Towards a Semantics Extractor for Interoperability of IoT Platforms

Wenbin Li
Easy Global Market
Sophia Antipolis, France
wenbin.li@eglobalmark.com

Gilles Privat
Orange Labs
Grenoble, France
gilles.privat@orange.com

Franck Le Gall
Easy Global Market
Sophia Antipolis, France
franck.le-gall@eglobalmark.com

***Abstract*—Achieving the interoperability of Internet of Things (IoT) platforms raises a key challenge, as most current IoT applications are vertically integrated within separate silos without horizontal communications. In response to this, we propose the Semantics Extractor, a framework for automatic extraction of semantic graphs from REST interfaces exposed by IoT platforms. Through an iterative process of extracting, identifying, inferring, enriching and refining, our framework is able to discover IoT resources, add semantics to platforms' payloads and capture latent links within and between resources exposed by IoT platforms, from which further semantics can be deduced. We show via examples how RDF graphs are incrementally generated and enriched by such iteration process to interoperate IoT platforms. At last we present an implementation architecture.**

***Index Terms*—Internet of Things, Web of Things, semantic interoperability, Linked Data, RDF, REST, HATEOAS.**

I. Introduction

IoT platforms are the operating essentials to, mediate information between applications and physical IoT environments and to manage the corresponding system of connected devices. Different IoT platforms, such as FIWARE [1] and openHAB [2], have been proposed with their own specifications of data structure and management operations. Nevertheless, they are mostly vertically integrated with their own applications and locked in separate silos. In such case, different IoT platforms cannot directly interact with each other and obtaining global information across different IoT platforms requires too much effort. So far, achieving semantic interoperability between IoT platforms has become a main challenge. The semantic interoperability of IoT platforms is the capability of information mediated by these IoT platforms to be "understood", which does not only include the communication, but also the automatic interpretation of information between IoT platforms and applications. To achieve semantic interoperability is a challenging task due to the heterogeneity of IoT objects (not only sensors and actuators), the variety of data models, the implicitness of resource descriptions, the lack of accessibility of IoT platforms [3].

Web of Things (WoT) emerges as a combination of IoT and Web to take advantage of the universal accessibility of the Web and make a truly open Internet of Things [4]. To achieve the transformation from IoT to WoT, most IoT platforms adopt Representational State Transfer (REST) as architectural style [5]. As a first step to achieve semantic interoperability, WoT, by combining IoT and REST, provides existing platforms with uniform operations for resource manipulation, which in a way manages the heterogeneity of IoT objects and improves the platforms' accessibility. However, semantic information is missing in resource data payloads, and IoT platforms are still separated as independent silos due to the variety of data models and the implicitness of resource description. Connecting IoT platforms via semantics is the key to achieve their interoperability.

In response to this, we propose the IoT Semantics Extractor: a framework for automatic generation of semantic graphs by discovering IoT resources from REST interfaces, annotating resource descriptions and capturing implicit links between IoT resources.

The remainder of this paper is organized as follows: section 2 presents the general framework of Semantics Extractor and section 3 introduces its modules. A reference implementation is presented in section 4 and related works are illustrated in section 5. Section 6 concludes our work and shed light on future perspectives.

II. Semantics Extractor Framework

Taking IoT platforms' REST interfaces as input, Semantics Extractor iteratively generates a semantically-rich RDF graph. An overview of the framework is presented in Fig. 1.

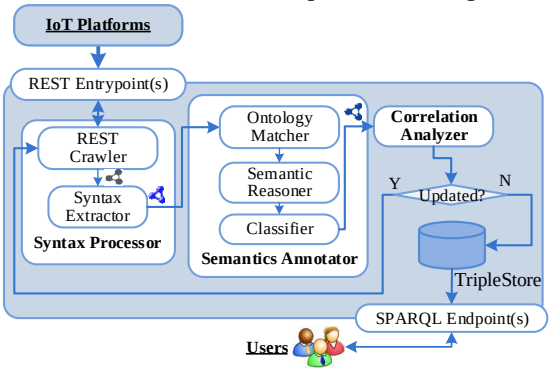

Fig. 1. Semantics Extractor Overview

Generally, the framework consists of three parts (i.e., Syntax Processor, Semantics Annotator and Correlation Analyzer) with six modules that iteratively run through cycles.

Syntax processor aims at generating a preliminary RDF graph by discovering REST resources and extracting syntax information from resource descriptions; semantics annotator semantically identifies and infers the nature of graph nodes and arcs (i.e., subjects, objects and predicates) by associating them with ontology elements (i.e., classes and properties); correlation analyzer captures potential links among IoT resources and platforms by analyzing data patterns.

Each internal module runs in sequence; at the end of one cycle, the generated RDF graph is sent to syntax processor to start a new cycle for two objectives: firstly, certain RDF graph elements deduced by later modules can possibly be used by previous modules in a later iteration to identify new graph elements and capture new relations; secondly syntax processor updates RDF graph if resources' descriptions have changed during the previous cycle. The RDF graph automatically and incrementally grows through these iterations. The iteration stops when the graph has not been updated from the previous iteration. The generated RDF graph, as the semantic descriptions of IoT resources, is the output stored in a triplestore providing a SPARQL endpoint for users' query. Moreover, IoT semantics extractor is able to resume the iteration as soon as new information is obtained by any means (such as via monitoring and notification) about any graph elements, and correspondently update the RDF graph.

In order to illustrate the idea of Semantics Extractor, we introduce an example of smart building, in which two separate rooms are connected by a corridor in the same floor. Room 1 is equipped with a presence sensor and an electrical door lock; room 2 is equipped with a door sensor and a presence sensor; the corridor is equipped with a presence sensor.

The three building entities use three different IoT platforms due to their different proprietors. Room 1 is based on Datavenue [6]; room 2 is based on FIWARE [1]; corridor is based on openHAB [2]. This typical IoT structure has two main problems: three IoT platforms cannot communicate with each other as different data formats and APIs are used; global information regarding the whole floor is difficult to obtain, and for instance in case of fire, users have to separately request each IoT platform to get the presence state of the whole floor.

The three distinct platforms all expose REST URIs as interfaces for resources descriptions. At the very beginning, we only get three separated URIs as three entry points without any further information. In the following, we present via examples how Semantics Extractor incrementally generates a semantically-rich and inter-connected RDF graph from the IoT REST interfaces. Due to the page limitation, a more detailed graphical presentation of this example is presented in [7]

III. Semantics Extractor Modules

We introduce the following notations for terminologies used throughout the paper.

Notation 1. Ontology. According to [8], an ontology O contains a core ontology S, axioms A, a knowledge base KB, and a lexicon Lex, and is defined as a 4-tuple $O:=(S, A, KB, Lex)$. Furthermore, a core ontology S is defined as a 5-tuple,

$$S:=(C, \leq C, R, \sigma, \leq R),$$

where C and R are two disjoint sets whose elements are called concepts c (a.k.a. classes) and relations r (a.k.a. properties), $c \in C$, $r \in R$; $\leq C$ and $\leq R$ are two partial orders on C and R named concept hierarchy and relation hierarchy; $\sigma: R \rightarrow C \times C$ is a function where $\sigma(r)=<dom(r), ran(r)>$ with $r \in R$, the domain or r is $dom(r)$, and the range of r is $ran(r)$.

In this paper, we use Ont to denote the set of ontologies used in IoT Semantics Extractor $Ont = \{O_1, O_2, \ldots, O_n\}$, and c and r to refer to a concept and a relation from the ontologies, $c \in Ont.S.C$, $r \in Ont.S.R$.

Notation 2. RDF. A RDF description is defined as a directed graph with the node set N and the predicate set P,

$$RDF:=(N, P),\ n \in N,\ p \in P,\ P=N \times N$$

A RDF statement is denoted as $stmt$ and is defined as a triple $stmt=<s, p, o>$, where s, p, and o respectively denotes the subject, predicate and object of the statement, $s \in N$, $p \in P$, $o \in N$.

A. REST Crawler

REST crawler works as a Web crawler to automatically discover REST resources following input REST entry point(s), and generates a URI graph. According to the HATEOAS principle, REST resources descriptions must have well-defined ways in which they expose links to related resources [5], and therefore REST clients are allowed to use resource identifiers and a media-type discovery process for discovery of services. A number of formats support HATEOAS in REST design, which can be generally divided into two categories: general serialization formats which define a link by a simple target URI denoted as *link.uri*; specifications which define links by the combination of *link.uri* and a relation type specifying the nature of this link, denoted as *link.rel*, such as Linker Header [9] and CoRE Link [10]. Discovery strategies for different formats are introduced in [11] and are out of our discussion scope.

REST crawler applies a recursive process of identifying the format of resources via media type description (e.g., HTTP header), extracting links from resource descriptions and generating the corresponding RDF graph. The process is formalized in Fig. 2. Fig. 3 presents the output example.

B. Syntax Extractor

In the first cycle, syntax extractor extracts information from discovered resources' descriptions and creates a stub RDF graph; from the second cycle the module updates existing RDF graph if resource descriptions change during iterations.

Several ways exist to extract information from serialization formats to construct a graph, and here we present a recursive method to generate a RDF graph from JSON-based resource description. Other serialization formats are firstly transformed to JSON in our framework and then are processed. JSON is built on two structures, i.e., a collection of key/value pairs and an array. The extraction process is introduced as follows:

1) A first subject is generated by use of the JSON-based resource URI;

2) for a collection of key/value pairs, the keys are used to generate predicates in RDF graph while the values are regarded as the objects. In case that the value of a key k is another JSON object obj instead of a simple data type, a dummy anonymous

node *anode* is created as the object of the key *k*, and *anode* is the subject of the key/value pair of *obj*; for an array of values, the predicate is regarded as "rdf: predicate", while the value elements in array are RDF objects.

Input. E: the set of REST entrypoints. **Output.** RDF
Step 1. Link Extraction
  for *e∈E* **do**
    RDF.N = RDF.N∪{e}
    e.description = getDescription(e)
    e.format = getFormat(e.description)
    L = extractLinks(e.format, e.description)
    (▷ *L* is the set of the extracted links)
Step 2. Graph Generation
  for *link∈L* **do**
    if *(link.rel≠null)∧ (link.uri≠e)*
      then *RDF=RDF.addStmt(<e, link.rel, link.uri>)*
    else if *(link.rel==null)∧ (link.uri≠e)*
  then *RDF=RDF.addStmt(<e, rdf:predicate, link.uri>)*
    recuseLinkExtraction(link)

Fig. 2. REST crawling process

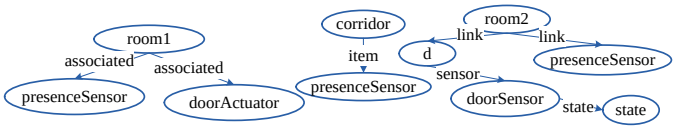

Fig. 3. REST crawling result

4) An elimination algorithm introduced in [12] is applied to reduce the redundancy of the RDF graph;

5) at the end of extraction, every RDF resource is associated with the class "rdfs:Resource", and every literal is associated with the class "rdfs:Literal".

From the second cycle, syntax extractor updates the RDF graph if changes have been notified/detected during iterations. Syntax extractor records the resource changes, and stores the information in a global table denoted as *T* with each change information *ch* defined as a 4-tuple *ch = <uri, timestamp, key, value>*, *ch∈T*, whereas *uri*, *timestamp*, *key* and *value* are respectively the changed resource's URI, change timestamp, the changed key and value. Many technologies support REST server push such as Server-Sent Event [13] and CoAP [14], which are out of our discussion scope. The process of syntax extractor is formalized in Fig. 4.

Fig. 5 illustrates the generated RDF graph where three sub graphs represent three IoT platforms in our example. To keep the representation as short and illustrative as possible, we only illustrate RDF nodes that are not blank nor literal.

C. Ontology Matcher

Ontology Matcher identifies RDF graph elements based on semantic keyword matching. In this module, the local names of RDF resources and descriptions of RDF predicates are matched against ontologies' classes and properties respectively. We adopt the semantic matching algorithm introduced in [15] due to its performance on heterogeneities and inconsistencies matching. The notation "a≡^{m}b" denotes that a graph element *a* successfully matches with an ontology element *b*. The ontology matching process is formalized in Fig. 6. The ontology alignment algorithm in [16] is applied to deal with multiple matchings between one graph element and ontology elements.

Input. RDF. **Output.** RDF
Cycle 1. Graph Extraction
  for *n∈RDF.N* **do**
    n.description = getDescription(n)
    RDF = extractDescriptions(n.description, JSON)
  for *n∈RDF.N* **do**
    if *(n.hasURI())*
      then *RDF=RDF.addStmt(<n, rdf:type, rdfs:Resource>)*
    else *RDF=RDF.addStmt(<n, rdf:type, rdfs:Literal>)*
Cycle i (i>1). Graph Update
  if *∃ch∈T, (ch.timestamp>cycle$_{i-2}$) ∧ (ch.timestamp<cycle$_{i}$)*
    then *RDF = RDF.updateDescriptions(ch, JSON)*

Fig. 4. Syntax extracting process

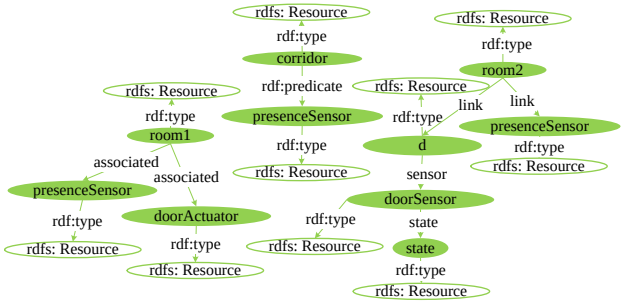

Fig. 5. Syntax extractor output

Input. RDF. **Output.** RDF
Step 1. Class Matching
  for *n∈RDF.N* **do**
    if *∃stmt, ∃c∈Ont.S.C, (n.hasURI())∧*
    (stmt==<n, rdf:type, rdfs:Resource>)∧(n≡^{m}c)
      then *RDF=RDF.removeStmt(stmt)*
        RDF=RDF.addStmt(<n, rdf:type, c>)
Step 2. Property Matching
  for *p∈RDF.P* **do**
    if *∃r∈Ont.S.R, ¬(p.hasNamespace())∧(p≡^{m}r)*
      then *RDF = RDF.replace(p, r)*
      (▷ replace predicate *p* by relation *r)*
    else if *∀r∈Ont.S.R, ¬(p.hasNamespace())∧¬(p≡^{m}r)*
      then RDF = RDF.*replace(p, rdf:predicate)*

Fig. 6. Ontology matching process

In our example, resources "room1", "presenceSensor" and "doorActuator" are respectively identified by "dogont:Room", "dogont:PresenceSensor" and "dogont:DoorActuator". This subgraph will be further inferred by semantic reasoner. The DogOnt ontology is presented in [17].

D. Semantic Reasoner

Semantic Reasoner first infers the classes and properties in RDF graphs based on ontology property's domain and range; and then it enriches RDF graph by adding additional predicates;

at last it refines RDF elements by following semantic rules. The inference process of semantic reasoner is presented in Fig. 7. The ontology alignment algorithm in [16] is again applied to deal with the case that more than one ontology elements are inferred for the same graph element.

Input. RDF. **Output.** RDF
Step 1. Class Inferring
if $\exists stmt, \exists r \in Ont.S.R, (stmt.p==r) \wedge (dom(r)\neq null) \wedge (ran(r)\neq null)$
then $RDF=RDF.addStmt(<stmt.s, rdf:type, dom(r)>)$
$RDF=RDF.addStmt(<stmt.o, rdf:type, ran(r)>)$
Step 2. Property Inferring
if $\exists stmt, \exists r \in Ont.S.R, (rdf:type(stmt.s,ran(r)) \wedge (rdf:type(stmt.o,dom(r)) \wedge (rdfs:subClassOf(r, stmt.p))$
then $RDF=RDF.addStmt(<stmt.s, r, stmt.o>)$
$RDF=RDF.removeStmt(stmt)$

Fig. 7. Inference process of semantic reasoner

In our example, after ontology matching, the predicate between the subject "room1" and the object "presenceSensor" remains unidentified, we can infer the predicate is "dogont:hasSensor.

The enriching process of semantic reasoner is formalized in Fig 8 and is based on the five characteristics of ontology properties specified in OWL as follows, where *p* is an ontology property, *x*, *y*, and *z* are three RDF nodes.

1) TransitiveProperty: p(x, y) ∧ p(y, z) => p(x, z);
2) SymmetricProperty: p(x, y) => p(y, x);
3) FunctionalProperty : p(x,y) ∧ p(x,z) => y==z;
4) InverseOf : p1(x,y) => p2(y, x);
5) InverseFunctionalProperty: p(y,x) ∧ p(z,x) => y==z.

Input. RDF. **Output**. RDF
Step 1. Transitive Property Enrichment
if $\exists stmt, \exists n_1, n_2, n_3 \in RDF.N, \exists p \in RDF.P, p(n_1, n_2) \wedge p(n_2, n_3) \wedge (stmt==<p, rdf:type, owl:TransitiveProperty>)$
then $RDF=RDF.addStmt(<n_1, p, n_3>)$
Step 2. Symmetric Property Enrichment
if $\exists stmt, \exists n_1, n_2 \in RDF.N, \exists p \in RDF.P, p(n_1, n_2) \wedge (stmt==<p, rdf:type, owl:SymmetricProperty>)$
then $RDF=RDF.addStmt(<n_2, p, n_1>)$
Step 3. Functional Property Enrichment
if $\exists stmt, \exists n_1, n_2, n_3 \in RDF.N, \exists p \in RDF.P, p(n_1, n_2) \wedge p(n_1, n_3) \wedge (stmt==<p, rdf:type, owl:FunctionalProperty >)$
then $RDF=RDF.addStmt(<n_2, owl{:}sameAs, n_3>)$
Step 4. InverseOf Enrichment
if $\exists stmt, \exists n_1, n_2 \in RDF.N, \exists p_1, p_2 \in RDF.P, (stmt==<p_1, owl:inverseOf, p_2>) \wedge p_1(n_1, n_2)$
then $RDF=RDF.addStmt(<n_2, p_2, n_1>)$
Step 5. Inverse Functional Property Enrichment
if $\exists stmt, \exists n_1, n_2, n_3 \in RDF.N, \exists p \in RDF.P, p(n_2, n_1) \wedge p(n_3, n_1) \wedge (stmt==<p, rdf:type, owl:owl:InverseFunctionalProperty>)$
then $RDF=RDF.addStmt(<n_2, owl{:}sameAs, n_3>)$

Fig. 8. Enriching process of semantic reasoner

In our example, a statement *stmt1* exists, where *stmt1 = <room1, dogont:hasSensor, presenceSensor>*. As the property "*dogont:hasSensor*" is *InverseOf* the property "*dogont:sensorOf*", semantic reasoner adds another statement to the RDF graph: *stmt2 = <presenceSensor, dogont:sensorOf, room1>*. The purpose of the enrichment is to provide a RDF graph as rich as possible to for reasoning and classifications.

The refining process of semantic reasoner refines RDF elements following semantic rules. A semantic rule *sr* has the form of an implication between an antecedent *ant* and consequent *con*, defined as *sr:= (ant=>con), sr∈SR* where *SR* is the set of semantic rules. SR in our framework comes from two ways: 1) axioms *A* in ontology definitions 2) patterns automatically extracted from linked data by mining algorithm introduced in [18] and then transformed to semantic rules. The semantic reasoning process is formalized in Fig. 9.

Input. RDF. **Output**. RDF
if $\exists sr \in SR, sr=(ant=>con), \exists subg, subg==ant$
($\triangleright$ *subg* is a subgraph of RDF graph)
then $RDF=RDF.addSubgraph(con)$

Fig. 9. Refining process of semantic reasoner

For example, a RDF resource "d" is an instance of ontology class "rdfs:Resource", and a statement $stmt_1$ exists in the graph, where *stmt1 = <d, dogont:hasSensor, doorSensor>*. A semantic rule exists as *sr=(dogont:hasSensor(x, dogont:DoorSensor)=>rdf:type(x, dogont:Door))*, by which semantic reasoner refines that the resource "d" is an instance of ontology class "dogont:Door".

E. Classifier

When no explicit rule is available for semantic reasoning, classifier analyzes graph elements' relations with other elements and uses pre-trained classifier to refine RDF element. We adopt the ontology classification algorithm introduced in [19] to create the classifier and use Maximum Entropy Markov Model as classification model due to the optimum performance for class induction. The training process to create classification model in design time consists of three steps: a) preprocessing, b) feature extracting and c) classifier training; when given data to classify, the classifying process based on the classification model in runtime consists of three steps: a) preprocessing, b) feature extracting and c) classifying to refine graph elements.

For example, one relation "skos:related" is created between corridor and two rooms by correlation analyzer (introduced in next section). The classifier can further refine this relation to "biotop:physicallyConnectedTo". The SKOS and S4EE ontologies are defined in [20] and [21].

F. Correlation Analyzer

Correlation Analyzer adds additional implicit links between IoT resources by capturing data change correlations. Data change correlation refers to the pattern among two or more resources: when the state of a resource changes, within a following time period *tp*, the state of one or more other resources also change. When such a correlation between two resources has been observed more than a number of times *tt*, we claim that a correlation pattern is detected, denoted as *ptn*. Correlation Analyzer deduces that a latent relationship exists between the two corresponding resources, and a relation "biotop:physicallyConnectedTo" is created between the related resources. The process is formalized in Fig. 10.

Input. RDF, T. **Output**. RDF
Step 1. Coincidence Detection
for $ch_i \in T$, $ch_j \in T$, $i \neq j$ **do**
if $(ch_i.timestamp - ch_j.timestamp < tp)$
then $ptn_{ij} = <ch_i.uri, ch_j.uri>$
$PTN = PTN \cup \{ptn_{ij}\}$
($\triangleright$ *PTN* is the repeatable set of change patterns)
Step 2. Relation Generation
if $\exists n_1 \in RDF.N, \exists n_2 \in RDF.N, (PTN.countPattern(n_1, n_2) > tt)$
then $RDF = RDF.addStmt(<n_1, skos:related, n_2>)$

Fig. 10. Link extracting process

In our example, we observe that each time the presence state of "room1" changes, the presence state of "corridor" changes; and each time the presence state of "room2" changes, the presence state of "corridor" changes. We then create two relations "skos:related" between "room1" and "corridor" and between "room2" and "corridor". This relation is further refined by the classifier.

IV. Implementation Architecture

In this section, we present a reference implementation architecture comprising our framework modules in Fig. 11.

REST crawler is developed for JSON based REST interfaces by use of Restlet. Syntax extractor is implemented via GSON to extract syntax from JSON, and via Restlet to create a REST server to record changes based on Server-Sent Events. In smart building domain, the value of tp is set as 2 minutes and the value of tt is 10 times. All RDF manipulating operations are achieved by Jena; the mining algorithm of extracting semantic rules from linked data and classifier are realized by Spark. Jena Fuski is used as the triple store for RDF graphs management and SPARQL query endpoint.

Fig. 12 illustrates the Semantics Extractor output of our example, and the used ontologies are from Linked Open Vocabularies [22]. Comparing to their previous isolation in three separated IoT platforms, resources in this RDF graph are linked via semantics to support internal communications and global query from a unified interface such as SPARQL.

V. Related Works

To achieve semantic interoperability, most IoT platforms adopt REST as architectural style to provide standard operations for resources. Here we survey the related work from data model aspect dealing with the challenges of data model variety and resource description implicitness. Related work is reported and compared in surveys on metadata interoperability [23], interoperability for traditional SOA based systems [24] and interoperability for cloud based infrastructures [25], in which solutions to improve semantic interoperability can be generally divided into four categories as follows.

Standards and Ontologies. Standards and ontologies provide reference data models for IoT platforms. For examples, GSMA [26] defines a declarative data model that describes IoT objects with RESTful interfaces to manage information; OneM2M ontology [27] constitutes a basis framework for specifying the semantics of data that are handled in the deployment of Machine-to-Machine applications. The models provide a good reference model for new IoT platforms but alone they do not solve interoperability problem.

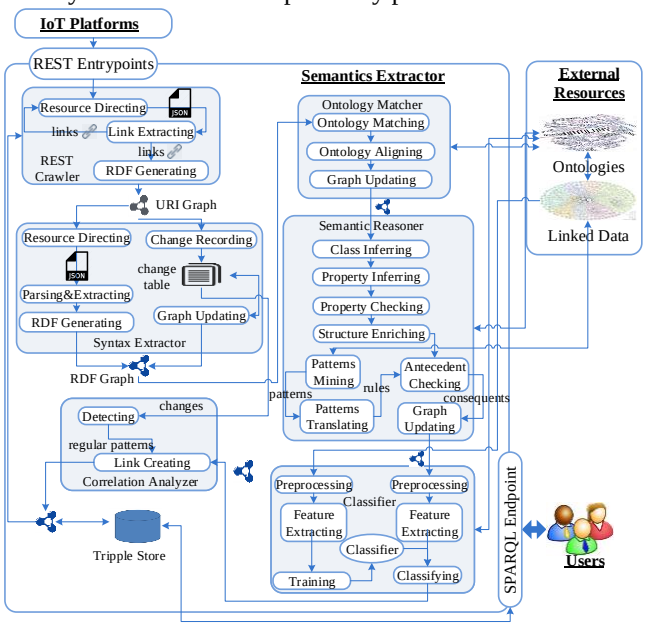

Fig. 11. Implementation Architecture

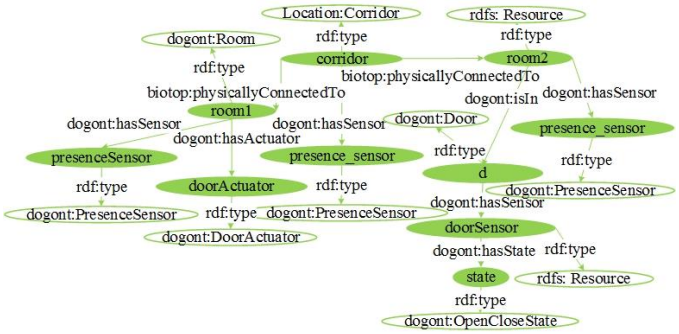

Fig. 12. RDF graph generated by Semantics Extractor

Mapping Technologies for Data Models. Mapping technologies establish mapped relations between data models to determine information similarity under different syntax and structures. A mapping model is introduced in [28] to establish interoperability between linguistic and terminological ontologies, which provide simple solution to connect ontologies. However, mapping relations need to be individually specified between models, which makes that vertical platforms are only interconnected within a bigger silo without flexibility to connect with external ones.

Data Integration and Data Exchange Systems. The data integration system [29] provides mechanisms for querying heterogeneous sources in a uniform way; while the data exchange system [30] restructures the data from the sources according to a global schema. Nevertheless, data integration and exchange systems require reconfigurations to take into account new data models every time.

Semantic Annotations. As one of the most popular solutions to achieve semantic interoperability, semantic

annotation updates existing data models by providing semantic labels to model elements. The survey [31] classifies semantic annotation approaches from the perspective of structural complexity, vocabulary type, and collaboration type. An approach is presented in [32] for automatic annotation of REST interfaces using a cross-domain ontology, which provides a good reference for annotating IoT infrastructures. However, the limitation of semantic annotation approaches is that they fail to capture implicit relations among resources and platforms.

Existing solutions are not rich enough to fully interoperate different IoT platforms; alternatively, our framework aims at automating all of resources discovery, semantic annotation and links extraction process to further interoperate IoT platforms.

VI. Conclusion

In this paper, we have presented the Semantics Extractor, a framework for automatically generating semantically-rich RDF graphs from IoT REST interfaces. Our framework provides functions of generating graphs from syntax data, semantically annotating graph elements and deducing additional relations among IoT resources and platforms. In the end, IoT platforms and the resources they expose can get interlinked with enriched semantics to support federated queries, data exchange and knowledge discovery.

Although we illustrate our framework via IoT REST interfaces, the processes could also be applied to other IoT interfaces such as SOAP and open databases, and the framework can be implemented on either cloud or edge side. As future work, we are investigating solutions, for open IoT platforms, to add semantics back to source platforms and update platform-side descriptions with semantics for good; we are also improving the effectiveness of correlation analyzer module by adding other correlation patterns to discover links.

Acknowledgment

Part of this work has been supported by the European Union's Horizon 2020 Research and Innovation Programme within the project WISE-IoT under grant agreement No. 723156.

References

[1] Connection to the Internet of Things FIWARE, https://www.fiware.org/devguides/connection-to-the-internet-of-things/

[2] OpenHAB, https://my.openhab.org/docs

[3] W. Li and G. Privat, "Cross-Fertilizing Data through Web of Things APIs with JSON-LD," in *Services and Applications over Linked APIs and Data*, ESWC2016 workshop, 2016.

[4] D. D. Guinard and V. M. Trifa, Building the Web of Things Book | Web of Things. Manning Publications, 2015.

[5] L. Richardson and S. Ruby, *RESTful web services*. Farnham: O'Reilly, 2007.

[6] Datavenue, https://datavenue.orange.com/live-objects-api

[7] WoTRDF_SemanticsExtractor, https://github.com/WoTRDF/SemanticsExtractor

[8] J. Cardoso and A. Sheth, Eds., *Semantic Web Services, Processes and Applications*. New York, NY: Springer, 2006.

[9] Web Linking, https://tools.ietf.org/html/rfc5988

[10] Constrained RESTful Environments (CoRE) Link Format, https://tools.ietf.org/html/rfc6690

[11] S. Mayer and D. Guinard, "An Extensible Discovery Service for Smart Things," in *Proceedings of the Second International Workshop on Web of Things*, 2011, p. 7.

[12] R. Pichler, A. Polleres, S. Skritek, and S. Woltran, "Redundancy Elimination on Rdf Graphs in the Presence of Rules, Constraints, and Queries," in *Web Reasoning and Rule Systems*, Springer, 2010, pp. 133–148.

[13] Server-Sent Events, http://www.w3.org/TR/eventsource/

[14] The Constrained Application Protocol (CoAP), https://tools.ietf.org/html/rfc7252

[15] S. Khan and M. Safyan, "Semantic Matching in Hierarchical Ontologies," *Journal of King Saud University - Computer and Information Sciences*, vol. 26, no. 3, pp. 247–257, Sep. 2014.

[16] M. Cheatham and P. Hitzler, "String similarity metrics for ontology alignment," in *The Semantic Web–ISWC 2013*, Springer, 2013, pp. 294–309.

[17] DogOnt, http://elite.polito.it/ontologies/dogont/dogont.html

[18] X. Zhang, C. Zhao, P. Wang, and F. Zhou, "Mining Link Patterns in Linked Data," in *Web-Age Information Management*, Springer, 2012, pp. 83–94.

[19] J. Gao and S. Mazumdar, "Exploiting Linked Open Data to Uncover Entity Types," presented at the Semantic Web Evaluation Challenge, 2015, pp. 51–62.

[20] Biotop, http://purl.org/biotop/biotop.owl

[21] SAREF4EE, http://ontology.tno.nl/saref4ee/

[22] Linked Open Vocabularies, http://lov.okfn.org/dataset/lov/

[23] B. Haslhofer and W. Klas, "A Survey of Techniques for Achieving Metadata Interoperability," ACM Computing Surveys, vol. 42, no. 2, pp. 1–37, Feb. 2010.

[24] N. bin M. Ibrahim, B. Hassan, and M. Fadzil, "A Survey on Different Interoperability Frameworks of SOA Systems Towards Seamless Interoperability," in *Information Technology (ITSim), 2010 International Symposium in*, 2010, vol. 3, pp. 1119–1123.

[25] Z. Zhang, C. Wu, and D. W. Cheung, "A Survey on Cloud Interoperability: Taxonomies, Standards, and Practice," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 13–22, 2013.

[26] GSMA, IoT Big Data Harmonised Data Model, 2016

[27] OneM2M Technical Specification, Base Ontology, 2015

[28] W. Peters, "Establishing Interoperability Between Linguistic and Terminological Ontologies," pp. 27–42, 2013.

[29] P. Lehti and P. Fankhauser, "XML data integration with OWL: Experiences and challenges," in *Applications and the Internet*, 2004, pp. 160–167.

[30] P. T. T. Thuy, Y.-K. Lee, S. Lee, and B.-S. Jeong, "Exploiting XML Schema for Interpreting XML Documents as RDF," 2008, pp. 555–558.

[31] P. Andrews, I. Zaihrayeu, and J. Pane, "A classification of semantic annotation systems," 2010.

[32] V. Saquicela, L. M. Vilches-Blázquez, and Ó. Corcho, *Semantic annotation of RESTful services using external resources*. Springer, 2010.