A Model Driven Approach Accelerating Ontology-based IoT Applications Development

Charbel El Kaed
Schneider Electric, IoT & Digital Transformation
first.last@schneider-electric.com

André Ponnouradjane
Schneider Electric, IoT & Digital Transformation
first.last@schneider-electric.com

ABSTRACT

The Internet of Things promises several exciting opportunities and added value services in several industrial contexts. Such opportunities are enabled by the interconnectivity and cooperation between various things. However, these promises are still facing the interoperability challenge. Semantic technology and linked data are well positioned to tackle the heterogeneity problem. Several efforts contributed to the development of ontology editors and tools for storing and querying linked data. However, despite the potential and the promises, semantic technology remains in the hands of the few, a minority of experts. In this paper, we propose a model driven methodology and a software module (OLGA) that completes existing ontology development libraries and frameworks in order to accelerate the adoption of ontology-based IoT application development. We validated our approach using the ETSI SAREF ontology.

KEYWORDS

Internet of Things, Ontology, Model Driven Engineering

1 INTRODUCTION

The Internet of Things (IoT) is expected to interconnect, at massive scale, numerous sensors, devices, gateways, and systems to tackle many challenges in the industry [15]. Such inter-connectivity will play an essential part in designing industrial systems with added value services which are more energy efficient with lower costs while contributing to a better environment. These promises promoted by the emergence of the industrial Internet of Things (IoT) have surged the importance of interoperability among the things to turn this vision into reality.

Designing IoT applications requires a shared understanding of the exchanged data among those connected things. Semantic technology, is one of the most promising fields in the knowledge representation domain, expected to enable interoperability in the IoT. The World Wide Web Consortium (W3C) defines a set of standards , such as RDF, OWL and SPARQL [11, 38, 42], to represent semantics and query linked data, offering an ideal ecosystem and opportunity to tackle the heterogeneity challenge in the IoT. In industrial environments and automation domains, semantic technology has been used to solve data interoperability issues [14, 24] and to provide context aware applications and services.

In addition to the W3C standardization activities, several efforts contributed to the development of the ontology editors [7], storage [6], inference engines, as well as graphical tools to represent, edit, and query linked data. Furthermore, serializers such as RDF4J[26] and Object Relational Mappers such as RomanticWeb [33] are available for developers that are not ontology experts.

Despite its potential and promises, semantic technology and ontology-based IoT applications still remain in the hands of a minority, the ontology experts, being too difficult to be adopted and applied by industrial practitioners. We attribute such retention among other factors to the absence of adequate methodology and tools involving several major actors participating in the design lifecycle of an IoT application, who are typically non-ontology experts. Thus, we propose in this work a model driven methodology along with a software module approach that aim at removing barriers for IoT developers and accelerating the adoption of semantic technologies. Our proposed module is validated based on the SAREF [5] ontology.

The rest of this paper is organized as follows, first, we depict some of the existing frameworks for developers in the related work section. Then, we share our experiences while working on IoT-based solutions with our internal teams and outline the motivation behind this work. In sections 4 and 5, we propose our methodology along with a software module OLGA to accelerate the ontology-based IoT development. Section 6 provides the implementation and evaluation of our solution. Finally, we conclude and discuss future work in section 7.

2 RELATED WORK

We split the available libraries and frameworks in various programming languages for ontology-based development in two categories, i.e., serializers and Object Relational Mappers (ORMs).

2.1 Serializers

Serializers provide reading/writing from/to an ontology file, a SPARQL endpoint, or a persistent RDF store. RDF Serializers are implemented in various programming languages, such as OWL API [25], RDF4J [26], and Jackson-Jsonld [18] in Java, dotNetRDF [30] in .Net, Redland [28] in C, and RDFLib [27] in Python. The serializers' APIs provide low level classes and functions to manipulate concepts directly mapped to the Rdf language without any higher level abstractions. Therefore, it is required by any IoT developer to be aware of the technical aspects and theory of the RDF concepts and principals in order to implement ontology-based IoT applications.

We discuss in the following the serializers offering basic code generation through a plugin which takes an ontology as an input and generates some code facilitators or stubs.

The Protégé code generation plugin [34], which can be easily integrated in Protégé [7] provides generation of Java code based on the OWL API [25]. However, the code generation is partial where only the class name and interface are provided along with an empty constructor. Then, it is up to the developer to complete the generated code by relying on the OWL API which requires a learning curve since it is directly mapped to the RDF concepts.

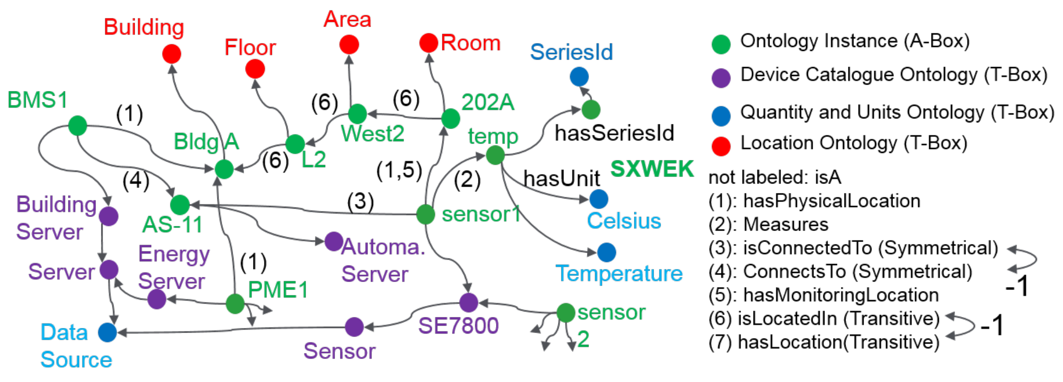

Figure 1: Part of the Ontology Representation [9, 19]

The RDF4J Schema generator [36] extends the RDF4J [26] API and provides an automatic generation of an RDF schema from an ontology. The generated output of the ontology is contained in one java file which contains only the IRI of each concept of the ontology. In other words, the code generation is flat, there are no classes, associations, or constraints between the generated elements. It is up to the developer to implement the association, mapping, and constraints manually.

AutoRDF [4] extends the Redland [28] library and proposes a generator which takes an ontology and generates C++ object oriented code to manipulate RDF concepts. The generated code is an abstraction layer which consists in a set of functions based on the ontology classes and relations available to be used by developers to generate ontology instances (A-Box).

2.2 Object Relational Mappers (ORMs)

ORMs are built on top of serializers and provide an object oriented abstraction layer allowing developers to manipulate objects instead of RDF concepts. Several ORMs are available in various programming languages, such as KOMMA [23], Empire [17] and AliBaba [29] in Java, RomanticWeb [33] and TrinityRDF [40] in .Net, and RDFAlchemy [35] in Python. ORMs rely on the code decoration where a developer annotates her code with tags referencing IRIs from the ontology terminology (T-Box). Most of the Java ORM rely on the Java Persistence API (JPA) [16] while the .Net ORMs rely on the Entity Framework [8]. During the code implementation of an application, the developer requests a factory to instantiate the ontology (A-Box) and can formulate SPARQL queries by relying on SPARQL query builders or adapters such as the LINQ to SPARQL in the .Net domain.

We discuss in the following the ORMs providing some code generation features.

AliBaba [29] offers the three following interesting features for ontology developers. a) The object server exposes the object factory through a REST API putting the available objects as resources on the web for manual annotation. b) The aspect behaviors which allow each object of the factory to intercept a method call and execute a specific behavior. Annotation such as precedes provides the developer with a better control with the behavior execution flow. c) SPARQL queries decoration on the getter/setters of objects which enables dynamic queries execution. In fact, compared to other ORMs, this feature is similar to invoking SPARQL queries in the implementation methods. AliBaba highlights a java code generator which seems to handle simple ontologies, it failed to generate code when tested with the SAREF ontology [32]. AliBaba provides interesting concepts for ontology based development, however, it is clearly targeting ontology and not IoT developers since RDF and SPARQL are part of its APIs and design.

KOMMA relies on the Eclipse Modeling Framework (EMF) [10] and is inspired by AliBaba's design. KOMMA provides a unified framework with the following three layers: an object mapping persistence layer, visualization tool, and an ontology editor based on the capabilities of the EMF. KOMMA mentions a code generator plugin, however, it is not integrated in visualization and editing layers, therefore, the mapping and implementation of the interfaces remain manual. KOMMA's unified approach clearly targets ontology developers with the integration of the three layer in a common framework. A learning curve is expected from an IoT developer for both the ontology editing and the object mapping which consists in decorating the code with concepts from the ontology.

In the following, we depict our industrial context, the feedback from our teams regarding ontology-based development, and the lessons learned.

3 INDUSTRIAL CONTEXT & LESSONS LEARNED

This section briefly outlines our industrial context capturing part of our lessons learned while prototyping with several of our internal teams applying semantic technologies in several domains such as the Smart Buildings and Smart Factories.

Schneider Electric is a world leader in Energy Management and heavily involved in the IoT and the digitization of its various products. The EcoStruxure[1] program proposed by Schneider Electric, is an open, interoperable, IoT-enabled system architecture and platform. EcoStruxure serves our business units which are grouped according to the following six domains of expertise [2] : Building, Power, IT, Machine, Plant, and Grid. Our innovation department is involved in the EcoStruxure program and accompanies any team

[1]http://www.schneider-electric.com/en/download/document/998-19885906_GMA-US_ESX/

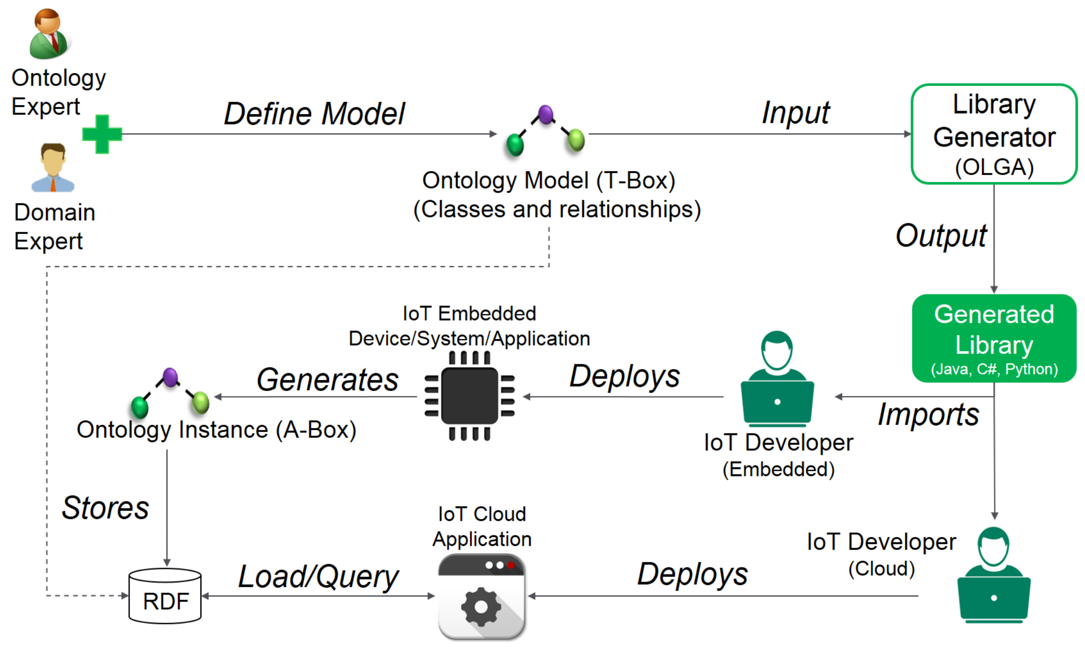

Figure 2: Proposed Methodology

from the six domains in their IoT and digitization journey. Part of our mission is to investigate new technology and apply it to our industrial context in any of the domains of expertise. Semantic technologies is one the most promising approaches to provide interoperability in the IoT domain. Therefore, we collaborate with those internal teams to enable faster IoT interoperable solutions by applying semantic technologies such as in [9, 19, 21].

Two main factors captured the curiosity and the interest of our internal teams and communities regarding ontology-based IoT applications development.

First, the proven technical feasibility of ontology-based vertical and horizontal solutions in our industrial context. Several ontology-based proof of concepts have been designed demonstrating the added value in two operating modes: vertical and horizontal deployments. The vertical silo mode is the classical sensor to gateway or device/system to cloud combined with a domain specific application [20, 21]. The horizontal mode consists of integrating different silos (verticals) driving cross domains interoperability among systems and devices. For example, in [9, 19] two industrial systems, a building management and a power solution systems, are connected to the cloud. Then, an adaptation layer exposes the two systems information through an ontology representation to be consumed by a business intelligence layer to extract advanced buildings insights. Furthermore, FOrTÉ [9] a cloud based federated query ontology and timeseries engine is deployed enabling big data queries for business and machine learning applications.

Second, the emergence of domain specific ontologies in the industry and standardization bodies. For example, in the Smart Buildings domain, ontologies such as SAREF [5] and its extensions[2], Haystack [3], and Brick [1] have emerged. In addition, efforts have been proposed regarding energy modeling and access such as Siemens Work [24], SPITFIRE [12], and the Optique Platform[3].

Based on these two factors, we started an internal Ontology Workshop [37] targeting our teams across business units. The aim is to demystify the ontology development for our internal product owners, architects, and developers by providing a theoretical presentation and two hands on lab sessions. The first lab session involves modeling a simple ontology (T-Box) in Protégé [7], similar to the ontology shown in Fig. 1. The second session handles the instantiation of the model (A-Box) and formulating queries to extract data, as shown in Fig. 1 (Ontology Instance), by relying on one of the available RDF development libraries depicted in the previous section.

We outline the lessons learned from these workshops in relation to this work:

(1) Learning curve is a luxury: relying on ontologies is powerful to represent internal systems topology such as the interconnectivity between the devices and sensors, their physical locations, and their functional interactions. However, using a serializer requires knowing in detail the ontology languages. The Object Relational Mappers provide a very interesting abstraction, however, a mapping is still required through code decoration. Thus, there is a necessary learning curve to use the ontologies in the IoT development that most of the teams do not have. IoT developers would leverage a *Just Instantiate and Link* library that they can import into their development environment, where the concepts and relationships of the ontology model (T-Box) are provided. Once such library is imported, they can instantiate the ontology concepts and link them together (A-Box). Such library can rely on the

[2]http://ontoology.linkeddata.es/publish/saref4bldg/

[3]http://optique-project.eu

existing SDKs and libraries presented in the related works section.

(2) Programming Languages: the diversity of our business units and offers translates into the diversity of the skill set of our teams which rely on different programming languages to implement IoT-based applications ranging from embedded devices (c, c++, python), systems (c++, .net, java) to cloud applications. Therefore, such *Just Instantiate and Link* library must support various dependencies to third party libraries and programming languages.

(3) Query Language: storing and retrieving data on a system or cloud is considered essential. The IoT development community is often seen divided regarding the query language between pro and anti SPARQL. Some developers are familiar with query languages such as SQL or LINQ and advocate for their reuse in the IoT ontology-based development. Other developers embrace the power of SPARQL and the necessity of evolving the query language since the underlying structure has also evolved and is no longer tabular. Alternatives such as simple query languages in SQenIoT [21] or visual query builders such as in [41] can be proposed to handle queries or to generate SPARQL through a visual tool and use it in the IoT development.

In the rest of this paper we propose our approach facilitating the ontology-based application development focusing on the first two items depicted earlier.

4 A MODEL DRIVEN METHODOLOGY

Based on the feedback gathered from our internal teams, we propose the following model driven methodology, shown in Fig. 2, for ontology-based IoT application development. We identify the three following major roles involved in the IoT development and leverage their expertise and strength:

(1) Ontology Expert: is an ontology practitioner. Has experience in the ontology development tools, languages, and storage infrastructure. His main tasks consist in assisting the domain expert in capturing the concepts to be used in an IoT application deployed in a specific domain. He creates the ontology concepts and relations (T-Box), and proposes an ontology.
(2) Domain Expert: is a product owner and/or a technical architect. Has a global and specific knowledge on how a specific system is deployed, commissioned, and operated, such as the Building Management System. She articulates the main concepts and the relations of a system to the Ontology Expert.
(3) IoT Developer: implements an application following the ontology model previously defined by the Experts. Imports a library containing the concepts and possible relations previously defined to instantiate the model. Her implementation is integrated in a cloud application or an embedded device.

The methodology shown in Fig. 2 is initiated by both the Ontology and Domain Experts (Product Owner and Architect). They first identify and draw the scope of the IoT application based on the gathered requirements from end clients and users. Then, the main concepts and relations of the application are extracted, for example, in a Smart Buildings domain, a `Floor` and a `Room` are two different concepts related to each other with the `contains` object property which can be declared as transitive. The experts can reuse or extend existing ontologies, after several iterations [13], they converge on a stable model (T-Box).

The Ontological model is then provided to an Ontology Library GenerAtor (OLGA), detailed in the next section. OLGA takes an ontology and its imported ontologies along with a selection of the desired library or framework depicted in section 2. The output of OLGA is a generated library conforming to the Ontology Model previously defined and ready to be used by an IoT developer.

The generated library hides all the ontology complexity and enables IoT developers to instantiate and link the ontology classes and relations previously defined by the Ontology and Domain Experts. The generated library can depend on a serializer or an object relation mapper library. It is used by any IoT developer and is embedded into a device, system, or an IoT cloud application. Once integrated into an existing system or device, it can have several usage. For example, in a commissioning software of a Building Management System [9, 19], it enables to instantiate an ontology model of a system for a given facility. The ontology instance captures several aspects of the system as shown in Fig. 1 (A-Box). It can then be sent along with the model to a cloud storage where a business application layer or an IoT cloud-based application can query and extract information about a specific Smart Building to drive better insights as detailed in [9, 19]. At the cloud level, a developer would also import a library generated by OLGA supporting his choice of the programming language and technology. The generated library conforms to the same ontology defined by the experts and will be used to interact with the RDF store to load and query instances of the model to fulfill the cloud application requirements.

This methodology enables the separation of aspects and roles, and places the complexity in areas where it can be solved by relying on adequate tools and domain expertise. A domain expert has both an overall and specific knowledge about a specific system and domain. An ontology expert is modeling practitioner but lacks the overall system's vision and domain knowledge expertise. And finally, IoT developers can now select the programming language and the framework of their choice to implement ontology-based IoT applications by relying on the generated library which conforms to the ontological model without the complexity of the ontology languages.

In the following section, we detail the Ontology Library GenerAtor (OLGA).

5 OLGA: AN ONTOLOGY LIBRARY GENERATOR

OLGA is a multi-library code generator, as shown in Fig. 3, it takes two parameters as input: one or more ontologies since an ontology can depend on other ontologies and a choice of a library dependency. In fact, the generated library will depend either on a serializer or a an object relational mapper. Thus, OLGA completes already existing libraries and frameworks, those depicted in section 2 by providing IoT developers with the variety of choice for the development of ontology-based IoT applications. OLGA enables the possibility for an IoT developer to choose and reuse existing open source libraries (serializers or ORMs) while offering an abstraction and a simpler

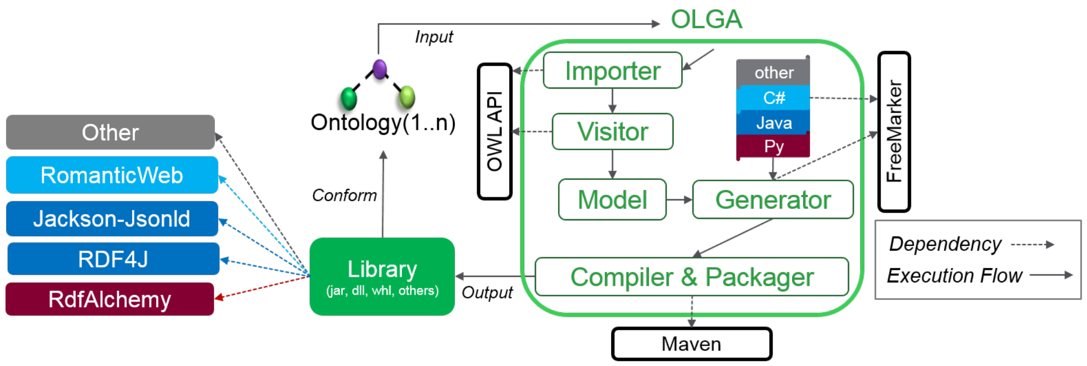

Figure 3: OLGA's Architecture

library to use which conforms to an ontology model previously specified by the experts.

```
<#if consolidatedImports??>
 <#list consolidatedImports as import>
        using ${import};
 </#list>
</#if>

namespace ${Zclass.getPackageName()}
{
 [Class("${Zclass.Iri()}")]
 public interface I${Zclass.ClassName()}
 <#if Zclass.SuperZClassList()??>
  <#if Zclass.SuperZClassList()?has_content> :
   <#list Zclass.SuperZClassList() as
        SuperZClassCurrentElementOfList>
          I${SuperZClassCurrentElementOfList.ClassName()}<#sep> ,
   </#list>
 <#else>: IEntity
 </#if>
</#if>
 {
 <#if listDataProperties??>
  <#list listDataProperties as DataPropertyList>
   [Property("${DataPropertyList.DataProperty()}")]
   ${DataPropertyList.RangeXSDType()}
    ${DataPropertyList
        .DataPropertyShortForm()?capitalize}{ get; set; }

  </#list>
 </#if>
 }

 <#if listObjectProperties??>
 ...
 </#if>
}
```

Listing 1: Part of the template used by OLGA to generate RomanticWeb-based C# code

OLGA consists of the following modules:

Importer: loads into memory one or more ontologies and merges them into one ontology easier to visit.

Visitor: traverses all the elements of a given ontology provided by the Importer. The visitor crosses the following elements: Classes, ObjectProperties, DataProperties, Individuals, Literals, and the various axioms to populate the internal model shown in Fig. 4.

Model: allows capturing the ontology information (T-Box) independent of any targeted library or programming language. Separating the model from any targeted implementation offers OLGA a huge flexibility making the support for an additional language or a dependent library simply a matter of adding templates. The model is populated by the visitor, consists of a representation layer which captures all the elements of an ontology, it is inspired by the work of Kalyanpur *et al.* [22]. The model is shown in Fig. 4. All the elements inherit from the super class Node which is identified by an IRI and a name parameter.

Each Class can have none (`owl:Thing`) or multiple super classes populated by the visitor based on the `owl:SubClassOf`. The namespace, packageName, and className are extracted based on the Class IRI for the code generator .

A Class may have none or several ObjectProperties, as shown in Fig 4. The visitor populates the following parameters for each ObjectProperty: a restriction type (`owl:AllValuesFrom`), an optional restriction cardinality (`owl:minCardinality`), a restriction number associated with the cardinality, a possible one or more characteristic of the property (`owl:TransitiveProperty`), and an optional expression (`owl:UnionOf`). An ObjectProperty associates the following concepts:

- Class-to-Class(es): one or several range classes depending on the restriction type and the expression. For example, a Building `contains Some` Floor.
- Class-to-Individual(s): one or several range individuals depending on the restriction type and the Expression. For example, a TemperatureMeasure Class `hasUnit` Degree_Celsius Individual.
- Individual-to-Individual: the restriction type, cardinality, expression, and characteristic are not populated by the visitor when an individual is associated to another individual. For example, a Building1 `contains` Floor1. Only the name parameter is used in the ObjectProperty for the code generation.

An Individual is an instance of a class. The visitor fills the namespace and the packageName for the code generator based on its IRI.

When a DataProperty is associated with an Individual, the visitor populates only the range (Literal). However, when the DataProperty is associated with a class, the visitor populates in addition to

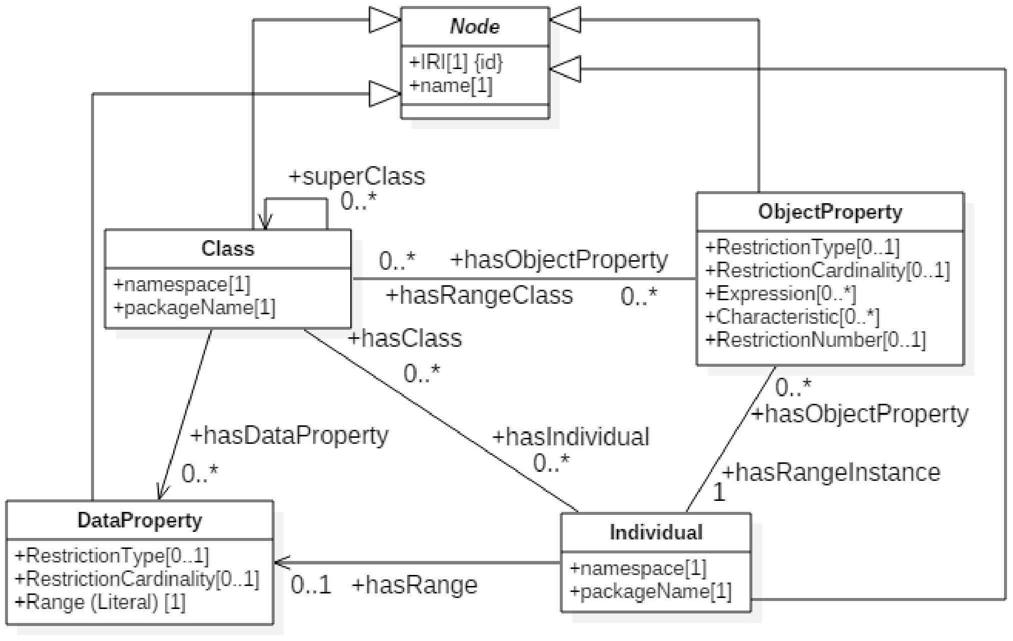

Figure 4: OLGA Internal Model - UML Object Diagram

the previous parameters, the restriction type and the restriction cardinality.

Templates: are arranged by library dependency, a serializer (Jackson-Jsonld [18], RDF4J [26]) or ORM (RomanticWeb [33], RDFAlchemy [35]). As shown in listing 1, each template contains a code snippet written according to a programming language syntax (Java, C#, Python, or others) and holds several information awaiting to be filled from the model such as the imports declaration of packages related to the dependent library, name of the class/interface to be generated. In addition, each of the data and object property will be transformed into a parameter with getter/setter functions. These templates will be loaded in memory by the Generator and the code snippets will be completed according to the information populated in the model.

Generator: based on the selected library dependency, adequate templates are loaded into memory to initiate the code generation from the populated model. The Generator injects the information from the model into the templates. The separation between the model and the templates provides flexibility and makes supporting an additional library a matter of templates extension.

In the case of a selected ORM dependency, each Class and Individual of the ontology will be generated into an interface. In fact, an ORM library provides a factory allowing developers to instantiate the interfaces into objects. OLGA handles multiple inheritance and composition by generating interfaces extending other interfaces. This makes the code generation simpler since only the interfaces need to be generated. Developers will rely on the factory to instantiate their objects based on the generated interfaces, therefore, the ORM's factory will handle multiple extensions of the interfaces and their declared functions.

In the case of a selected serializer dependency, interfaces and their implementations are generated since a serializer do not provide a factory to instantiate/implement a class. OLGA handles multiple inheritance and composition by propagating the functions' implementations from the extended interfaces into the implemented classes and by relying on the `Override` annotation in Java (or similar) annotations in other programming languages.

Compiler & Packager: once the code is injected into the templates, the generator creates files containing the expected code. Then, the compilation and packaging phase can start. According to the selected library and its programming language a compiler is loaded. Once the compilation ends successfully, the packaging is triggered to prepare the adequate format (.jar, .dll, .whl, or others).

Once the generated library is packaged, it can be imported and ready to be used by any IoT developer. An example of a SAREF generated library is provided in Listing. 2, where an IoT developer can *Just Instantiate and Link*, an Idegree_Celsius is refereed to, then an indoor measurement is created and linked to a specific instance of a Temperature Sensor.

The code implemented by the IoT developer can be deployed on a cloud connected device or gateway such as in [20, 21]. However, on systems with more abundant resources such as a Building Management Server, a generated library dependent on an object relational mapper can be used since such systems can host local applications which can interact with a local RDF store such as in [19].

6 IMPLEMENTATION & EVALUATION

OLGA is implemented in Java 8 and relies on the OWL API [25] to import and merge one or more interdependent ontologies. The visitor module relies on the OWL API as well to traverse the merged

Lib/Time	Overall	Code Gen.	Compile (ms)
RomanticWeb	14 sec 201 ms	516 ms	05 sec 317 ms
RDF4J	45 sec 109 ms	680 ms	39 sec 59 ms
Jackson-Jsonld	24 sec 187 ms	639 ms	19 sec 05 ms
RdfAlchemy	16 sec 544 ms	526 ms	9 sec 122 ms

Table 1: Code Generation Evaluation for SAREF by OLGA

ontology and populate the model shown in Fig. 4. The Generator depends on the FreeMarker [39] Template Engine and the templates are written in the FreeMarker Template Language. FreeMarker offers a powerful and flexible mechanism to inject parameters from our model into the templates to generate code files. The Compiler and Packager module relies on three Maven plugins depending on the selected programming language, .Net[4], Java[5], and Python[6]. In fact, OLGA generates the adequate pom.xml file which is then used with the correspondent Maven plugin to compile and package.

The list of supported dependent libraries is expected to grow with time based on the adoption and requests from our internal teams. So far, OLGA supports the java-based serializer Jackson-Jsonld [18], RDF4J [26], and the ORMs RomanticWeb [33] and RDFAlchemy [35].

The Smart Appliances REFerence ontology (SAREF) [5] was selected to validate the generated libraries. SAREF imports the Time Ontology[7]. The two ontologies are loaded and merged by OLGA, the merged ontology contains 117 classes, 63 Object Properties, 31 Data Properties, and 55 Individuals. Four libraries are generated based on SAREF with a dependency to Jackson-Jsonld, RDF4J, RDFAlchemy, and RomanticWeb. For each of the generated library, the following is provided on the external github [31]:

(1) Packaged generated library in .dll and .jar formats.
(2) Generated source code in C#, Java, and Python.
(3) Generated ontology instance (A-Box) of an instantiation SAREF example.
(4) An instantiation and usage examples for each of the generated SAREF libraries (Jackson-Jsonld, RDF4J, RDFAlchemy, and RomanticWeb). The examples demonstrate how any IoT developer can import the generated packages and use them in his development without any knowledge regarding ontologies. The provided examples show the instantiation of a SAREF temperature sensor[8] with a measurement temperature in degree Celsius, and other information such as the manufacturer and the model number, as shown in listing. 2.

OLGA is deployed on a 64 bit windows machine with an I7 Code and 32 GB of RAM with a JVM of 512 MB of maximum heap size. Table 1 depicts the code generation evaluation for SAREF with a dependency on RomanticWeb, RDFAlchemy, RDF4J, and on Jackson-Jsonld where the values represent an average of 10 executions. The table depicts the overall generation time from import until packaging. It also details the code generation, compilation, and packaging time. The overall time for a serializer is longer than an

```
public static void Create_SAREF_Topology()
{
    string clientURI = "http://www.saref.instance/example";

    //refer to the unit
    Idegree_Celsius degreeCelius = context
     .Create<Idegree_Celsius>(new Uri(clientURI + "#1"));

    //Create a measurement from the factory
    IMeasurement indoorTemperature = context
     .Create<IMeasurement>(new Uri(clientURI + "#2"));
    indoorTemperature.AddIsmeasuredin_Only_UnitOfMeasure
     .Add(degreeCelius);
    indoorTemperature.Hasvalue = 32;
    indoorTemperature.Hastimestamp = DateTime.UtcNow;

    //Link it to Temperature
    ITemperature temperature = context
     .Create<ITemperature>(new Uri(clientURI + "#3"));
    temperature.AddRelatestomeasurement_Only_Measurement
     .Add(indoorTemperature);

    //Create a Temperature Sensor
    ITemperatureSensor temperatureSensor = context
     .Create<ITemperatureSensor>(new Uri(clientURI + "#4"));
    temperatureSensor.Hasmanufacturer = "CompanyA";
    temperatureSensor.Hasmodel = "M321";
    temperatureSensor
     .Hasdescription = "Low range Zigee temperature sensor";

    //add its measurement
    temperatureSensor.AddMakesmeasurement_Only_Measurement
     .Add(indoorTemperature);

    // commit data to factory
    context.Commit();
}
```

Listing 2: Example of SAREF instantiation code [31] in C# by an IoT developer

ORM since for a serializer OLGA generates interfaces and their implementation classes while for an ORM only the interfaces are needed, the instantiation goes through the factory. In the case of the RDF4J, the compilation and packaging phase takes longer since it pulls several RDF4J dependencies. Package optimization will be part of our future work.

7 CONCLUSION

In this paper, we presented a model driven methodology which relies on the separation of concern between two aspects. The first aspect is the model creation and its instantiation. The second aspect is the distinction between the three different actors and their skills set in the IoT development. We proposed a methodology which highlights and leverages the expertise of each actor and places it in its adequate posture in order to accelerate the ontology-based IoT application developments.

In addition, based on the requirements gathered from our industrial context and internal development teams, we propose OLGA, an Ontology Library Generator which completes existing libraries and frameworks in the ontology development arena. OLGA is an enabler and a facilitator for the adoption of libraries and frameworks depicted in the related work section. It complements and does not competes with the effort put by the ontology community regarding the software tools suite. OLGA leverages this such software suite by automatically providing a bridge towards its usage in an abstract

[4]github.com/kaspersorensen/dotnet-maven-plugin
[5]maven-compiler-plugin on mvnrepository.com
[6]http://www.mojohaus.org/exec-maven-plugin
[7]www.w3.org/TR/owl-time
[8]$ontology.tno.nl/saref/saref_TemperatureSensor.html$

and more adequate manner for IoT developers. OLGA takes an ontology previously defined by the domain and ontology experts and generates a library hiding all the complexity of ontology based development. We validated OLGA by relying on the existing Smart Appliances REFerence Ontology [5], the generated libraries depend on four existing libraries in three different programming languages and paradigms. Such flexibility and easy of extensions is possible thanks to OLGA's internal model which captures the ontology information independent of any targeted library or programming language. Separating the model from any implementation offers OLGA a huge flexibility making supporting another language or library a matter of adding templates.

For now, OLGA is in its minimum viable product and will evolve over time based on our internal teams requirements. In our future work, we intend to extend the list of the dependent libraries to support the diversity of our internal teams' technical choices and environments. In addition, the query formulation part depicted earlier, might be partially solved by LINQ for .Net developers, however, it still needs to be tackled for other programming environments. More over, constraints handling from the ontology model to the generated code will be part of our next steps.

REFERENCES

[1] Balaji Bharathan et al. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments, BuildSys*. ACM, 2016.
[2] Brochure. Ecostruxure. http://www.schneider-electric.com/en/download/document/998-19885906_GMA-US_ESX/, Online; accessed 9-July-2017.
[3] V. Charpenay, S. Käbisch, D. Anicic, and H. Kosch. An ontology design pattern for iot device tagging systems. In *2015 5th International Conference on the Internet of Things*, 2015.
[4] Fabien Chevalier. Autordf - using owl as an object graph mapping (ogm) specification language. In *The Semantic Web: ESWC 2016 Satellite Events*. Springer, 2016.
[5] Laura Daniele, Frank den Hartog, and Jasper Roes. Created in close interaction with the industry: the smart appliances reference (saref) ontology. In *International Workshop Formal Ontologies Meet Industries*, 2015.
[6] DB. Stardog. http://www.stardog.com, Online; accessed 9-July-2017.
[7] Editor. Protege. http://protege.stanford.edu, Online; accessed 9-July-2017.
[8] EF. Entity framework. https://github.com/aspnet/EntityFramework6, Online; accessed 9-July-2017.
[9] Charbel El Kaed and Matthieu Boujonnier. Forte: A federated ontology and timeseries query engine. In *The 3rd IEEE International Conference on Smart Data*. IEEE, 2017.
[10] EMF. Eclipse modeling framework. eclipse.org/emf, Online; accessed 9-July-2017.
[11] Prudhommeaux Eric and Seaborne Andy. Sparql query language for rdf. www.w3.org/TR/rdfsparql-query, 2004. SPARQL Query Language for RDF, W3C.
[12] D. Pfisterer et al. Spitfire: toward a semantic web of things. *IEEE Communications Magazine*, 2011.
[13] Mariano Ferndndez and Natalia Juristo. Methontology: From ontological art towards ontological engineering. In *In Proceedings of the AAAI Spring Symposium*, 1997.
[14] A. Gyrard, S. K. Datta, C. Bonnet, and K. Boudaoud. Cross-domain internet of things application development: M3 framework and evaluation. In *3rd International Conference on Future Internet of Things and Cloud*, 2015.
[15] Kagermann Henning, Wahlster Wolfgang, and Helbig Johannes. Recommendations for implementing the strategic initiative industrie 4.0, 2013.
[16] JPA. Java persistence api. http://www.oracle.com/technetwork/articles/javase/persistenceapi-135534.html, 2007.
[17] JPA. Empire. https://github.com/mhgrove/Empire, Online; accessed 9-July-2017.
[18] jsonld. Jackson-jsonld. https://github.com/io-informatics/jackson-jsonld, Online; accessed 9-July-2017.
[19] C. E. Kaed, B. Leida, and T. Gray. Building management insights driven by a multi-system semantic representation approach. In *2016 IEEE 3rd World Forum on Internet of Things*, 2016.
[20] C. El Kaed, Y. Denneulin, and F. G. Ottogalli. Dynamic service adaptation for plug and play device interoperability. In *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011.
[21] Charbel El Kaed, Imran Khan, Hicham Hossayni, and Philippe Nappey. SQenIoT: Semantic Query Engine for Industrial Internet-Of-Things Gateways. In *IEEE 3rd World Forum on Internet of Things*, 2016.
[22] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A Padget. Automatic mapping of owl ontologies into java. In *SEKE*, 2004.
[23] Wenzel Ken. Ontology-driven application architectures with komma. In *7th International Workshop on Semantic Web Enabled Software Engineering*. Springer, 2011.
[24] Evgeny Kharlamov et al. How semantic technologies can enhance data access at siemens energy. In *The International Semantic Web Conference*. Springer, 2014.
[25] Lib. Owl api. https://github.com/owlcs/owlapi/wiki, Online; accessed 9-July-2017.
[26] Lib. Rdf4j. http://rdf4j.org, Online; accessed 9-July-2017.
[27] Lib. Rdflib. https://rdflib.readthedocs.io, Online; accessed 9-July-2017.
[28] Lib. Redland. librdf.org, Online; accessed 9-July-2017.
[29] Lib. Alibaba. https://bitbucket.org/openrdf/alibaba, Online; accessed 9-July-2017.
[30] Lib. dotnetrdf. http://www.dotnetrdf.org, Online; accessed 9-July-2017.
[31] OLGA. Saref generated by olga. https://github.com/InnovationSE/SAREF-Generated-By-OLGA, Online; accessed 9-July-2017.
[32] Ontology. Saref. http://ontology.tno.nl/saref, Online; accessed 9-July-2017.
[33] ORM. Romanticweb. http://romanticweb.net, Online; accessed 9-July-2017.
[34] Protege. Code generator plugin. https://github.com/protegeproject/code-generation, Online; accessed 9-July-2017.
[35] python. Rdfalchemy. http://rdfalchemy.readthedocs.io, Online; accessed 9-July-2017.
[36] RDF4J. Rdf4j schema generator. https://github.com/ansell/rdf4j-schema-generator, Online; accessed 9-July-2017.
[37] SchneiderElectric. Internal workshop. https://github.com/InnovationSE/Ontology-Workshop, Online; accessed 9-July-2017.
[38] Bechhofer Sean, van Harmelen Frank, Hendler Jim, Horrocks Ian, McGuinness Deborah, Patel-Schneider Peter, and Andrea Stein. Web ontology language. http://www.w3.org/TR/owl-features/, 2004.
[39] Template Engine. Freemarker. http://freemarker.org, Online; accessed 9-July-2017.
[40] Trinity. Semiodesk. https://bitbucket.org/semiodesk/trinity, Online; accessed 9-July-2017.
[41] VSB. Visual query builder. https://leipert.github.io/vsb, Online; accessed 9-July-2017.
[42] W3C. Resource description framework. http://www.w3.org/RDF, 1999.