 6 depicts the structured JSON output for the parking sensor example, containing the value information. The type information (which contains the structure of the output) is omitted. We can use this information later, durint the storage of the data.

```
{"deviceEui":"00-00-00-00-00-1a-fb-1d",
"raw": {
  "type": 2,
  "occ": false,
  "meas": 961},
"freq":"868.1",
"snr":"8.5" }
```

Code 6: The structured output.

In the example, the structure and syntax of the generated L4JS code are very similar to that of the L4SDD script, since the example is simple. However, in case of more sophisticated, real-life scenarios, L4SDD can also simplify the syntax of the script.

7 CONCLUSIONS

With the steeply growing popularity of IoT devices, there is an increasing need to simplify interoperability between these devices. Defining a universal data format is not enough since most of the sensors do not support adapting to a specified format. Our research aims to create a new method that includes a universal self-describing data format and a transformation language able to convert data of all kinds to this format. Our solution is based on a new language referred to as the Language for Sensor Data Description. The paper elaborated on the main properties of the language and also discussed its environment, the workbench and showed how the language fits into the ecosystem of our generic IoT framework.

Our current results are promising, but we have many plans for the future. Currently we are working on a debugging system for the script to simplify finding and correcting the errors in the script. We also plan to add semantic interpretation of the input data (supporting semantic interoperability) and to apply more case studies in real, industrial environments. Moreover we plan to give access for a wider audience to use and therefore test our solution.

ACKNOWLEDGEMENTS

The work presented in this paper has been carried out in the frame of project no. 2017-1.3.1-VKE-2017-00042 and performed in the frame of the FIEK_16-1-2016-0007 project, both implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2017-1.3 and the FIEK_16 funding scheme respectively.

REFERENCES

Andreas, T., Diallo, S. Y. & Turnitsa, C. D., 2013. Applying the Levels of Conceptual Interoperability Model in Support of Integratability, Interoperability, and Composability for System-of-Systems Engineering. Systemics, Cybernetics And Informatics , 5(5), pp. 65-74.

Berrios, V. et al., 2017. *Cross-industry semantic interoperability, part one.* [Online] http://www.embedded-computing.com/embedded-computing-design/cross-industry-semantic-interoperability-part-one [Accessed 20 04 2018].

Eysholdt, M. & Behrens, H., 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion,* pp. 307-309.

Joshi, R., Mellor, S. & Didier, P., 2017. *The Industrial Internet of Things Volume G5: Connectivity Framework.* s.l.: Industrial Internet Consortium.

Libelium Smart Parking, 2018. [Online] http://www.libelium.com/downloads/documentation/plug_and_sense_smart_parking_technical_guide.pdf [Accessed 20 04 2018].

McGrath, R. E., Kastner, J. & Myers, J. F. M., 2009. *Experiments in Data Format Interoperation Using Defuddle,* Illinois: University of Illinois.

Network Working Group: Sensor Markup Language, 2013. *SENML*. [Online] https://tools.ietf.org/html/draft-jennings-senml-10

OMG: Data Distribution Service, 2015. *DDS*. [Online] https://www.omg.org/spec/DDS/1.4/PDF [Accessed 20 04 2018].

OneM2M: Base Ontology, 2018. [Online] www.onem2m.org [Accessed 20 04 2018].

OPC-Unified Architecture, 2015. *OPC-UA*. [Online] https://opcfoundation.org/developer-tools/specifications-unified-architecture [Accessed 20 04 2018].

Open Grid Forum: Data Format Description Language, 2014. *DFDL*. [Online] https://www.ogf.org/ogf/doku.php/standards/dfdl/dfdl [Accessed 20 04 2018].

The Xtext Framework, 2018. *http://www.eclipse.org/Xtext/*. [Online] www.eclipse.org/Xtext/ [Accessed 20 04 2018].