An Ontology-based Framework for Describing Discoverable Data Services

Xavier Oriol and Ernest Teniente

Universitat Politècnica de Catalunya, Barcelona, Spain
{xoriol,teniente}@essi.upc.edu

Abstract. *Note: This is a preprint version that deviates slightly from the published version of this paper.*
Data-services are applications in charge of retrieving certain data when they are called. They are found in different communities such as the Internet Of Things, Cloud Computing, Big Data, etc. So, there is a real need to discover how can an application that requires some data automatically find a data-service which is providing it. To our knowledge, the problem of automatically discovering these data-services is still open. To make a step forward in this direction, we propose an ontology-based framework to address this problem. In our framework, input and output values of the request are mapped into concepts of the domain ontology. Then, data-services specify how to obtain the output from the input by stating the relationship between the mapped concepts of the ontology.

Keywords: Data-service, Data-service discovery, Internet of Things

1 Introduction

Data-services are applications whose main concern is to provide data to their client applications. Data-services play a key role in areas like the Internet of Things (IoT), where smart objects obtain data from sensors (or other devices) and then make it available to others by means of data-services. Additionally, smart objects can also be interested in consuming data coming from other external data-services, so that they can use it to take their smart decisions. This idea is being currently exploted in several european projects like, for instance, the BIG IoT European project [1].

In IoT, smart objects should be as autonomous as possible due to the huge amount of devices and data that are constantly added. Hence, they should be able to offer/discover data-services on the fly, without human intervention. E.g., an autonomous car looking for parking in its current street should be able to automatically discover data-services retrieving available parking places of a street.

To make data-services discoverable, the usual strategy is to register data-services in some kind of service-broker, i.e., a marketplace where data-services are publicly offered [2]. Then, smart objects query the service-broker, and the service-broker is responsible to match the request with its data-services. How to perform this matching automatically is still an open problem in IoT [3].

In this sense, we propose a framework for specifying data-services so that they can be automatically discovered. To achieve it, we provide unambiguous descriptions of the data-services and the request, together with a mechanism capable of interpreting these descriptions and check whether they match. Our solution is grounded on ontology-based data integration and can be applied in the IoT context, altouh it can also be used in any other domain involving the discovery of applications retrieving data.

An ontology is a common set of terms (a vocabulary) with semantic relationhips among them (e.g. subclass/superclass relationships, etc.) that describes the real world. Figure 1 shows a UML ontology regarding parking concepts.

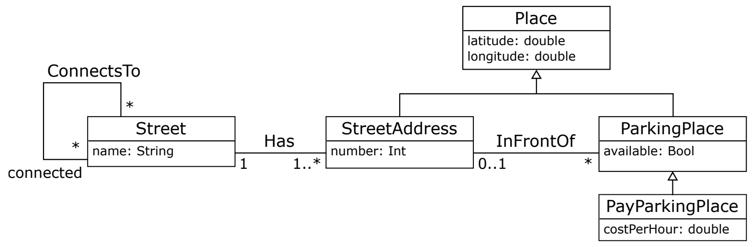


Fig. 1. Ontology for the parking domain in UML

Assume now a data-service receiving as input a *string* and returning a list of *paired numbers*. Clearly, with no description, a service-broker cannot determine what does the data-service compute since it cannot know what does such string stands for, what are the numbers describing, and which is the relationship between the string and the numbers. Hence, automatic discovery is not possible.

However, using the previous ontology we can state that the input are *Streets* (described by *street names* strings) and the output are *Parking places* (described by *latitude/longitude geolocalizations* paired numbers). These descriptions are somehow related to Semantic Web mark-ups provided by, for instance, *schema.org*, but these mark-ups alone are not enough since we need to know which is the relationship between the input *Street* and the output *Parking places* to know what the service is actually doing.

In our framework, we describe this relationship between the input and the output by defining a new association in the ontology linking their corresponding classes. The contents of this association is defined by means of a constraint (aka ontology axiom), which states what objects from the output are obtained from those in the input using the properties of the ontology. In this manner, we can specify that, for instance, given a street name, a data-service provides exactly the available parking places that are located in front of the street addresses that the street has. All these descriptions can be specified with ontological meta-concepts

(concepts/properties/axioms), thus, no need of special new meta-concepts is required (such as operation, or query).

In this way, our framework reduces the capability of storing data-service descriptions to that of storing ontologies, so, any existing tool for managing ontologies can be immediately used by the service-broker. Moreover, we also have that the problem of data-service/request matching is reduced to that of ontology reasoning. More precisely, to relation subsumption which is a very well-known and studied problem in the field of automated ontology reasoning [4,5].

We first define our framework in an abstract way, independently of the particular language used for specifying the ontology. Secondly, as a proof of concept that this framework can already be used in practice, we further develop it in the UML/OCL language with UML/OCL reasoning tools. We argue that using UML/OCL as our ontology languages are a natural choice since 1) they are widely used in software engineering, 2) they are expressive enough to specify all descriptions of our framework, and 3) there exist several reasoning tools that can be used for the service description/request matching. We finally comment on some experiments which show that these current reasoning tools have good response-times when applied in these settings.

In summary, the main contributions of the paper are: 1) *An ontology-based framework to describe discoverable data-services.* We define a set of descriptions to unambigously specify data-services and, thus, enabling its automatical discovey. These descriptions only involve meta-concepts already existing in modeling languages (e.g., relationships, axioms); 2) *UML/OCL suitability for the framework.* We show how to use UML/OCL for writing these descriptions, and show that UML/OCL already has tool support for solving the discoverability matching problem; and 3) *Efficiency evaluation.* We evaluate the efficiency for matching a service description/request using a UML/OCL reasoning tool.

It is worth to mention that this framework is an adoption of Semantic Web contributions (starting from local-as-view data integration [6]) to the context of Data-Services in IoT using software engineering languages (UML/OCL).

2 Basic Concepts

Data-service we refer as data-service to any application that receives some data as input and returns some data as output. During the paper, a data-service might be thought as a smart object providing information on demand.

Data-request we refer as data-request to the description of some data-service required by some application. During th paper, a data-request might be thought as the request from some smart object to find another smart object providing its interested information.

Data-service/request matching we refer as data-service/request matching to the problem of identifying whether some data-request coincides with the description of some existing data-service. For the sake of simplicity, we tackle the problem only in the semantic level. That is, we are not intended to match technological aspects of the service (e.g., SOAP/REST calls, XML/JSON formatting answers, etc.). Thus, we assume that the community using our proposed framework has an initial agreement about the communication technologies involved.

Service-broker we refer as service-broker to an application responsible of storing the data-service descriptions and resolving the data-request invoked by means of the data-service/request matching.

Ontology an ontology is a set of concepts and properties describing some real world domain. Ontology axioms (aka constraints) state conditions over these concepts/properties that hold in the real world. A typical axiom is the *isA* hierarchy between two concepts which states that instances of the first concept are also instances of the second. For our purposes, we require the ontology to include *primary key constraints*, i.e., to state which properties take unique values.

3 Framework essentials

In the following, we summarize how does the framework specify data-services in an abstract way (i.e., without bounding to any particular language), and show how to discover the service (i.e. how to match a data-service with a data-request).

3.1 Describing data-services

The framework assumes the use of one or several ontologies to describe data-services and application data-requests. These ontologies should be stored and maintained by the service-broker in which the data-services are registered.

The basic idea is that, if a service receives as input *Streets* and returns as output its available *Parkings places*, this service can be seen as a new relationship in the ontology between *Streets* and *Parking places*, whose contents is computed by the service. In general, a service that receives concepts $I_1, ..., I_M$ and returns concepts $O_1, ..., O_N$ can be described as a new M+N-ary relationship in the ontology.

The difficulty to describe data-service as a new ontology-relationship is that it does not receive/return objects, but values describing the real-world objects. Indeed, a data-service never returns a *Parking*, but some values describing it (e.g. a pair of integers representing its latitude/longitude). Note, in addition, that different data-services might describe the same object through different properties (e.g. a strings encoding the street address number of the parking). Thus, to specify the input and output of a data-service in terms of a relationship, we first need to map the input and output parameters to the ontology concepts.

Describing input/output concepts To identify the concepts handled by a data-service, we need to group its parameters and specify which concept of the ontology are they describing. Formally, given the set of parameters *PARAMS*, its powerset $\mathcal{P}(PARAMS)$, and the set of ontology concepts *CONCEPT*, this description corresponds to the input/output mappings:

$$Input_C : \mathcal{P}(PARAMS) \rightarrow CONCEPT \qquad Output_C : \mathcal{P}(PARAMS) \rightarrow CONCEPT$$

For instance, if some output paramaters *lat*, *long* describe an object *ParkingPlace*, we map $\{lat, long\}$ to *ParkingPlace*.

Moreover, since each parameter represents a specific property of the described concept, we map each parameter into the particular property it represents. Formally, given the parameters *PARAMS*, and the ontology properties *PROPERTIES*, this description corresponds to the maps:

$$Input_P : PARAMS \rightarrow PROPERTIES \qquad Output_P : PARAMS \rightarrow PROPERTIES$$

For instance, the output parameters *lat* and *long* would be mapped to the properties *latitude* and *longitude* from *ParkingPlace*.

The important thing here is that the set of parameters describing an input object should univocally determine the real-world object. In essence, this means that the description should include, at least, one primary key of its corresponding concept. For instance, if some data-service returns properties for describing *ParkingPlaces*, these properties should include, at least, a latitude/longitude pair or a street number address. Note that, without the primary key, we cannot identify which object from the real world do the properties belong to and, therefore, the data-service would not be able to identify the input objects for which computing the data required by the output.

The service-broker should be responsible of checking that the data-service is unambigously describing their objects. It is worth saying that this condition is not necessarily required by the output objects since we may be interested in anonymizing some kind of sensible data which should not appear in the output.

Describing the input/output relationship Once we know the input and output concepts to which the parameters refer to, we can define the input/output relationship computed by the service, i.e. the logic of the service. That is, the computation that the data-service performs in terms of a navigation through the ontology input concept to the ontology output concept.

This relationship is specified only in terms of the known basic concepts/properties of the ontology by means of an axiom (also called constraint). Specifically, this axiom defines the contents of the relationship in terms of contents of the input/output concepts and its related properties. Equivalently, we can see this relationship as *derived* from the rest of the ontology terms.

Formally, assuming that our data-service describes n input objects and m output objects, we have to define an axiom with the form:

$$R(I_1, ...I_n, O_1, ..., O_m) \leftrightarrow \phi(I_1, ...I_n, O_1, ..., O_m)$$

where R is the relation computed by the data-service, and ϕ is a statement (usually a first-order formula) that defines the contents of R in terms of the rest of ontology properties.

As an example, assuming a different predicate for each concept and relationship in our ontology, the input/output relationship of a service computing the *ParkingPlaces* that are *available* and located *infrontof* some *streetAddress* that has the input *Street* would be specified in logic by means of the following rule:

$$R(s,p) \leftrightarrow Street(s) \wedge HasStreetAdress(s,st) \wedge InFrontOf(st,p) \\ \wedge ParkingPlace(p) \wedge IsAvailable(p,a) \wedge a = \mathbf{true}$$

A service-broker might check whether an input/output relationship R description is correct by means of checking whether R is lively. We say that R is lively if there exists a state of the real-world domain I (that is, a valid instantiation of the whole ontology) in which R has some contents. Clearly, if no such real-world domain I exists, this means that this data-service is not outcoming any output from any input, which would mean that such data-service is useless. Checking relationship liveliness is a well-known ontology reasoning task [4,5].

Summary of a data-service description The description of a data-service is a tuple of the form: $< Input_P, Input_C, Output_P, Output_C, R(x) \leftrightarrow \phi(x)) >$

It is worth noting that our framework is only meant for data-services that retrieve data. Thus, it is not suited for services modifying this data. Moreover, our current proposal does not consider preconditions either, although they could be easily added at the ontological level as new ontological axioms. It is also important to remark that a data service should adapt to the terms in the ontology of the service-broker used (instead of being able to use its own ontology to define them). This limitation is caused because we need data-services and data-service requests to use the same ontological terms to enable its matching.

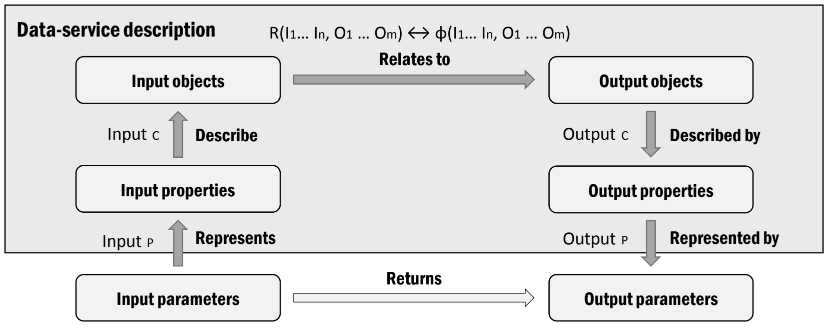


Fig. 2. Data-service description

Finally, we would like to stress that the descriptions proposed in our framework are unambiguous and capable of describing services considering several input/output parameters (even describing several input/output objects). The descriptions are unambiguous because, fixed a real-world state and some input values, there is only one valid collection of output values that satisfies them. Intuitively, this is so because all the semantic descriptions stated in Figure 2 can be composed into a unique function that converts input parameter values into input objects, input objects into output objects, and output objects as output values again. The description can handle services with an arbitrary number of input/output parameters as long as the language we use to state them allows n-ary associations. This is also a motivation for using UML as specification language (instead of OWL/RDF) since it directly supports n-ary associations.

3.2 Discovering data-services

To discover a data-service, a data consumer application should make a request to the service-broker so that it can check which data-services match the request.

The essential idea is that the request should be described in the same way as a data-service. I.e., with the input/output mappings that state how parameters describe concepts, and with a new ontology relationship and an axiom defining which input/output object relation is the requester looking for. Intuitively, a data request is described exactly as the hypothetical data-service it requires.

Then, the service-broker can match the request with its data-services by checking which relationships describing the services are subsumed by the relationship in the request. By subsumption, we mean that the contents of the relationship computed by the service is necessarily included in the contents of the one desired by the request. If the service relationship is subsumed by the request, this means that all the objects retrieved by the service are data that the requester wants to find. Thus, our proposal is semantic-based (i.e., independant from the particular syntactic way of writing the request/service in OCL).

Note that we only need to check if a data-service relationship is subsumed by a request relation if they agree on the related input/output concepts. Moreover, such discovery is performed only at the ontology level, without taking the underlying raw data into account.

Given a data-service and a data-request, the distinction between *how the service describes the objects* and *how the service relates the input/output objects* allows us to dinstinguish the following cases:

1. *Different relationships*: the data-service may compute an input/output relationship different than the one desired by the data-request. Thus, there is no match between the two. E.g.: a data service relates Streets with Parkings, but our data-request looks for a relationship between Streets and Streets.
2. *Same relationship, different descriptions*: it might happen that, at the ontology level, the data-service computes the same input/output relationship as the one requested, but the parameters given to describe these input/output are not the expected by the data-request. E.g.: a data service relates Streets with Parkings, and the data-request looks exactly for this relationship but, however, the service describes parkings by means of latitudes/longitudes and the requester expects street address numbers as output.
3. *Same relationship and same descriptions*: the data-service computes the input/output relation desired by the data-request and the parameters used are those it expects. E.g.: a data service relates Streets with Parkings, the data-request looks exactly for this relationship, and they both agree on the properties used to describe both concepts.

Obviously, when looking for some data-service matching a data-request, the best case is the third one. However, the second case should not be neglected. In fact, the second case should further dinstinguish whether the descriptions misagreement is because the service provides too much information (which might not be a problem at all), or because it does not provide all the desired information (which might be a problem, or not).

4 Using UML/OCL as ontology language

We show now how to use UML/OCL as the ontology language for the framework. UML/OCL is a good candidate language because it is well-known in the software engineering community, it has all necessary elements for encoding the framework descriptions, and there are already some reasoning tools that can be used to solve the matching problem.

In the following, we first discuss the usage of UML/OCL for describing the common ontology to be used by the service-broker. Then, we show how a data-service provider can use UML/OCL to describe a new service. Finally, we show how to discover a data-service by means of a UML/OCL reasoning tool.

4.1 Using UML/OCL to define the service-broker ontology

The service-broker should provide, at least, one common ontology so that all their published data-services are described over it. That is, all the data-services are described using the terms of the ontology provided by the broker, and assuming all the axioms defined in this ontology.

In Figure 1 we have shown a UML class diagram describing *Parking* concepts. In this ontology, all parkings are places with a latitude and a longitude, they can be available or not, and some of them have an associated cost per hour. These parkings may be in front of street addresses, which are also places with a latitude and a longitude. These street addresses belong to a street, and the streets can be connected to other streets.

To complete the ontology, the UML class diagram must be complemented with a set of OCL constraints stating conditions that the real-world satisfies. In the following, we show some OCL constraints stating that there are no two places with the same latitude/longitude, nor two streets with the same name, and that fixed one street, there are no two addresses with the same number:

```
context Street inv StreetPrimaryKey: Street.allInstances()->isUnique(name)
context StreetAddress inv StreetAddressPrimaryKey: StreetAddress.allInstances()
  ->forAll(a, b |a = b or a.number <> b.number or a.street <> b.street)
context Place inv PlacePrimaryKey: Place.allInstances()
  ->forAll(a, b | a = b or a.latitude <> b.latitude or a.longitude <> b.longitude)
```

4.2 Using UML/OCL to describe the service

Once we have the common ontology specified, we can use it to define a service along the framework defined in Section 3. That is, by means of describing how the parameters describe input/output concepts, and how the input and output concepts are related at the ontology level. These ideas are similar to [7], where they model queries as UML classes and inputs/outputs as associations to these classes.

Describing input/output concepts through UML/OCL To describe the mappings from the parameters to the concepts (i.e., the $Input_C/Output_C$ mappings), we define a new UML class and association for each described concept.

That is, if our parking service has an input parameter *name* that is mapped to the concept *Street*, we add a new UML class in the class diagram with an attribute name and associate this class to *Street*. Figure 3 illustrates this example in the UML class diagram.

In general, we may need to create several of these classes in case our input/output parameters describe several objects of different classes. To keep track of the newly added classes because of describing these mappings, we propose to use two new UML stereotypes: *ServiceInput*, and *ServiceOutput*.

Then, for describing the mappings to the properties (i.e., the $Input_P/Output_P$ mappings), we rely on the usage of OCL constraints. In particular, we define a new OCL invariant for each parameter that, in essence, equates its value to some attribute from the common ontology. Figure 3 also exemplifies how to do so.

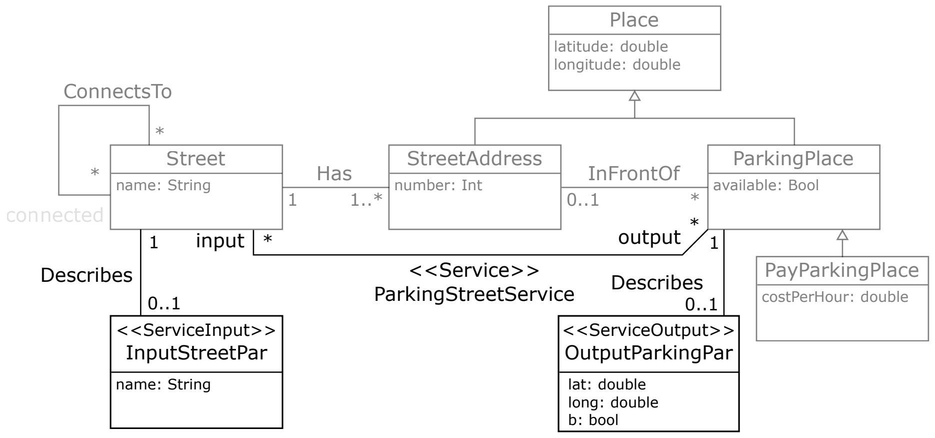
To correctly and unambigously describe the objects through the properties, two instances of parameters with exactly the same values should describe exactly the same object (e.g., two *InputStreetPar* with the name *Broadway* should describe exactly the same real-world *Street*: *Broadway*). Ensuring that two equal instances of parameters cannot describe (i.e., be associated to) the same object is a reasoning task that can be solved by many current UML/OCL reasoning tools [8,9,10,5].

Describing the input/output relation through UML/OCL Next, we add a new association in the UML class diagram to describe the input/output relationship computed by the service. This new association relates the described input and output concepts. Similarly as before, we propose distinguishing this new association using a UML stereotype: *Service*.

In addition, a new OCL invariant *InputOutputRelationship* establishing the contents of this association has to be specified, as shown in Figure 3. In our example, the invariant states that all the retrieved output parking places are available parking places from some street address of the given input street.

Intuitively, this invariant specifies the logics of the service in terms of the common ontology. Note that several equivalent ways of specifying the same logics may exist (i.e., by using different OCL operators, or using different OCL navigations, etc) and the service designer is free to choose the one he/she likes the most. The important thing is that all the equivalent ways of defining the same invariant will match with the same equivalent requests during service discovery.

Finally, it is worth noting that this way of specifying services scales for any number of input/output described objects (and not only one object for the input,



context Input **inv** DescribesStreet:
self.street.name = self.name
context Output **inv** DescribesParkingPlace:
self.parkingplace.latitude = self.lat and self.parkingplace.longitude=self.long and self.parkingplace.available = self.b
context Street **inv** InputOutputRelationship:
self.output = self.streetAddress.parkingPlace->select(p|p.available)

Fig. 3. Description of a data-service in the UML/OCL ontology

and one object class for the output). Indeed, if the service relates N objects from the input (e.g. Street and Place) to M output objects (e.g. StreetAddress and ParkingPlace), we only need to define an $(N + M)$-ary relation.

There is a special case when $N = 0$ and $M = 1$ because UML has no unary associations. However, this case can be handled by using a subclass. Indeed, consider that our parking service returns all parking places that are available (thus, no input is required, and the output is composed of only 1 object). In this case, our service is not relating any input to any output, but is considering a new class of parkings (i.e., the class of those parkings that it is retrieving -e.g., the available ones-). So, the service is specified by means of a subclass rather than an association. Similarly as before, this new subclass should be paired with the OCL constraint specifying its contents in terms of the common ontology.

4.3 Discovering the services

Now, the idea is that, when a service-broker receives a data-request, it must look for those data-services that match the request. More specifically, taking in account that services/requests logics are, at the end, defined through an association in the UML class diagram, it should look for those Services associations that are subsumed by the Request association.

There is a match between the data-service and the data-request when the association encoding the data-service R_S is subsumed by the association encoding the request R_R. Recall that, an association R_S is subsumed by R_R if the contents of R_S are, for any valid instance of the ontology, a subset of the contents of R_R.

Checking if R_S is subsumed by R_R can be done by analyzing whether the UML schema admits a finite valid instantiation I such that: 1) it satisfies all the integrity constraints of the UML class diagram, and 2) it contains some instance i of R_S not appearing in R_R. If such instantiation does not exist, this means that all instances of R_S are contained for sure in R_R and, thus, R_S is subsumed by R_R. On the contrary, if such instantiation I actually exists, I represents a state of the real world in which the data-service would provide some output i not desired according to the request.

As an example, consider two data-requests described with an association from *Streets* to *ParkingPlace* by means of the following invariants:

context Street **inv** Request1: self.output = self.streetAddress.parkingPlace->select(a|
not a.oclIsTypeOf(PayParkingPlace))
context Street **inv** Request2: self.output = self.streetAddress.parkingPlace
->union(self.connected.streetAddress.parkingPlace)

Intuitively, the first request is asking for parking places that are *free* (in the sense that you do not have to pay for their use), while the second one is asking for parking places in the given street and its connected streets. The first request does not subsume our data-service running example, but the second does.

The first data-request does not subsume our data-service because there might be a real-world state of the domain I in which the data-service computes some output undesired by the data-request. E.g., consider a real-world state I with a Street *Abbey Road* which contains a parking place in the geocoordinates *51.532, 0.177* such that is of kind *PayParkingPlace*. In this state of the domain I, our data-service would retrieve the previous parking, when the data-request is asking to avoid *PayParkingPlaces*. Thus, there is no match between the two.

On the contrary, the second data-request subsumes our service. Indeed, in any real-world state of the domain I, all the contents retrieved by the service are contents desired by the request, although our data-service is limited to only retrieve a subset of it (those that are available and in the given street, thus, ignoring the non-available parkings and the parkings from the connected streets).

In case we were only interested in data-services retrieving exactly all our requested contents, we could additionally check if the data-service also subsumes the data-request. Indeed, if they both mutually subsume each other, then, they are exactly requesting/computing the same relationship.

Reasoning association subsumption is a known problem in the conceptual schema reasoning field and it can be performed by several of their UML/OCL reasoners. E.g., we could use Alloy [8], USE[9], UMLtoCSP[10], or AuRUS [5]. Moreover, these tools are capable of computing the counterexample I that proves when a data-service would compute an undesired output for the data-request.

It is important to note that we are only checking whether the service request and the data-service offered coincide in terms of their intended logics (that is, their semantics). This checking does not take into account the different descriptions/representations that this services might have for their concepts. That is, it only checks whether the service/request matches the intended *Parking*, but it does not check if the way parkings are described also matches (which might be checked with a simple comparison between the actual/expected descriptions). This behavior is the one we already discussed while presenting the framework essentials in Section 3.2.

5 Experiments

To show the feasibility of our framework, together with the suitability of current UML/OCL reasoners to solve the matching problem, we have conducted some experiments with real data-services developed in the BIG IoT Project [1], where we participate and that has motivated the ideas proposed in this paper.

In particular, we have used our framework to describe 3 different traffic data-services from BIG IoT: a data-service for finding parking places of a given current location, a data-service for retrieving the current traffic status of a given street, and a third data-service for obtaining the average car speed of some predefined streets. These data-services cover several relevant aspects of our framework: the first one receives and retreives different concepts (i.e, it receives *Street* and returns *ParkingPlaces*), the second receives and retrieves the same concept (i.e., *Streets*), and the third has no input and only retrieves one concept (i.e., *Streets*).

To define the common ontology for these services we have departed from its current semantic anotation present in the BIG IoT project. In essence, BIG IoT tags the parameters of the services with the ontology property they represent (i.e., it defines the $Input_P/Output_P$ maps) using an extension of the *schema.org* ontology. Thus, we have built in UML/OCL the fragment of *schema.org* involved, and added the necessary extensions to define those concepts and properties which are not present in *schema.org*. Then, we have described all mentioned data-services in terms of it[1], using the constructions specified in Section 4.

Then, we have used the AuRUS[5] UML/OCL reasoner for evaluating the execution times for: 1) checking the correctness of the descriptions, 2) matching data-services with data-requests. We have run all experiments over a Windows 8 in an Intel i7-4710HQ up to 3.5GHz machine with 8GB of RAM.

Table 1 shows the results when checking the correctness of a data-service description. The first two checks are aimed at determining whether the input and output parameters, respectively, univoquely identify one ontology object. Moreover, we have checked the *liveliness* of the data-service relationship that allows computing the output from the input. That is, we have analyzed whether there is at least one possible real wold domain instantiation I where the described data-service would return, at least, one instance.

Table 2 shows the results for matching the services with some requests. In particular, we have evaluated the matching execution time against three kinds of data-requests: one that was equal to the intended service, another that was

[1] Descriptions available at `http://www.essi.upc.edu/~xoriol/obf-files/`

Table 1. Checking data-service correct descriptions execution times

	Checking input	Checking output	Checking service liveliness
ParkingService	531 ms	485 ms	546 ms
TrafficStatusService	515 ms	505 ms	531 ms
CarAverageSpeedService	-	516 ms	531 ms

subsumed by the service, and a third one that was subsuming the request. It is important to note that for the *CarAverageSpeedService* we could not define a request for exactly matching the data-service. This is because this service only returns data for those streets for which it has a sensor (which is not a selection criteria that can be described through the ontology).

Table 2. Matching data-services with data-requests

	Same request	Subsumed request	Subsumed service
ParkingService	203 ms	203 ms	156 ms
TrafficStatusService	125 ms	203 ms	125 ms
CarAverageSpeedService	-	218 ms	125 ms

As it can be seen, all the performed tests behaved satisfactorily and were executed in ms. It is also worth to remark that those tests aimed at performing data-service matching (i.e. the ones in Table 2) took always below 0.3 seconds. For these reasons, we believe that our proposed framework works properly for the automated discovery of data-services and it is also feasible in practice since it offers an efficient way to implement the data-service matching problem.

6 Related Work

Our work is based on the *local-as-view* approach to data integration as defined by [6]. In this approach, data-services and requests are defined as queries over a common ontology which describes the real-world. In this manner, the problem of data-service/request matching reduces to database query containment. However, and differently from [6], we distinguish among the concepts that the data-service deals with, from the properties used to describe these concepts. Thus, a data-request or data-service computing exactly the same conceptual relation will match in our framework, whereas they will not match in [6] if they use different properties to refer to the same concepts.

This distinction between concepts and their descriptions also appears in OWL-S [11]. OWL-S is, in essence, an ontology for describing data-services. The main idea is that any data-service can be described through instantiating the OWL-S concepts and properties. Other approaches working similarly are SAWSDL [12] and WSMO/WSML [13] according to a recent survey [14]. However, in the context of IoT, the difficulty to learn these languages has already been claimed to be a barrier to provide semantic descriptions for sensors [15].

We argue that these frameworks are difficult to use because they require instantiating very particular concepts/properties defined over them. In contrast, our framework is purely based on the observation that services can be seen as relationships. Thus, we can model services as any other relationship (without learning new vocabularies). In MOF terms, whereas approaches such as OWL-S or WSMO/WSML look at particular services as instances of the M0 level, M1 as the schema used to describe the service, and M2 as the language that enables so; in our framework, the data retrieved by the service belongs to M0, M1 corresponds to the particular service that relates the data, and M2 is the usual modelling language to specify the relationship (in our case, UML).

It can be argued that we could have chosen OWL instead of UML as our modelling language for defining the services. However, regarding the syntax, UML brings as natural support for specifying n-ary associations (which are mandatory to define services associating more than 2 objects through its input/output), while OWL does not. Under the point of view of semantics, UML/OCL is interpreted under the close-world assumption, thus considering only finite real-world states. In contrast, OWL is interpreted under the open-world assumption, thus considering infinite real-world states. As a consequence, it might happen that two services that compute exactly the same input/output relation are determined to be equal under a UML specification, but to be different in OWL. However, we think it is not realistic to consider infinite real-world states (states with an infinite number of *Parkings*, *Streets*, etc.). It is not clear how approaches based on pure OWL (such as SSWAP [16]) handle these cases.

Regarding the context of IoT, the current proposals that we know for semantically describing IoT sensors to enable their discovery are not fully automatic [17,1,18]. The approach presented in [17] is meant for helping human users to find their desired semantically-described sensors through a GUI and, thus, it implements manual discovery. In the case of [1], and despite pursuing automatic discovery, for the moment its unique way to describe the input/output relation is by means of a single tag annotation, so, its discovery process is reduced to checking the coincidence of such tag, which is a syntactic check rather than a semantic one. On the other side, the proposal in [18], although it can be fully automated, is in essence based on selecting sensors according to non-functional criteria (e.g. time-response, availability, cost) rather than semantics.

7 Conclusions and Future Work

We have presented a framework to specify IoT data-services so that they can be automatically discovered. Our framework specifies a data-service in terms of how it describes its input and output objects (that is, which properties of the objects does it speak about), and how it relates the input to the output objects.

We have seen that these descriptions can be written in UML/OCL. That is, both input and output can be modelled as UML classes related to the concepts they refer to, and the data-service logics can be seen as a new UML association between such classes whose contents are specified by means of a constraint. In this way, the service matching problem is reduced to relation subsumption.

In the experiments, we have demonstrated that our framework could be used to describe and discover three real IoT data-services that are currently being implemented in the BIG IoT European Project. Moreover, we have also shown that current UML/OCL reasoners can be used to solve the matching problem, and obtaining execution times below 0.3s. As future work, we plan to study how to deal with different data formats and how to chain/orchestrate several data-services to accomplish a data-request.

Acknowledgements Work partially supported by the BIG IoT project (European Commission's Horizon 2020 research and innovation programme) and by the Spanish Ministerio de Economía, Industría y Competitividad, under project TIN2017-87610-R.

References

1. Bröring, A., Schmid, S., Schindhelm, C., Khelil, A., Käbisch, S., Kramer, D., Phuoc, D.L., Mitic, J., Anicic, D., Teniente, E.: Enabling iot ecosystems through platform interoperability. IEEE Software **34**(1) (2017) 54–61
2. Papazoglou, M.P., van den Heuvel, W.: Service oriented architectures: approaches, technologies and research issues. VLDB J. **16**(3) (2007) 389–415
3. Perera, C., Zaslavsky, A.B., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. IEEE Communications Surveys and Tutorials **16**(1) (2014) 414–454
4. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artif. Intell. **168**(1-2) (2005) 70–118
5. Rull, G., Farré, C., Queralt, A., Teniente, E., Urpí, T.: Aurus: explaining the validation of UML/OCL conceptual schemas. Software and System Modeling **14**(2) (2015) 953–980
6. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases. (1996) 251–262
7. Olivé, A., Raventós, R.: Modeling events as entities in object-oriented conceptual modeling languages. Data Knowl. Eng. **58**(3) (2006) 243–262
8. Cunha, A., Garis, A.G., Riesco, D.: Translating between alloy specifications and UML class diagrams annotated with OCL. Software and System Modeling **14**(1) (2015) 5–25
9. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: Objects, Models, Components, Patterns - 49th International Conference, TOOLS. Proceedings. (2011) 290–306
10. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: 22nd IEEE/ACM Int. Conference on Automated Software Engineering (ASE 2007). (2007) 547–548
11. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with OWL-S. World Wide Web (3) (2007) 243–277
12. Kopecký, J., Vitvar, T., Bournez, C., Farrell, J.: SAWSDL: semantic annotations for WSDL and XML schema. IEEE Internet Computing **11**(6) (2007) 60–67
13. De Bruijn, J., Lausen, H., Polleres, A., Fensel, D.: The web service modeling language wsml: an overview. In: European Semantic Web Conference, Springer (2006) 590–604

14. Klusch, M., Kapahnke, P., Schulte, S., Lécué, F., Bernstein, A.: Semantic web service search: A brief survey. KI **30**(2) (2016) 139–147
15. Song, Z., Cárdenas, A.A., Masuoka, R.: Semantic middleware for the internet of things. In: Internet of Things (IOT), IoT for a green Planet, Proceedings. (2010)
16. Gessler, D.D., Bulka, B., Sirin, E., Vasquez-Gross, H., Yu, J., Wegrzyn, J.: i-Plant SSWAP (simple semantic web architecture and protocol) enables semantic pipelines for biodiversity. Semantics for Biodiversity (S4BioDiv) (2013) 101
17. Perera, C., Vasilakos, A.V.: A knowledge-based resource discovery for internet of things. Knowl.-Based Syst. **109** (2016) 122–136
18. Perera, C., Zaslavsky, A.B., Christen, P., Compton, M., Georgakopoulos, D.: Context-aware sensor search, selection and ranking model for internet of things middleware. In: IEEE 14th Int. Conference on Mobile Data Management - Volume 1. (2013) 314–322