Semantic Interoperability in the IoT
Extending the Web of Things Architecture

OSCAR NOVO, Ericsson Research, Finland and Aalto University, Finland
MARIO DI FRANCESCO, Aalto University, Finland

The adoption of the Internet of Things is gradually increasing. However, there is still a significant obstacle that hinders its adoption as a truly ubiquitous technology: the ability of constrained devices to unambiguously exchange data with shared meaning. In this respect, the World Wide Web Consortium has developed the Web of Things architecture to provide semantic data exchange. However, such an architecture does not cover all possible use cases and still has important limitations. This article specifically addresses these issues. In particular, it discusses the design and implementation of a solution that extends the Web of Things architecture to achieve a higher level of semantic interoperability for the Internet of Things. The proposed solution relies on a human-assisted translation process and defines an architecture that enhances the semantic compatibility between components in the World Wide Web Consortium and the Internet Engineering Task Force. The effectiveness of the proposed solution is demonstrated through both a quantitative and a qualitative evaluation, in terms of performance and key properties in comparison with the state of the art.

CCS Concepts:
• Information systems → Web data description languages;
• Software and its engineering → Interoperability.

Additional Key Words and Phrases: Web of Things, Internet of Things, Semantic Interoperability, Semantic Technologies

ACM Reference Format:

1 INTRODUCTION

The Internet of Things (IoT) is suffering from a lack of semantic interoperability at the application layer [46, 48]. In fact, information models and computational ontologies in the IoT have been developed by different organizations, thereby creating semantic silos within different IoT ecosystems. According to [46], there are almost 300 different ontologies defined for the IoT. This is because the representation of knowledge is inconsistent throughout several standards and solutions, thereby causing the fragmentation of semantic information. In other words, current IoT ontologies are defined in very specific semantic domains, resulting in a semantic diversity that causes a lack of understanding between IoT devices belonging to different platforms. As a consequence, related information models and ontologies are often incompatible. A different aspect is the high evolvability of IoT systems, namely, when semantics are updated or extended over time. Indeed, changes to...
the semantics of an IoT system can lead to incompatibility between IoT applications supporting different versions of the same semantic.

Many standardization bodies have developed new protocols and architectures for the IoT. Among them, the Internet Engineering Task Force (IETF) is one of the few ones with the authority to develop global Internet standards. However, the IETF has focused on defining an architecture for resource-oriented applications running on top of constrained devices over the Internet. In contrast, the World Wide Web Consortium (W3C) has created the Web of Things working group to develop standards for enable easy semantic integration across IoT platforms. The Web of Things (WoT) is a concept that establishes the use of web technologies as a foundation for IoT application and services. In particular, the WoT has developed the Web of Things (WoT) architecture [51] at the application layer of the Internet protocol stack. The WoT architecture consists of rich metadata that describe the actual information and the interaction models exposed by IoT applications. Accordingly, each IoT device exchanges a metadata file with other devices it needs to interact with, which allows to interpret the corresponding interfaces. In other words, the WoT architecture enables IoT applications to share the same semantic meaning.

Research challenges. The WoT architecture is a key building block for IoT applications; however, it also has several important limitations. In fact, the WoT architecture currently relies on appropriate interpretation of semantics. However, an IoT device needs generic contextual knowledge to correctly interpret the semantic information of another device. In other words, many IoT devices may not able to communicate with each other unless they are capable of understanding the information in the metadata. For instance, an IoT device might semantically represent a temperature sensor using the “temperature” definition from the Schema.org ontology while other IoT device might use the “3303” representation from IPSO. Consequently, an IoT device might not have the ability to semantically interpret the function of another IoT device, thereby preventing the two of them to meaningfully interact.

Some tools have been developed within the WoT architecture itself [51]; automatic solutions have also been proposed, particularly, for ontology matching [36]. However, these solutions are still unable to reliably understand the environment in terms of the related high-level information [2]. Moreover, the WoT architecture is not fully interoperable with the existing IoT standards developed by the IETF; thus, further integration is needed despite the ongoing standardization efforts.

Our contributions. We specifically address these challenges by extending the Web of Things architecture to enable seamless semantic interoperability between IoT devices. Our proposed solution builds on web standards and on semantic translations to cope with heterogeneous and evolving systems. In particular, we leverage user-based translations of semantic information, as humans are able to accurately interpret information due to their extremely broad contextual knowledge, world knowledge, and experience [14]. We design and implement a prototype of the system, then show its applicability to a real use case. Experimental results demonstrate the scalability of our solution for IoT scenarios involving a large number of devices. Finally, a qualitative evaluation contrasts the properties of the proposed approach against the state of the art.

The major contributions of this work are the following.

- We identify the shortcomings of the solutions in the W3C and IETF standards and describe the related limitations in enabling semantic interoperability for the IoT. In particular, we thoroughly review the background on ontologies, interoperability, and standards in the specific context of the IoT.
- We devise a solution that integrates the W3C and IETF standards by extending the WoT architecture so as to enable interpretation of metadata even when devices understand different
semantics. The proposed approach leverages a human-assisted semantic translation process to improve the semantic interoperability between IoT devices.

- We demonstrate the effectiveness of our solution through two different evaluation studies: a quantitative performance evaluation, showing that our scheme has limited overhead and scales with the number of devices; a qualitative evaluation of the translation mechanism through a comparison with similar solutions in the state of the art.

**Outline.** The rest of this article is organized as follows. Section 2 describes the different standards defined by the IETF and the W3C at the application layer, with focus on the current W3C WoT architecture, as well as the state of the art on semantic interoperability in the IoT. Section 3 examines the drawbacks of the W3C WoT architecture in detail. Section 4 introduces a use case in logistic and transportation as a motivating scenario. Section 5 details our proposed solution for semantic interoperability in the IoT, then Section 6 describes our prototype implementation. Section 7 evaluates the effectiveness of the proposed solution through both a quantitative and a qualitative study. Finally, Section 8 presents some concluding remarks as well as directions for future research.

## 2 BACKGROUND

Before proceeding further, it is important to understand how the work done by the W3C on semantic interoperability coexists and integrates with the different IoT standards defined at the application layer. There are many organizations defining new mechanisms and frameworks for the IoT. However, the IETF is one of the few organizations with the ability to develop global Internet standards. In addition, the related standards complement those defined in the W3C which are commonly used for IoT applications. Accordingly, this section first overviews the most recent IoT standards developed by the IETF. It then describes the WoT architecture defined by the W3C and discusses the state of the art on semantic interoperability for the IoT. Finally, it reviews existing ontologies in the IoT ecosystem by means of their properties and limitations.

### 2.1 IoT Standards in the IETF

The IETF plays a significant role in defining new standards for the IoT. Some of the most widely-used standards defined by the IETF in this context are the **Constrained Application Protocol (CoAP)** [41], **IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN)** [27] and the **IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL)** [52]. The **Constrained Application Protocol (CoAP)** is a specialized Internet Application Protocol intended for use in resource-constrained Internet devices. 6LoWPAN provides a mechanism to accommodate IPv6 data packets on IEEE 802.15.4 networks [1]. RPL, on the other hand, is the IPv6 Routing Protocol for constrained networks. In addition to them, the IETF has developed other IoT standards at the application layer, including a new service discovery for the IoT [42] and new serializations such as the **Constrained RESTful Environments (CoRE) Link Format** [40]. Figure 1 depicts these new standards defined or adopted by the IETF for the IoT.

As shown in the figure, all the different IoT devices and resources in the IETF stack are uniquely identified through **Uniform Resource Identifiers (URIs)** [4]. These identifiers are represented by different character encodings, typically UTF-8 [53] or UTF-16 [18]. Moreover, the different IoT standards at the application layer support commonly used data formats, including **JavaScript Object Notation (JSON)** [7], **Concise Binary Object Representation (CBOR)** [5], and the **Extensible Markup Language (XML)** [6]. Finally, the IETF stack targets security in the IoT by providing communication security for datagram protocols such as CoAP. The new security protocol defined by the IETF in this context is **Datagram Transport Layer Security (DTLS)** [38].
Fig. 1. IoT Standards Developed by the IETF and W3C at the Application Layer

On the other hand, the use of web services and the Representational State Transfer (REST) architectural style have become ubiquitous in Internet applications. REST is an architectural style that allows interoperability between different constrained devices by using textual representations of resources and a predefined set of stateless operations mapped onto the methods of the Hypertext Transfer Protocol (HTTP) [15]. Thereby, the IETF has also defined a new link serialization suitable for constrained RESTful environments called Core Link Format [40]. Such a format is employed by different services to describe resources, attributes and other relationships between constrained devices.

Furthermore, direct discovery of resources is not feasible in many IoT scenarios due to intermittent availability of nodes (for instance, due to power management based on duty-cycling) or networks with unreliable support for multicast traffic. As a consequence, the IETF has defined a new component called Resource Directory [42] to discover resources in different networks. The Resource Directory hosts descriptions of the resources on an IoT network and allows to retrieve the IP addresses associated with these resources.

The example below shows all the different technologies described in this section combined in a single operation. The example shows a client performing a resource lookup to a Resource Directory. The client sends a CoAP GET request [41] to the Resource Directory. In this example, the Resource Directory replies with the available resources in the network matching the query. The information in the payload is encoded according to the UTF-8 unicode CoRE link format, and several URIs such as /power or /sensor are used through the example.

Listing 1. Resource Directory: Lookup operation

<table>
<thead>
<tr>
<th>Req: GET /rd-lookup/ep?et=power-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res: 2.05 Content</td>
</tr>
<tr>
<td><a href="">coap://[2001:db8:3::127]:5683/power</a>;ep=&quot;node5&quot;;et=&quot;power-node&quot;;ct=&quot;40&quot;;lt=&quot;600&quot;,</td>
</tr>
<tr>
<td><a href="">coap://[2001:db8:3::129]:5683/sensor</a>;ep=&quot;node7&quot;;et=&quot;power-node&quot;;ct=&quot;40&quot;;de=&quot;floor-3&quot;</td>
</tr>
</tbody>
</table>

2.2 IoT Standards in the W3C

Motivated by the lack of interoperability across different IoT platforms, the W3C has focused on defining a vendor- and application-independent framework that supports interoperability between different IoT devices. Such a framework is called the W3C Web of Things (WoT) architecture [51] and allows IoT devices to communicate with each other independent of their specific implementation.

The W3C WoT architecture [51] is built on existing well-known web standards and provides a standard way to describe IoT interfaces. Figure 1 illustrates the different components defined in the WoT architecture. Every device has to provision a metadata file called Thing Description (TD) [49].
A TD provides a semantic description of a specific IoT device, as well as its interactions. In other words, the TD defines the semantics and the way of interacting with a specific IoT device.

The default interaction types of a TD are called Property, Action, and Event. These were found to cover the majority of IoT interactions. Properties describe the attributes of an IoT device (e.g., humidity value, door locked or unlocked). Actions invoke processes (e.g., switch on a light, open a window) while Events asynchronously notify when a specific condition occurs (e.g., alarm activated, windows opened). The TD relies on the Resource Description Framework (RDF) [37] as the underlying data model. RDF is a W3C standard originally designed as a metadata data model for data interchange on the Web. Moreover, TD instances are serialized in JSON for Linked Data (JSON-LD) format by default. JSON-LD [25] adds a semantic layer on top of the JSON specification, meaning that the terms that appear in a TD are associated with concepts from a shared vocabulary.

Figure 2 and Figure 3 show sample representations of TDs file serialized in JSON-LD format. The TD described in Figure 2 represents a temperature sensor named temperature that has a unique interaction property. The interaction property provides the temperature of the sensor under the "GET coap://www.example.com:5683/temp/val" operation serialized as a JSON value. The vocabulary used in the sample is defined by the W3C under the context keyword. The sensor and resource types belong to the document defined in line 5 of Figure 2, and the location vocabulary (line 17) belongs to the document defined in line 4. Furthermore, all the key terms in the TD – such as link or href – are defined in the http://www.w3.org/ns/td document. Each TD instance must embed such a document to understand the key terms in the TD.
TD instances can be placed in a component called Thing Directory [12]. A Thing Directory stores TDs and provides an interface for registration, registration updates, and registration removal of TDs. Such an interface also provides lookup operations based on SPARQL queries. SPARQL [44] is a semantic query language capable of retrieving and manipulating data stored in RDF format.

Once a device obtains the TD of another IoT device from a Thing Directory, it must first interpret and understand the information therein to successfully interact with the IoT device. Afterwards, the communication between the devices can be done automatically – without any manual intervention.

2.3 Existing Efforts on Semantic Interoperability in the IoT

In addition to W3C and its Web of Things (WoT) architecture, other standardization groups have taken similar efforts to bring some level of consistency to the IoT semantics. One of these organizations is the Internet Architecture Board (IAB) that through the IoT Semantic Interoperability Workshop [20] gathered the different approaches used by companies and standardization bodies to accomplish semantic interoperability at the application layer in IoT scenarios. The workshop particularly focused on the lack of common semantic interfaces and semantic translations, runtime semantic discovery, and security. The lack of common semantic interfaces concerns the way in which the information is represented through different serialization formats such as CBOR [5], JSON [7] and XML [6]. The operation of semantic translation provides the ability for gateways and similar devices to dynamically translate semantic information [43]. The runtime semantic discovery approach allows devices to dynamically discover data. An example of this approach is the Hypermedia as the Engine of Application State (HATEOAS) [29, 32] principle, a mechanism allowing IoT devices to dynamically navigate to the appropriate resource of an application by traversing hypermedia links without any prior knowledge on the application. Security tackles the implications of actual security models in semantics [34, 50]. In addition, the workshop described some of the existing taxonomies and semantic representations [19, 39] in the IoT.

The Internet Research Task Force (IRTF) organized a different event called Workshop on IoT Semantic Interoperability (WISHI) [21] where similar issues were addressed. The outcome of both workshops was similar. Among all the different topics discussed, semantic translation has some similarities with the work done in the W3C. However, the solutions for semantic translation discussed in the workshops focus on creating and normalizing translators between different data semantics. In contrast, the W3C provides a platform-independent framework that enables easy integration across IoT platforms and application domains.

Industry-specific standard organizations have also created their own solutions and specifications for heterogeneous IoT systems. OneM2M\(^1\) is an international partnership project consisting of the major ICT standard organizations worldwide. The goal of oneM2M is to develop a common service layer for diverse hardware and software components in the IoT, so as to ensure that devices can communicate on a global scale and avoid fragmentation [45]. Although OneM2M has the same objective than the W3C WoT working group, their solutions differ in several aspects. As stated in [35], OneM2M lacks the connection to the web capabilities enabled by W3C standards. On the other hand, W3C WoT provides additional descriptive capabilities beyond OneM2M services.

Moreover, the ETSI Industry Specification Group for cross-cutting Context Information Management (ETSI ISG CIM) specifies an API for context information management specifically targeting Smart Cities [9]. The API allows users to produce and consume context information from IoT data sources. The API includes what is described by the data, ownership, and validity of the data. However, the API specified by ISG CIM provides context between domains and data models, while the W3C WoT specification defines the interactions between IoT devices.

\(^1\) http://www.onem2m.org/
Beyond that, current research activities outside standardization bodies on the IoT targeted the design of adaptive middleware [30, 33] and system architectures [8, 26, 54] for specific scenarios, including vehicle-to-vehicle communication. In contrast, this work is not proposing a general ontology for the IoT, but it rather builds on the W3C architecture to enable the interconnection between the devices that cannot interpret the information in the TDs.

Furthermore, a number of projects within the Horizon 2020 program of the European Union have addressed interoperability and semantics in the context of the IoT. Among them, the Federated Interoperable Semantic IoT Testbeds and Applications (FIESTA-IoT) project provides an IoT testbed infrastructure to enable the easy submission of IoT experiments – [28] specifies the building blocks of the IoT testbed platform. One of them is the IoT Registry, a component that stores all the data injected by the different testbeds, offering dynamic discovery of all the resources in the FIESTA-IoT infrastructure. The FIESTA-IoT platform is evaluated in [47] through the outcomes from eleven IoT deployments. Despite its usability, FIESTA-IoT does not provide a solution for systems that are semantically incompatible. Its testbed ecosystem helps to verify the interoperability across multiple platforms; however, the platform itself does not provide any solutions for semantically inconsistent platforms.

Worldwide interoperability for semantics IoT (Wise-IoT) is another related project within the same EU program. Wise-IoT aims at consolidating the interoperability of IoT existing systems through a semantic mediator. The semantic mediator, known as Morphing Mediation Gateway (MMG), acts as a bridge between multiple platforms: it translates and converts data from one domain to another using oneM2M semantic annotations. Still, the focus of Wise-IoT is on how information is modeled, not on how IoT devices and the related services are described, which is instead the main focus of the W3C Web of Things architecture.

Other EU projects include: BIG IoT, which defines a generic, unified Web API for smart objects; AGILE, which develops an IoT gateway that provides support for several wireless and wired IoT networking technologies (e.g., ZWave, ZigBee, and Bluetooth Low Energy); and TagItSmart!, which specifies a smart tag technology for tracking and monitoring IoT devices. Unfortunately, none of these projects is related to the main goal of the W3C WoT architecture – namely, on how IoT devices can interact with each other and how the services provided by them can be described.

2.4 Ontologies in the IoT

Ontologies provide a structural framework to describe knowledge by using classes, properties, and relations. The heterogeneity of data in the IoT has resulted in the creation of several ontologies that address issues of interoperability among sensor data. Some IoT ontologies describe core concepts common to all IoT applications. That is the case of the Semantic Sensor Network (SSN) ontology [10] published as a W3C Recommendation. SSN provides generic concepts for describing sensors and actuators, their properties, observations, and procedures. The Sensor, Observation, Sample, and Actuator (SOSA) ontology [23] is a lightweight version of SSN. It is a general-purpose ontology with elementary classes and properties. Both SSN and SOSA are meant to support a wide range of IoT use cases and applications.

Unlike SSN and SOSA, other ontologies describe concepts that are specific to certain domains. The Smart Appliance REFerence (SAREF) ontology [11] – standardized by ETSI – specifies concepts in the smart appliances domain. The SAREF ontology provides a list of functions with associated commands. For instance the function type “switching on / off” is associated with the command

“switching on”, “switching off”, and “toggle”. Depending on the function, an IoT device can be associated to a certain state (i.e., being on or off).

SSN, SOSA, and SAREF are ontologies that rely on the semantic web. However, there is another set of ontologies that do not rely on it, such as IPSO Smart Objects or Bluetooth Low Energy profiles.

Over the past years, a wide number of ontologies have been developed to map the IoT ecosystem. Bajaj et al. [3] overview semantic ontologies in the context of the IoT. The study therein concludes that existing ontologies are not comprehensive enough to document all the knowledge necessary for IoT applications. Ultimately, ontologies simply move the interoperability problem at a higher level; the lack of interoperability still exists when IoT components refer to different ontologies.

One way to address this problem is to construct an alignment ontology. An alignment ontology provides a mechanism to relate concepts and vocabularies to different ontologies. Such an approach requires sustained commitment: alignments need to be fully consistent and maintained over time. This is seldom achieved with the current ontology alignments: according to the study in [16], the majority of the considered 97 ontology alignment tools stopped being maintained after 1 or 2 years from creation. Besides that, ontology alignment requires that the IoT devices understand the alignment representations. As a consequence, such a solution comes at a cost related to the confidence in the appropriate interpretation of semantic meaning. Automated ontology alignment still provides unsatisfactory results for many use cases: according to [2], they only obtain an accuracy of 40% for multi-language matching and even lower values (e.g., 20% on the average) for complex matching – with correspondences involving more than two entities. Therefore, incorporating user interaction in alignment ontology tools is still required to provide quality alignments which are hard to achieve by fully-automated systems.

The WoT architecture provides a domain-specific metadata (i.e., through a TD) for which the knowledge on IoT systems can be conceptualized. However, any domain-specific vocabulary is outside the scope of the activities in the W3C WoT working group. As a consequence, certain IoT devices are not able to use ontology alignment to map the information in the TDs. Ontologies are designed with certain background knowledge and, in a given context, that knowledge does not become part of the ontology specification. Such a lack of background knowledge increases the difficulty of alignment mechanisms and creates several ambiguities. In addition, the constrained nature of many IoT devices as well as the large heterogeneity in the mechanisms for ontology alignment hinders a holistic semantic interpretation in the WoT architecture. In these cases, the W3C WoT working group does not provide any measures. Instead, our solution involves end-users in achieving semantic interoperability within the WoT architecture so as to avoid errors and ambiguous results.

3 PROBLEM STATEMENT

The beginning of this article has introduced the limitations of the WoT architecture. This section expands on these limitations by providing additional details.

- **Lack of generic contextual knowledge:** As explained above, all the terms that appear in a TD document are associated with uniquely identified concepts from shared ontologies. Each ontology is defined in an external context document. In this respect, the WoT architecture assumes that all parties are able to understand all the information specified in the TD instances. In other words, the W3C WoT working group does not cover domain-specific knowledge, as it is outside of its scope. In many scenarios, however, IoT devices are not able to interpret relevant information from those TD instances, as they are unable to understand their context. For instance, the sample TD in Figure 2 assumes that all parties understand the vocabulary...
defined in lines 4 and 5 therein. However, an IoT device may not understand that specific context information, thus, failing to associate a semantic meaning with the sensor or location terms defined in that TD. As a consequence, the device could not interpret the information in that TD.

- **Information deficiency:** The WoT architecture assumes that every IoT device uses appropriate tools to understand the content of a TD. The drawback of such an approach is that different tools might need different information to interpret the context of a TD. Otherwise, a TD can lead to different interpretations of the same information or even prevent correct interpretation by others. Going back to the example in Figure 2, a specific IoT device could understand that the property interaction returns the maximum temperature that a device can operate, instead of the general temperature of the environment. In general, every IoT device can interpret the semantics of the TD instances differently, according to the current implementation of their parsing tools.

- **Low evolvability:** Evolvability is the degree to which parts of the semantics can be incrementally changed during time. Changes to the semantics of an IoT system can lead to semantic incompatibility between the IoT applications supporting different versions of the same semantic. The current WoT architecture is designed with a low level of evolvability in mind as to handling different versions of the same ontology. That is, the architecture itself does not adapt easily to the incremental changes in the semantics of the IoT devices.

- **Lack of interoperability with the IETF standards:** The W3C and the IETF standards for the IoT have evolved over the years according to slightly different paths, thereby causing issues in the binding between some of their modules. One of the challenges is finding the right information by means of different IoT discovery mechanisms. While the CoRE Resource Directory [42] allows to discover resources and perform lookups on them, it is the Thing Directory that provides the related semantic information. However, once a resource and its lookup information are discovered, the Resource Directory does not provide any additional information to associate that with the semantic information of such a resource in the Thing Directory. In other words, once a device finds the address lookup of a sensor through the CoRE Resource Directory, the device lacks the means to easily find the TD instance of that sensor in the Thing Directory.

Indeed, the solution proposed in this article (and described next) targets all these limitations by extending the WoT architecture with a standard-based approach.

### 4 USE CASE

While the WoT architecture is general and applies to any IoT scenarios, the rest of the article focuses on a representative application domain – namely, **logistic and transportation** – for the sake of clarity.

The considered use case is represented by an international port, in which shipping containers carried on cargo ships are handed over to different transport vehicles. Each shipping container runs an IoT system that collects data from within the container; among them, temperature and humidity are particularly important. In fact, condensation may result in the stored items to get damp; in the worst case, some areas of the shipping container can get sodden and start growing molds. Moreover, some goods need to be stored at a specific temperature range to prevent damage.

When shipping containers arrive to the international port, the **Port Management System** checks the temperature and humidity values of each container before moving them to ground transportation vehicles. The Port Management System plans the priority of each container accordingly before informing transportation companies on the pertinent arrivals.
In this context, the ultimate goal of the Port Management System is to understand the information provided by each shipping container. We assume that the international port handles shipping containers from all over the world; as a consequence, the actual IoT systems in each container might be different. Therefore, the Port Management System needs to discover first the interfaces in each container system before interacting with them; such a discovery leverages the WoT architecture. In addition, the new shipping containers that arrive to the international port are discovered using the Resource Directory, as described in Section 2.1.

In the following, we restrict our attention to the temperature values of the shipping containers for the sake of simplicity, even though many other parameters could be monitored in a real scenario.

5 SYSTEM DESIGN

The current WoT architecture relies on the proper interpretation of the TD information [49]. A device needs generic contextual knowledge to interpret the semantic information of another IoT device. Even though the WoT architecture provides some tools to correctly interpret such information [51], current technologies have not reached the point to efficiently understand the environment and interpret the information upon it [2]. On the other hand, humans have the capacity to interpret the information accurately due to their extremely broad contextual knowledge, world knowledge, and experience [14].

We have extended the W3C WoT architecture as illustrated in Figure 4. Our solution provides a new function for translating and interpreting TD instances that cannot be understood by IoT devices. Such a function is based on human-made translations, for instance, by domain experts or application developers. The proposed solution also provides a new interface that facilitates the interaction of users with the Thing Directory, which is extended with the ability to store the translations. Finally, the proposed solution extends the Resource Directory with additional information to enable a smooth interconnection with the Thing Directory. The extensions are backward compatible with the current W3C WoT architecture and follow the W3C standards.

5.1 Components

On a high level, our solution is divided into the following logical entities:

- The Thing Directory [12] is a database of TDs. The Thing Directory features an API to create, read, update and delete a TD. It assigns a new and unique identifier to each TD uploaded into the system. Such an identifier is used for accessing and maintaining the information associated with a TD. The Thing Directory is extended in our system with translations. Several users
can simultaneously create translations in our system. Translations are specifically created between two devices where one of them (the receiver of the TD) cannot interpret the TD of the other. Human operators adapt the TD accordingly and generates a new TD called translation from the original TD. Translations are based on the TD information of the receiver since it is the one describing the information and vocabulary that a device is able to understand. Translations are stored in the Thing Directory, and they can be created, updated, deleted or read through a newly implemented API. The Thing Directory also assigns a new and unique identifier to each translation.

- The Management User Interface is a new component that provides an easy-to-use graphical interface for users to interact with the Thing Directory. Such a component connects with the Thing Directory over HTTP and employs its TD and translation APIs. Figure 5 shows snapshots of the interface. The user interface allows to perform the following operations (Figure 5a):
  - display a list of all available TDs in the Thing Directory;
  - display a list of all available translations in the Thing Directory;
  - display a list of all unhandled translations in the Thing Directory;
  - create a new TD or update an existing TD in the Thing Directory;
  - create a new translation or update an existing translation in the Thing Directory;
  - delete a TD in the Thing Directory;
  - delete a translation in the Thing Directory.

- The Endpoints are constrained devices connected via a short-range wireless communication technology. Depending on their type, devices provide a CoAP client or a CoAP server interface [41] for actuation and data access; in some cases, devices can provide both interfaces at the same time. All devices are associated with an IP address and port.

- The Resource Directory [42] provides a mechanism to discover IoT resources in constrained networks. It also implements a CoAP REST interface [41] for discovery, registration, and lookup of these resources. Our solution extends the Resource Directory with a new parameter,
namely, the identifier assigned by the Thing Directory to the TD of a specific endpoint (this is necessary as every endpoint has a different TD). Such information is included in the Resource Directory during the registration phase by the endpoint. As a consequence, TDs can easily be located from the Thing Directory after the TD identifier is obtained from a query targeting the Resource Directory.

5.2 Interfaces

This section describes the interactions between the different components in our use case (described in Section 4), with focus on their interfaces.

In particular, Figure 6 illustrates two CoAP endpoints, Endpoint A and Endpoint B, where Endpoint A acts as a CoAP server (temperature sensor in the shipping container) and Endpoint B acts as a CoAP client (Port Management System). In this scenario, the Port Management System is interested in the temperature of the shipping container. Our solution assumes the most complex situation where the Port Management System does not have any knowledge of the resources in the network (temperature sensors and shipping containers) or about their semantics. Consequently, the location of the endpoints and their semantics are unknown and have to be discovered by the Port Management System. We assume that every endpoint in the architecture contains its own TD. The TD of Endpoint A is described in Figure 2 and the TD of Endpoint B is described in Figure 3.

As shown in Figure 6, the whole communication process can be divided in four phases: registration of information, discovery of information, retrieval of semantic information, and information retrieval. We detail these phases below.

**Information Registration:** During this phase, all CoAP endpoints register their TDs through the registration interface defined in the Thing Directory specification [51]. Before a CoAP endpoint can register its TD into the Thing Directory, it must first know the address and port of the Thing Directory. Currently, the specification does not define any mechanism to obtain that information.
Consequently, our solution simply pre-configures the address and port of the Thing Directory in each CoAP endpoint.

The example below shows the TD’s registration operation performed by the Temperature endpoint of a shipping container identified as container_4:

Listing 2. Thing Directory: Registering a TD

```
Req: POST coap://td.example.com/td
  Content-Type: application/ld+json
  Payload:
  {
    "@context": ["http://www.w3.org/ns/td",
                 "http://www.w3.org/2003/01/geo/wgs84_pos#",
                 "http://uri.etsi.org/m2m/saref#",
                 "saref:Sensor",
                 "saref:Resource"],
    "@type": ["Sensor"],
    "name": "temperature",
    "base": "coap://www.example.com:5683/temp",
    "interaction": [{
      "@type": ["Property", "Resource"],
      "name": "getTemperature",
      "outputData": {"type": "number"},
      "writable": false,
      "geo:location": "container_4",
      "link": [{"href": "val"}, "application/json"]
    }]
  }

Res: 2.01 Created
  Location-Path: /td/a912a00f
```

If the TD is stored successfully in the Thing Directory, the Thing Directory returns a 201 Created response code and an identifier that uniquely represents the TD instance in the directory. Such an identifier can be used to access, update or delete the TD from the system.

After registering the TDs in the Thing Directory, the CoAP endpoints acting as CoAP servers register their resources into the Resource Directory through the registration interface defined in the Resource Directory specification [42]. In this case, the endpoints also need to know the address and port of the Resource Directory before registering their resources. Even though the Resource Directory document defines a different mechanism for discovering a Resource Directory, our solution uses default information in each endpoint. In most cases, only the CoAP servers (such as Endpoint A in Figure 6) register their resource information into the Resource Directory to let other endpoints discover them. Since the interfaces in the Resource Directory are specified in Core Link Format [40], each CoAP endpoint willing to interact with the Resource Directory must first extract and translate the information of its TD into the CoAP Link format. Generally, the information in the TDs is serialized in JSON-LD format. After translating the information in the TD, each CoAP server includes in the Resource Directory’s registration the identifier of its TD assigned by the Thing Directory. Hence, the endpoints discovering the location of a CoAP server from a Resource Directory can easily retrieve the TD information of that CoAP server from the Thing Directory.

The example below shows the Resource Directory’s registration operation of a Temperature endpoint:

Listing 3. Resource Directory: Registration operation

```
Req: POST coap://rd.example.com/rd?ep=temperature_4&sem=a912a00f
  Content-Format: 40
  Payload: <temp/val>;ct=50;rt='getTemperature'

Res: 2.01 Created
  Location-Path: /rd/4521
```

The operation shows how the Temperature endpoint with name temperature_4 registers a new resource type (rt) getTemperature. The operation does not include a lifetime variable in the request and assumes the Resource Directory is accessible on its default port 5683. If no lifetime is included,
a default value of one day will be assumed by the Resource Directory. The operation includes the
TD identifier \textit{a912a00f} of this endpoint – assigned by the Thing Directory – in the \textit{sem} parameter.

\textbf{Information Discovery:} During the information discovery phase, endpoints acting as CoAP
clients (such as Endpoint B in Figure 6) perform specific lookup operations via the Resource
Directory to find resources in the network. In addition, the Resource Directory supports lookup
observations \cite{17}, meaning that endpoints can subscribe to a lookup query and the Resource
Directory will provide the resource information every time the lookup query is satisfied by any new
registered resource. The following operation shows an example of a lookup observation performed
by the \textit{Port Management System} endpoint.

\textbf{Semantic Retrieval:} A CoAP client can already reach a CoAP endpoint after obtaining its
address and port. However, the CoAP client still does not know the interfaces of such a CoAP
endpoint at that time. In this phase, our solution defines a new method where a CoAP client requests
the information associated with the TD of the CoAP endpoint from the Thing Directory so as
to interpret its semantics. In detail, the CoAP client finds the specific TD in the Thing Directory
through the TD identifier provided in the lookup response of the Resource Directory. The following
operation shows how the \textit{Port Management System} requests the TD information of the \textit{Temperature}
endpoint from the Thing Directory.

The operation illustrates the request and the first response of a lookup observation of the
endpoints in the network with name \textit{temperature\_4}. The operation returns the address, port and
TD identifier of an endpoint matching the lookup operation.

For the sake of simplicity, the example does not show the payload of the response operation,
namely, the TD of the \textit{Temperature} endpoint (i.e., the one in Figure 2).

The W3C WoT architecture assumes that TDs are understood by all the CoAP endpoints. However,
that condition is not satisfied in some scenarios as explained in Section 3. In fact, a CoAP endpoint
might not be able to understand and interpret a TD. In those particular cases, our solution goes one
step further and provides a specific solution where the CoAP endpoint requests a translation from
the Thing Directory.

The operation below shows how the \textit{Port Management System} requests a translation of the
\textit{Temperature}’s TD. The operation includes under the \textit{source} value the associated TD identifier of
the endpoint requesting the translation, the \textit{Port Management System}. The operation also includes
under the \textit{target} parameter the identifier of the TD that needs to be translated, the \textit{Temperature}
endpoint.
Listing 6. Thing Directory: Lookup of a Translation

```
Req: GET coap://td.example.com/translate-lookup/?source=3900a12a &target=a912a00f
Res: 2.00 OK
Content-Type: application/ld+json
Payload:
   {Translation in TD format}
```

For the sake of simplicity, the example does not show the payload of the response operation, namely, the modified version of the TD for the Temperature endpoint (i.e., the one in Figure 7).

The operation returns the modified TD of the Temperature endpoint. Specifically, the TD is modified in such a way that the Port Management System can interpret it correctly. If a translation for a specific TD is not available in the Thing Directory, the Thing Directory stores the missing translation in the system and informs the CoAP endpoint about it. Thereafter, a user can manually create and store the missing translation in the Thing Directory through the Management User Interface. The CoAP endpoint can immediately request a translation once it is available in the Thing Directory.

**Information Retrieval:** By this time, a CoAP client with the TD (or alternatively its translation) and location of a CoAP server has all the necessary information to connect and interact with it. As a consequence, it can directly fetch the intended data.

Listing 7. Retrieve the Information of an Endpoint

```
Req: GET coap://[FDFD::1]:5683/temp/val
Res: 2.00 OK
Payload:
   17.1
```

5.3 Translations

A translation is a modified version of a TD. The translation process requires the involvement of a domain expert who should provide the relevant conceptual knowledge in each scenario. In particular, a domain expert should have knowledge about the Port Management System and the IoT systems of the shipping containers for our considered use case (described in Section 4). Moreover, the domain expert should be able to understand the context described in the TDs of the shipping containers. The purpose of the translation is to allow an endpoint to correctly interpret a TD. Translations are serialized into JSON-LD; they must be valid and well-formed JSON instances.

Every TD uniquely defines the interactions and the metadata of a specific endpoint. Since every endpoint is identified by a single TD, every translation is uniquely identified within the context of two TDs. In other words, a translation is the result of modifying a TD by employing the context and the vocabulary of another TD. A translation provides an interpretation of a specific TD to an endpoint. In fact, in the worst case scenario, every TD can have as many translations as the number of TDs in the system.

The Management User Interface provides to the domain expert the original TD that cannot be understood by a particular endpoint (target), as well as the TD of that endpoint (source). Figure 5b shows an example of the Management User Interface. The domain expert leverages the vocabulary defined in the source TD under the context keyword to verify that all the terms are properly understood and the target TD is interpreted in the right context. The domain expert can also access a translation through the Management User Interface after it has been created and stored in the Thing Directory.
Fig. 7. Translation between two TDs. The result translation is a modified TD of the temperature endpoint

The example in Figure 6 shows how Endpoint B (the **Port Management System**) needs the translation of Endpoint A (a **Temperature sensor**). The TD of Endpoint A (target TD) is shown in Figure 7a, while the TD of Endpoint B (source TD) is shown in Figure 7b. Note that Endpoint B acts as a CoAP client and, therefore, does not provide any interface to other endpoints. However, the TD of Endpoint B provides the information of the vocabulary understood by this specific endpoint under the `context` field. The generic TD keywords (e.g., base, name, @type) are defined in the https://www.w3.org/ns/td document. This document is a default document included in all TDs. The w3c-wot-common-context document is a specific document that extends the capabilities of an endpoint with further vocabulary and semantics. According to this document, Endpoint B understands the semantics of vocabulary as diverse as `power`, `pressure`, and `humidity`. The document is depicted in Figure 8.

In this particular case, the TD of Endpoint B (Figure 7b) does not provide any definition of the vocabulary `resource` and `location`. On the other hand, the vocabulary `sensor` described in the TD of the Endpoint A is defined in a different document in the TD of Endpoint B, specifically, in line 10 of the w3c-wot-common-context document (Figure 8). The TD of Endpoint A defines the vocabulary `sensor` under the `saref` document instead. A domain expert aware of that situation modifies the TD in Figure 7a accordingly, by replacing the `resource` vocabulary by `temperature` and `location` by `buildingSpace` from the TD. Since users are able to understand the semantic context of the information, they can interpret that the meaning of `resource` is similar in this context to the meaning of `temperature` defined in the TD of Endpoint B under the w3c-wot-common-context...
document (line 13 of Figure 8). In addition, the domain expert should be aware of the parsing methods utilized by an endpoint before modifying a TD in a way where parsing is still possible. In this case, the deletion of the location information as well as the geo and saref context documents from the original TD of Endpoint A should not affect the parsing of Endpoint B.

The result of the translation is a modified TD of Endpoint A, the Temperature endpoint; such a translation is shown in Figure 7c. In the figure, all the information removed from the original TD of Endpoint A has been left blank in the translation for the sake of clarity. Figure 5b shows how the translation operation looks like in the Management User Interface.

5.4 Properties of the Proposed Architecture

Our proposed architecture is characterized by several properties that help improve flexibility and scalability in scenarios with a large degree of heterogeneity.

- **Reusability**: All translations are cached in the Thing Directory and can be reused afterwards, thereby reducing the number of additional translations. As a result, domain experts only need to create a single translation between two incompatible TDs. For instance, a centralized Port Management System would need a single translation to make most of the temperature sensors from a shipping container interoperable. This is due to the fact that the temperature sensors in the shipping containers are similar and they understand the same semantics.
- **Discoverability**: All existing translations in the Thing Directory are automatically discovered, without any user intervention. A cached translation is automatically shared following endpoint requests.

- **Adaptability**: One of the main drawbacks of evolving ontologies manifests when they are modified or extended: all existing translations have to be re-generated accordingly. However, our approach does not provide a tight integration of the translations. In fact, each translation is modular and independent of each other so as to make our proposed solution less vulnerable to changes. As a result, only the translation of a given TD may require being generated again when that TD is extended or modified. Despite that, the translations of a modified TD would still be valid as long as the semantic context is still properly interpreted by the endpoints.

- **Replicability**: The translations are stored in the Thing Directory as RDF graphs. Consequently, they can be easily exported to different Thing Directories or other frameworks, thereby reducing the number of manual translations through sharing.

6 IMPLEMENTATION

We implemented a prototype of our proposed architecture following the use case described in Section 4, wherein a Port Management System gathers environmental information from sensors in several shipping containers and prioritizes them accordingly. The implementation follows the logical architecture illustrated in Figure 4; each logical entity described therein corresponds to a different software component as follows.

- The **Resource Directory** is based on [libcoap](https://libcoap.net/), an open-source and multi-platform library written in the C language. The library is designed to support all the features in the IETF CoAP specification [41]. Unfortunately, libcoap implements a deprecated version of the resource directory [42]. Therefore, we implemented a new resource directory on top of the libcoap library. Moreover, we implemented the new semantic parameter described in Section 5.1 in all its interfaces.

- The **Thing Directory** is based on the Java-based W3C WoT Thing Directory reference implementation [12]. It uses Eclipse Californium [8] for CoAP-related functions and Apache Jena [9] to process semantic data. We extended the software with the new features described in Section 5.1.

- The **Management User Interface** connects with the Thing Directory over its HTTP API and it provides a web-based interface to manage the Thing Directory, including the definition of translations. We implemented the entire component in JavaScript by using the Express [10] web application framework for NodeJS [11]. Figure 5 shows screenshots of the user interface corresponding to the different functions therein.

- The **Endpoint** components include a basic CoAP server and client application. We implemented both applications in JavaScript by using the NodeJS framework as runtime.

7 EVALUATION

This section evaluates the proposed solution through two different studies: an experimental evaluation of the architecture, with focus on scalability and performance; and a qualitative evaluation of the translation mechanism through a comparison with similar solutions in the state of the art.

7.1 Experimental Evaluation

The following presents a quantitative evaluation of the proposed solution through testbed experiments. In particular, we address the scalability and the performance of the architecture by
focusing on the registration phase, which is the most resource-intensive functionality – it carries out multiple operations involving the Resource Directory, the Thing Directory, and the IoT devices. The rest of the section introduces first the experimental setup, then discusses the obtained results.

7.1.1 Testbed Setup. We set up a testbed consisting of two computers equipped with an Intel Core i7-950 CPU at 3.07 GHz, connected through a dedicated WiFi network (i.e., with no other wireless devices). One machine ran the Resource Directory and the Thing Directory, while the other one ran CoAPBench\textsuperscript{12} to simulate a different number of endpoints, according to the scenario depicted in Figure 4. CoAPBench is a tool – similar to ApacheBench – which allows to measure the performance of CoAP servers, including how many requests they are able to handle. Specifically, CoAPBench employs virtual clients to meet a certain concurrency factor by sending Confirmable (i.e., acknowledged) CoAP requests. A client that does not receive a response within 10 seconds from a request, times out and records the loss.

As mentioned above, our evaluation focuses on the registration phase of our solution (described in Section 5.2), as it is the most resource-intensive operation in the system. Accordingly, we configured CoAPBench to register resources in the Resource Directory and TDs in the Thing Directory. We carried out experiments to characterize different network sizes and concurrent requests. In both cases, we varied the number of virtual clients in CoAPBench from 1 to 10,000. We run 10 different iterations of each experiment; the figures report the average values obtained from these iterations.

7.1.2 Experimental Results. The experimental evaluation focuses on two key parameters for IoT scenarios: scalability with the number of devices and overall performance. The obtained results are discussed accordingly below.

Scalability. We first characterize scalability in terms of the server latency for different number of clients. Figure 9 shows the related cumulative distribution functions, reported separately for the Resource Directory (i.e., Figure 9a) and the Thing Directory (i.e., Figure 9b). Figure 9a clearly shows that the latency of the Resource Directory is relatively small (i.e., below 600 ms) for up to 100 concurrent clients. The latency increases with the number of virtual client by reaching the maximum value of about 2 s for 10,000 concurrent clients (the corresponding average value is still 677 ms). This happens as registration updates the database in the Resource Directory. In contrast, Figure 9b shows a much more significant latency for the Thing Directory, up to five times higher.

\textsuperscript{12} https://github.com/eclipse/californium.tools
than that of the Resource Directory. The response time also has a much higher deviation that substantially depends on the number of virtual clients. Such a trend can be explained in terms of one specific factor. The Thing Directory must update its own database for every registration operation. The process of adding a new TD into the database of the Thing Directory indeed causes long response times and slowdowns, since the information of newly registered TDs must be compared with the information of every TD in the database. Overall, the registration of a TD is performed a single time during the lifetime of an endpoint in practice. Hence, the latencies shown for the Thing Directory would not drastically affect the overall performance of the system as it would otherwise appear from a direct comparison with the Resource Directory.

**Performance.** We now characterize the performance of the two servers, again, with reference to the registration operation (as discussed above). In particular, Figure 10a shows the throughput as a function of the number of virtual clients. The upper part of the figure shows that the Resource Directory can handle a high number of transactions per second (i.e., more than 1,000) almost irrespective of the number of virtual clients, even though there is a slight decrease when they exceed 5,000. On the other hand, the Thing Directory achieves its peak performance with less than 20 requests per second with 100 concurrent clients. The throughput in this case slightly increases with the number of virtual clients, an opposite trend with respect to the Resource Directory. This can be explained with similar arguments as those presented for the scalability analysis.

Figure 10b shows the number of requests timed out in both servers. The graph shows a consistent increase of the timeouts in relation to the number of virtual clients. The reason for the lower throughput for the registration operations is similar to the reason explained in the previous subsection. The complexity of the registration operations is due to the fact that the Resource Directory and the Thing Directory need to update their databases for each registration, slowing down considerably the performance of both servers.

### 7.2 Qualitative Evaluation

We have so far evaluated our approach without considering the user-based translation process. Next, we conduct an evaluation of user interactions in the considered scenario by following the
Class | Dimension | Attributes | COMA [31] | LogMap [24] | RepOSE [22] | This work
---|---|---|---|---|---|---
System services | Stage of involvement | Before (B), after (A), during (D), iterative (I) | B, A, I | A | A, I | D, I
Suggestion selection | Threshold / advanced filtering | | | | | 
Feedback propagation | Re-computation | | | | | 
| Conflict detection / blocking / revalidation | | | | | 

| User interface | Alignment presentation | Seven visual information seeking tasks | | | |
| | | Visual analytics | | | |
| | | Alternative views | | | |
| | | Grouping | | | |
| | | Validated / candidate mappings | | | |
| | | Metadata and context | | | |
| | | Ranking / recommendations | | | |
| | | Mapping explanation | | | |
| | | Justification | | | |
| | | Impact of their actions | | | |
| Alignment interaction | Accept / reject mapping | | | | 
| | | Create / refine mapping | | | 
| | | Search | | | 
| | | User annotation | | | 
| | | Session | | | 
| | | Create temporary mapping | | | 

Table 1. A qualitative evaluation of user interactions in our approach and the state of the art. The symbols in the table indicate the level of supported features: full, partial, marginal; or no support.

guidelines established by the Ontology Alignment Evaluation Initiative (OAEI)

7.2.1 OAEI measures. The evaluation measures identified by OAEI are summarized in [36]. We briefly introduce them before comparing our approach against the state of the art.

System services. Systems must limit user interaction as much as possible, since users capable of performing validation are a scarce resource. In particular, systems can adopt different strategies depending on the stage of involvement of the user in the validation process. Validation can happen before the matching process: the user provides an initial validation of the matching which is then applied by the matching process of the system. Validation can also happen after the matching process; in this case, the input of the user is not employed for aligning the ontologies. With iterative validation, the user is asked for feedback during different iterations of the process and the alignment is improved at each iteration. When the validation is done during the matching process, feedback propagation techniques can be applied. According to these, the matching process is applied to similar mappings. One form of feedback propagation is validating user feedback against possible errors and reporting them to the user. Another way to limit user involvement is the implementation of suggestion detection techniques. These techniques employ thresholds values for different alignments and filtering according to some principles (e.g., consistency and locality).

User interface. Ontology alignment is a complex operation for users; therefore, supporting systems should be aware of the cognitive constraints of human beings (e.g., memory or visual retention). Moreover, user interfaces have to provide enough information in a clear format for the user to perform their tasks without errors. Several functionalities, referred to as alignment presentation, address presenting visual information to support the decision process. The Visual Information Seeking Mantra defines seven tasks that need to be supported by user interfaces. In detail, the tasks

13 http://oaei.ontologymatching.org/
are the following: overview, filter, zoom, details on demand, history, relate, and extract. These tasks provide a trade-off between providing information to the user and avoiding information overload. Visual analytics combine interactive visualization techniques to support analytic reasoning. Another technique is alternative views; different views are more suitable for different tasks. Moreover, systems should not convey all the information into a single view, otherwise the user could be overwhelmed. The grouping strategy categorizes information by different criteria to help identify patterns, while candidate mappings help distinguish between different types of mappings. For instance, color-coding implements both strategies at the same time. Recommendations provide definitions of terms and give suggestions to users about specific mappings. Another important component is the explanation of the mapping suggestions. Users should provide a justification, and they should be provided with feedback about the impact of their actions. Moreover, other functionalities in the context of alignment interaction assist the user in validating the mapping. The most basic interaction allows a user to accept or reject a mapping. Some systems may not properly capture a mapping; consequently, adding a mapping manually or refining a mapping are also critical in these cases. Searching is another important functionality to minimize the cognitive load of the user. Allowing the user to add metadata in the form of annotations and saving the information in sessions are also related functions. The creation of temporary mappings helps users in testing their decisions and understanding the related consequences.

7.2.2 Feature evaluation and comparison with state of the art. Ganzha et al. [16] summarize the state of the art on tools for ontologies. From the 97 tools investigated, only 9 were deemed qualified for further evaluation; the rest of the tools were seriously limited or no longer maintained. We have followed a similar approach in choosing the most relevant tools for a comparative evaluation; additional details on them can be found in [13].

Specifically, the considered tools are COMA, LogMap, and RepOSE. COMA [31] is a framework that supports several matching algorithms. It is open source and provides a full GUI support for all its operations. The user assigns a confidence value to each matching axiom. LogMap [24] is an open-source tool that can match very large ontologies. LogMap is developed at the University of Oxford and the code is available as open source. RepOSE [22] is a framework for detecting and repairing ontology alignments. The system allows debugging missing and incorrect relations as well as mappings in a semi-automatic way.

We evaluated COMA, LogMap, and RepOSE – as well as our approach – according to the measures described at the beginning of the section. The results of the evaluation are summarized in Table 1.

Regarding system services, most of the considered systems ask for validation after running the matching algorithms (LogMap, RepOSE), while COMA allows validations both before and after; our approach allows user validation during the whole process. LogMap, COMA, and RepOSE employ some form of boundaries to present the mappings to the user. In terms of feedback propagation, interactions to support the seven visual information seeking tasks are provided to some extent by all systems except for LogMap. History and search are supported by all the considered systems, even though with some limitations. Moreover, all systems implement some sort of feedback. Revalidation is supported in RepOSE as well as in our approach. Regarding alignment representation, the systems normally represent the ontologies as trees or graphs; our approach leverages the tree representation as it is more widely used. On the other hand, few systems provide support for explaining the mappings. With the exception of LogMap, alignment interactions for accepting, rejecting, creating, and manually refining mappings are supported by the considered systems. However, none of them allows users to annotate mappings during the validation process; moreover, temporary mappings are only incidentally created in our approach.
As can be noted from Table 1, the features of the proposed system match those supported by the state of the art. In particular, our approach handles ALIGNMENT INTERACTION better than the others, even though it does not address ALIGNMENT PRESENTATION to the extent supported by the rest of the considered systems. Nevertheless, our approach is competitive with the state of the art across several dimensions.

8 CONCLUSION
This article introduced an extension of the W3C WoT architecture to increase semantic interoperability between IoT devices. Our proposed solution leverages user-assisted translations for Thing Description instances that cannot be otherwise interpreted by constrained devices according to the current W3C WoT architecture. Furthermore, we extended the Resource Directory specification so as to achieve better interoperability with the Thing Directory. We evaluated our solution based on a use case in logistic and transportation both quantitatively and qualitatively. Experimental results demonstrated that the system is scalable, thus suitable for scenarios involving a large number of devices. A feature comparison of the user translation process against the state of the art showed that our approach is competitive, and even better than existing solutions as for alignment interaction. Our proposed solution can be easily integrated with the W3C WoT architecture due to its compliance with the W3C and IETF standards. We actually identified a number of issues with the W3C WoT architecture and the Resource Directory specification while implementing our solution. We leveraged that opportunity to inform and contribute to the relevant standards in the W3C and IETF. As a future work, we are seeking to automate semantic translation by means of artificial intelligence. We are also particularly interested in enhancing our solution for scenarios where multiple Resource Directories and Thing Directories are distributed over the network.

9 ACKNOWLEDGMENTS
This work was partially supported by the Academy of Finland under grant number 299222. The authors would like to thank Nicklas Beijar, Alireza Ranjbar and Roberto Morabito for their suggestions.

REFERENCES


Received September 2018; revised September 2019; accepted December 2019