A Web Service Composition Method based on OpenAPI Semantic Annotations

Andrei Netedu[1], Sabin C. Buraga[1], Paul Diac[1], and Liana Ţucăr[1]

Alexandru Ioan Cuza University of Iaşi, Romania
{mircea.netedu, busaco, paul.diac, stefania.tucar}@info.uaic.ro

Abstract. Automatic Web service composition is a research direction aimed to improve the process of aggregating multiple Web services to create some new, specific functionality. The use of semantics is required as the proper semantic model with annotation standards is enabling the automation of reasoning required to solve non-trivial cases. Most previous models are limited in describing service parameters as concepts of a simple hierarchy. Our proposal is increasing the expressiveness at the parameter level, using inherited concept properties that define attributesThe paper also describes how parameters are matched to create, in an automatic manner, valid compositions. The composition algorithm is practically used on descriptions of Web services implemented by REST APIs expressed by OpenAPI specifications. Our proposal uses knowledge models to enhance these OpenAPI constructs with JSON-LD annotations in order to obtain better compositions for involved services.

Keywords: Web Service Composition, Semantics, JSON-LD, OpenAPI.

1 INTRODUCTION

In the current software landscape, *Web services* are the core elements of Service-Oriented Architectures (SOAs) [6] that has been already used for many years with vast popularity. A Web Service provides a straightforward functionality defined through a public interface. In the enterprise context, this interface was traditionally expressed in WSDL (Web Service Description Language) – a standardized XML (Extensible Markup Language) dialect. Nowadays, Web services are usually built according to the REST (REpresentational State Transfer) architectural style [8]. There are several pragmatic solutions able to describe their APIs (Application Programming Interfaces) by using lightweight formats such as JSON (JavaScript Object Notation), a well-known data-interchange format based on ECMAScript programming language. In order to compose REST-based Web services, the paper proposes a novel method able to select suitable services automatically by using a knowledge-based approach described in Section 2. Without actually executing services, valid compositions are automatically generated depending on concepts from an ontology that denotes the semantics of input parameters, properties, and output (the result). The proposed algorithm

– producing certain encouraging results – is presented, evaluated, and discussed in Section 3.

We think that our proposal improves productivity and reduces costs within a complex (micro)service-based system. Additionally, the involved knowledge could be easily described, in a pragmatic way, by JSON-LD semantic constructs augmenting the OpenAPI description of each involved service, in order to offer the proper support for intelligent business operations – see Section 4.

For a motivating case study, several experiments were also conducted by using the proposed algorithm. From a pragmatic point of view, the popular *schema.org* model was chosen to convey Web services conceptual descriptions augmenting input parameters and expected results. Our approach added a suitable conceptualization that was necessary to discover compositions on cases where it could not be possible before, such as in the study presented in Section 5.

The paper enumerates various related approaches (Section 6), and ends with conclusions and further directions of research.

2 PROBLEM DEFINITION

2.1 Preliminaries

A *Web service* represents a software system designed to support inter-operable machine-to-machine interaction over a network and can be viewed as an abstract resource performing tasks that form *a coherent functionality* from the point of view of provider's entities and requester's entities.[1]

From a computational point of view, a Web service is a set of related methods described by the same interface. This interface could be declared by adopting various specifications: WSDL 2.0 – a classical XML-based Web standard[2] – and OpenAPI 3.0 Specification – a modern solution declaring the public interface of a Web service (REST API) in different formats like JSON or YAML (Yet Another Markup Language).

We refer to a Web service as a *single method* or the minimal endpoint that can be accessed or invoked at a time using some values as parameters – i.e. having some prior knowledge. In our context, we also work only with "information providing" Web services, or *stateless services* that do not alter their state and are not sensitive to the outside world states, time or any external factors.

The main focus of the proposed method is on improving the semantic description of services motivated by the lack of means to express several composition techniques on previous models and software solutions in Section 6.

We enhanced the service composition problem by modeling semantics in the manner described below. It was inspired by our previous experience and the shortcomings we found on expressing certain natural cases of composition. More

[1] Web Services Glossary, W3C Working Group Note, 2004 – `https://www.w3.org/TR/ws-gloss/`

[2] Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007 – `https://www.w3.org/TR/wsdl20/`

precisely, it was not possible to describe service parameters with properties or any relations/interaction between these parameters. Adding the new elements to the problem definition is also done with inspiration from the data model used by popular ontologies such as *schema.org* [10].

Using this approach, we managed to fix the issues that appeared in examples where the previous model failed. Our main addition is consisting of concept properties and how they are used for composition, allowing interaction between concepts and their properties based on simple constructions that increase the expressiveness.

2.2 Proposed Formal Model

Service parameters are defined over a set of *concepts*. Let $\mathbb{C}$ be the set of all concepts that appear in a repository of services, that are all possible concepts or the problem universe. As in the previous modeling, the concepts are first organized by using the *isA* or *subsumes* relation. This is a binary relation between concepts and can be considered somewhat similar to the inheritance in object-oriented programming. If a concept c_a *isA* c_b then c_a can substitute when there is need of a c_b. Also, for any concept c in $\mathbb{C}$, there can be only one direct, more generic concept then c, i.e. we do not allow multiple inheritance. Obviously, *isA* is transitive and, for convenience, reflexive: c_a *isA* c_a. This implies that $\mathbb{C}$ together with the *isA* relation form a tree (a taxonomy) or, more generally, a forest (a set of taxonomies or a complex ontology).

However, the new element added to the problem are concept *properties*. Any concept has a set of properties, possibly empty. Each property p is a pair $\langle name, type\rangle$. The name is just an identifier of the property, and for simplicity, it can be seen as a string – for concrete cases, this identifier is actually an IRI (Internationalized Resource Identifiers)[3], a superset of URIs (Uniform Resource Identifiers). The type is also a concept from the same set of concepts $\mathbb{C}$ and can be considered as the range of a property.

From an ontological point of view [1], a property p is defined as a relation between Web resources. The values of a property are instances of one or more concepts (classes) – expressed by *range* property. Any resource having a given property is an instance of one or more concepts denoted by *domain* property.

Properties are inherited: if c_a *isA* c_b, and c_b has some property $\langle name_x, type_x\rangle$ then c_a also has property $\langle name_x, type_x\rangle$. For example, if an *apple* is a *fruit*, and *fruit* has property $\langle hasColor, Color\rangle$ expressing that *fruit* instances have a color, then *apple*s also must have a color.

It is important that property names do not repeat for any unrelated concepts. For any concepts that are in a *isA* relation, all properties are passed to the specialization one by inheritance. This restriction is only imposed to avoid confusion and does not reduce the expressiveness, as properties can be renamed.

Syntactically, to define a property, the following are to be known: its name, its type, and the most general concept that the property can describe. For example,

[3] IRI (Internationalized Resource Identifiers – `https://tools.ietf.org/html/rfc3987`

consider that the *hasColor* property can describe the *apple* concept, but also the more general *fruit* concept. If concept *fruit isA physicalObject*, the next more general concept than *fruit* – i.e., its parent in the concepts tree –, under the assumption that not all physical objects are colored, then we can say that *hasColor* can most generally describe *fruit*, but not any *physicalObject* or any other more general concept. However, it can describe other concepts in the tree, together with all their descendants. For simplicity, we consider further that all the properties are defined within $\mathbb{C}$, thus the concepts, *isA* relation and properties structure are in $\mathbb{C}$ – the particular ontological model.

A *partially defined concept* denotes a pair *(c, propSet)*, where *c* is a concept from $\mathbb{C}$ and *propSet* is a subset of the properties that *c* has defined directly or through inheritance from more generic concepts. At some moment of time (or in some stage of a workflow), a partially defined concept describes what is currently known about a concept. It does not refer to a specific concept instance, but rather generally to the information that could potentially be found for any instance of that concept.

A Web Service $\mathbf{w}$ is defined by a pair of input and output parameters: ($\mathbf{w}_{in}$, $\mathbf{w}_{out}$). Both are sets of partially defined concepts. All are defined over the same structure $\mathbb{C}$, so all service providers must adhere to $\mathbb{C}$, thus adding the requirement that $\mathbb{C}$ is publicly available and defined ahead of time. In order to be able to validly call a service, all input parameters in $\mathbf{w}_{in}$ must be known together with their specified required properties. After calling the service, all output parameters $\mathbf{w}_{out}$ will be learned with the properties specified at output.

Parameter matching. Let $\mathbf{P}$ be a set of partially defined concepts, and $\mathbf{w}$ = ($\mathbf{w}_{in}$, $\mathbf{w}_{out}$) a Web service. The set $\mathbf{P}$ matches service $\mathbf{w}$ (or, equivalently, $\mathbf{w}$ is *callable* if $\mathbf{P}$ is known) if and only if $\forall$ partially defined concept $\mathbf{pdc}$ = *(c, propSet)* $\in \mathbf{w}_{in}$, $\exists$ $\mathbf{p}$ = *(c_{spec}, propSuperSet)* $\in \mathbf{P}$ such that c_{spec} *isA c* and $propSet \subseteq propSuperSet$. We define the addition of $\mathbf{w}_{out}$ to $\mathbf{P}$: $\mathbf{P} \oplus \mathbf{w}$ as:

$$\left\{ \left(c, \left\{ p \middle| \begin{matrix} & c\ has\ p \\ \exists (c', propSet') \in w_{out} & and\ p \in propSet' \\ & c'\ isA\ c \end{matrix} \right\} \right) \middle| \nexists (c, pSet) \in P \right\} \bigcup$$

$$\left\{ \left(c, pSet \cup \left\{ p \middle| \begin{matrix} & c\ has\ p \\ \exists (c', propSet') \in w_{out} & and\ p \in propSet' \\ & c'\ isA\ c \end{matrix} \right\} \right) \middle| \exists (c, pSet) \in P \right\}$$

or the union of $\mathbf{w}_{out}$ with $\mathbf{P}$ under the constraint of $\mathbf{P}$ matching $\mathbf{w}$ (defined as parameter matching above). Also, by *c has p* we refer to the fact that property p is stated for c directly or by inheritance. $\mathbf{P} \oplus \mathbf{w}$ contains (1) new concepts that are in $\mathbf{w}_{out}$ and (2) concepts already in $\mathbf{P}$ possibly with new properties from $\mathbf{w}_{out}$ specified for corresponding concepts or their specializations.

In words, after a call to a service, all its output parameters are selected, and for each concept together with its selected properties *(c, propSet)* in $\mathbf{w}_{out}$, *propSet* is added to c, c's parent in the concepts tree or the ascendants until we reach the first node that gains no new information or the root. More precisely, for each p in *propSet* we add p to our knowledge base for c, for the parent of c, and so on until p is no longer defined for the node we reached. The node where this process stops can differ from one p property to another p' property, but once the process stops for all properties in *propSet* there is no need to go further.

Chained matching. Let $\mathbf{P}$ be a set of partially defined concepts and $(w_1, w_2, \dots, w_k)$ an ordered list of services. We say that $\mathbf{P} \oplus w_1 \oplus w_2 \oplus \dots \oplus w_k$ is a chain of matching services iff w_i matches $\mathbf{P} \oplus w_1 \oplus w_2 \oplus \dots \oplus w_{i-1}; \forall i = 1 \dots k$. This is the rather primitive model for multiple service calls, that is a requirement for defining the composition. For simplicity, we avoid for now more complex workflows that could handle parallel and sequential service execution constructs.

Web Service Composition problem. Given an ontology having a set of concepts $\mathbb{C}$ and a repository of Web services $W = (w_1, w_2, \dots, w_n)$, and two sets of partially defined concepts **Init** and **Goal**, all defined over $\mathbb{C}$, find a chain of matching services $(w_{c1}, w_{c2}, \dots w_{ck})$ such that $(\emptyset, \mathbf{Init}) \oplus w_{c1} \oplus w_{c2} \oplus \dots \oplus w_{ck} \oplus (\mathbf{Goal}, \emptyset)$.

The $(\emptyset, \mathbf{Init})$ and $(\mathbf{Goal}, \emptyset)$ are just short ways of writing the initially known and finally required parameters, by using mock services.

We can also imagine (**Init**, **Goal**) as a Web (micro-)service – in this context, the problem requires finding an “implementation” of a Web (micro-)service using the services available in a certain development environment (e.g., a public or private repository).

3 AUTOMATIC SERVICE COMPOSITION

3.1 Algorithm Description

The proposed algorithm is intended to describe a generic solution that generates a valid composition. Tough it considers some basic optimizations like special data structures and indexes, there are many ways in which it can be improved, so we shortly describe some of them after the basic algorithm description.

In a simplified form, considered main entities have the following structure:

```
class Concept {      // full or partial type
    String name;     // a label
    Concept parent; // isA relation
    Set<Property> properties; } // proper and inherited
class Property {
    String name;     // a label
    Concept type; } // property's range
class WebService {
    String name;
    Set<Concept> in, out; } // I/O parameters
```

Global data structures that are most important and often used by the algorithm are presented below:

```
Set<Concept> C; // knowledge: concepts, isA, properties
Set <WebService> webServices; // service repository
```

```
WebService Init, Goal; // user’s query as two fictive services
Map <Concept,Set<Property>> known;
// partial concepts: known.get(c) = concept’s known properties
Map <Concept,Map<Property, Set<WebService>>> required;
// .get(c).get(p) services that have property p of C in input
Map <WebService, Map<Concept, Set<Property>>> remaining;
// .get(w).get(c) = properties of C necessary to call W
Set<WebService> callableServices; // with all input known
```

The algorithm described next uses the above structures, and is composed of three methods: initialization, the main composition search method which calls the last (utility) method, that updates the knowledge base with the new outputs learned form a service call.

Several obvious instructions are skipped for simplicity like parsing input data and initializing empty containers.

```
void initialize() { // read problem instance
 Init.in = Goal.out = ∅; webServices.add(Goal);
 for (WebService ws : webServices) {
  for (Concept c : ws.in) {
   for (Property p : c.properties) {
    required.get(c).get(p).add(ws);
    remaining.get(ws).get(c).add(p);
}}}} // container creation skipped
```

After reading the problem instance, the described data structures have to be *loaded*. **Init** and **Goal** can be used as web services to reduce the implementation size, if they are initialized as above. Then, for each parameter in service’s input, we add the corresponding concepts with their specified properties to the *indexes* (maps) that efficiently get the services that have those properties at input and the properties that remain yet unknown but required to validly call a service.

```
List<WebService> findComp(WebService Init, Goal) {
 List<WebService> composition; //result
 callService(Init); // learn initial

 while (!(required.get(Goal).isEmpty() ||
          callableServices.isEmpty())) {
  WebService ws = callableServices.first();
  callableServices.remove(ws);
  composition.add(ws);
  callWebService(ws);
 }
 if (remaining.get(Goal).isEmpty()) { return composition; }
 } else { return null; } // no solution
}
```

The main method that searches for a valid composition satisfying user’s query is *findComp()*. The result is simplified for now as an ordered list of services. As long as the **Goal** service is not yet callable, but we can call any other new service, we pick the *first* service from the *callableServices* set. Then we simulate its call, basically by learning all its output parameter information, with the help of *callWebService()* method below. We add the selected service to the composition and remove it from *callableServices* so it won't get called again. If *callableServices* empties before reaching the **Goal**, then the query is unsolvable.

```
void callWebService(WebService ws) {
 for (Concept c : ws.out) {
  Concept cp = c; // concept that goes up in tree
  boolean added = true; // if anything new was learned
  while (added && cp != null) {
   added = false;
   for (Property p : c.properties) {
    if (cp.properties.contains(p)&&!known.get(cp).contains(p)) {
     added = true; known.get(cp).add(p); // learn p at cp level
     for (WebService ws: required.get(cp).get(p)) {
      remaining.get(ws).get(cp).remove(p);
      if (remaining.get(ws).get(cp) .isEmpty()) {
       // all properties of cp in ws.in are known
       remaining.get(ws).remove(cp); }
      if (remaining.get(ws). isEmpty()) {
       // all concepts in ws.in known
       callableServices.add(ws); }
   }}}
   cp = cp.parent;
}}}
```

When calling a Web service, its output is learned and also expanded (we mark as learned properties for more generic concepts). This improves the algorithm's complexity, as it is better to prepare detection of newly callable services than to search for them by iteration after any call. This is possible by marking in the tree the known properties for each level and iterating only to levels that get any new information, as higher current service's output would be already known.

The optimization also comes from the fact that all services with inputs with these properties are hence updated only once (for each learned concept and property). As it gets learned, information is removed from the *remaining* data structure first at the property level and then at the concept level. When there's no property of any concept left to learn, the service gets callable. This can happen only once per service. The main loop might stop before reaching the tree root if at some generalization level, all current services' output was already known, and this is determined by the *added* flag variable.

3.2 Possible Improvements

One important metric that the algorithm does not consider is the size of the produced composition. As can be seen from the overview description above, the solution is both deterministic and of polynomial time complexity. This is possible because the length of the composition is not a necessary minimum. Finding the shortest composition is `NP-Hard` even for problem definitions that do not model semantics. The proposed model introduces *properties*; this addition does not significantly increase the computational problem complexity. Even if the shortest composition is hard to find, there are at least two simple ways to favor finding shorter compositions with good results.

One is based on the observation that when a service is called, it is chosen from possibly multiple callable services. This choice is not guided by any criteria. It is possible to add at least a simple *heuristic score* to each service that would estimate how useful is the information gained by that service call. Also, this score can be updated for the remaining services when information is learned.

Another improvement is based on the observation that services can be added to the composition even if they might produce no useful information – there is no condition check that they add anything new, or the information produced could also be found from services added later to the composition. To mitigate this, another algorithm can be implemented that would search the resulting composition *backward* and remove services that proved useless in the end. Both of the above improvements can have an impact on the running time as well, as the algorithm stops when the goal is reached.

3.3 Empiric Evaluation

To assess the performance on larger instances, a random test generator was built.

The generator first creates a conceptual model with random concepts and properties and then a service repository based on the generated ontology. Service parameters are chosen from direct and inherited properties. In the last step, an ordered list of services is altered by rebuilding each service input. The first one gets its input from **Init**, which is randomly generated at the beginning. Each other service in order gets its input rebuilt from all previous services outputs, or from valid properties of generalizations of those concepts. Finally, **Goal** has assigned a subset of the set of all outputs of the services in the list. The total number of services and the list size are input parameters for the generator. The intended dependency between services in the list is not guaranteed, so shorter compositions could potentially exist.

Table 1. Algorithm run times and resulting composition size on random generated instances.

ontology size (#classes + #props.)	repository size: #services	run time in seconds	result composition size: #services	dependency list size: #services
10 (5 + 5)	10	0.002	**3**	5
20 (10 + 10)	20	0.003	**4**	10
50 (30 + 20)	20	0.007	**12**	20
20 (10 + 10)	50	0.011	**6**	20

Table 1 shows the algorithm performance. The first two columns briefly describe the input size, by the number of concepts and properties in the generated ontology and total number of services in the repository. The column *result composition size* measures the length of the composition found by the algorithm.

The last column, *dependency list size*, measures the length of the composition generated by the tests generator algorithm. The *dependency list* constitutes a valid composition, hidden within the repository and may contain useless services as the *dependency* is not guaranteed.

4 EXTENDING OPENAPI WITH JSON-LD

Another aim of this research is to show how current OpenAPI specification[4] and our proposed extension to the JSON-LD model can be used for automatic Web service composition.

4.1 From Formalism to Semantic Descriptions

As a first step, we applied the above mathematical model and algorithm for a set of Web services defined with the help of OpenAPI specification expressed by JSON constructs.

OpenAPI specification is used to describe Web services (APIs) aligned to the REST (REpresentational State Transfer) architectural style [8] and defines in a standardized manner a meta-model to declare the interfaces to RESTful APIs in a programming language-agnostic manner.

These APIs correspond to a set of Web services that forms a repository. For our conducted experiments, we considered an API as a collection of services where each different URI path of the API denoted a different service – this can also be useful in the context of microservices. Thus, we can group services based on information related to the location of the group of services. In practice, this is useful as generally a Web server would likely host a multitude of Web services.

We used OpenAPI specification to describe the input and output parameters of each different service. Those parameters are Web resources that represent, in our mathematical model, partially defined concepts. OpenAPI also helps us match parameters on a syntactic level by specifying the data types of the parameters.

Example. As a real-life case, the public REST API (Web service) provided by Lyft[5] is described. For the operation of getting details about an authenticated user (`GET /profile`), the result – i.e., *w.out* in our mathematical model and, in fact, output data as a JSON object composed by various properties such as *id*: string (authenticated user's identifier), *first_name*: string, *last_name*: string, *has_taken_a_ride*: boolean (indicates if the user has taken at least one Lyft ride).

Each property can have as a result – the *range* mentioned in Section 2 – a datatype denoted by a concept, according to the proposed mathematical model. The Web resources processed by a service are then linked to assertions embedded into JSON-LD descriptions. JSON-LD model supports machine-readable annotations of involved properties for a given JSON object. All JSON-LD statements

[4] OpenAPI Spec – `https://github.com/OAI/OpenAPI-Specification`
[5] Lyft API – `https://developer.lyft.com/docs`

could be converted into RDF (Resource Description Framework) or OWL (Web Ontology Language) [1] to be further processed, including automated reasoning.

For this example, the *first_name* property of the returned JSON object could be mapped – via JSON-LD annotations – to the *givenName* property of *Person* concept (class) defined by the schema.org conceptual model or by FOAF vocabulary[6]. Additionally, the output parameter – i.e., the returned JSON object – could be seen as an instance of *Person* class. This approach could be further refined, if a *Customer* and *Driver* class are defined as a subclass of *Person* concept. The *subclass* relation from the (onto)logical model – formally written as $Customer \sqsubseteq Person$ – is equivalent to the *isA* relation of the mathematical formalism presented in Section 2. Using this taxonomy of classes, the algorithm could select the Web service as a candidate solution for Web service composition.

Secondly, we chose the JSON-LD model to describe the entities used by a group of services. For each different Web resource specified in the OpenAPI description of considered Web services, a corresponding semantic description could be stated. This semantic description attaches to each Web resource a URI specifying a concept in an ontology. Similarly, for each property of the corresponding Web resource.

4.2 Generic Approach

Generally, considering an OpenAPI specification stored in a JSON document:

```
{ "paths": {
    "/resource": {
      "get": {
        "operationId": "service",
        "parameters": [ {
          "name": "parameter",
          "in": "query", "schema": { "type": "string" } } ],
        "responses": {
          "200": {
            "description": "Success",
            "schema": { "$ref":"#/definitions/Response"
     }}} /* a response object composed by properties and
         values, e.g. { response: string } */
}}}}
```

an abstract JSON-LD annotation has the following form, where the context is a knowledge model used to denote Web service entities and the *Concept1* and *Concept2* classes are used for each parameter instead of a generic JSON datatype (in this case, a string).

```
{ "@context": "http://ontology.info/",
  "@id": "parameter", "@type": "Concept1" // w.in
  "response": { "@type": "Concept2" } // w.out
}
```

[6] FOAF (Friend Of A Friend) Specification – http://xmlns.com/foaf/spec/

Also, a convenient mapping (a-priori given or automatically generated) between JSON datatypes (string, number, boolean, object, array) and ontological concepts could be attached as meta-data for the considered set of Web services in order to facilitate the matching process. This enhancement is inspired by ontology alignment strategies [22]. The mapping itself could be directly expressed in JSON-LD via `@context` construct used to map terms (in our case, datatype names and/or property names) to concepts denoted by URIs. The concepts, properties, restrictions, and related entities (such as individuals and annotations) form the knowledge base – usually, specified by using OWL and RDF [1].

5 CASE STUDY: A TRANSPORT AGENCY

To illustrate the benefits of our approach, we have considered the following example, according to the problem definition in Section 2. Also, we have used concepts from schema.org. The services' interfaces are stored in an OpenAPI compliant document. The resources (service parameters) are described via JSON-LD by using the method exposed in Section 4.

The case study specifies the car company operating processes via web services. The car company services are simplistic and perform small tasks that can significantly benefit from a web service composition solution. Describing the services using OpenAPI and JSON-LD, the approach showcases how easy it is to represent complex relations between resources by combining object-oriented concepts in both structure and semantics. On top of this, because the REST services are similar on an architectural level, the solution guarantees that the situation presented in the case study can easily be extended and applied in real-world scenarios. Our study contains six services, several resources, but also the query to be solved by a composition of all six services in the correct order. We are supposing that a customer needs a *vehicle* to transport a given *payload*. This person knows his/her current *GeoLocation(latitude, longitude)* and a time frame *Action(startTime, endTime)* in which the transport should arrive. The **Goal** is to obtain the *Action(location)* where the *vehicle* will arrive. The six services – each of them implementing a single operation – are specified below:

```
getCountryFromLocation  in  = GeoLocation(lat,lon)
                        out = Country(name)
getTransportCompany  in  = AdministrativeArea(name)
                     out = Organization(name)
getClosestCity  in  = GeoLocation(lat,lon)
                out = City(name)
getLocalSubsidiary  in  = Organization(name), City(name)
                    out = LocalBusiness(email)
getVehicle  in = Vehicle(payload), LocalBusiness(email)
            out= Vehicle(vehicleIdentificationNumber)
makeArrangements
   in = Vehicle(vehicleIdentificationNumber),
        Organization(name,email), Action(startTime,endTime)
   out = Action(location)
```

In OpenAPI terms, a HTTP GET method is defined to obtain a JSON representation of the desired Web resource, for each *getResource* operation – i.e. using `GET /country` with *GeoLocation* as input parameter and *Country* as output. Without JSON-LD constructs, these parameters have regular JSON datatypes like string or number. As defined by schema.org, *LocalBusiness*[7] *isA Organization* – or, equivalent, in (onto)logic terms: *LocalBusiness* $\sqsubseteq$ *Organization*. Similarly, *Country isA AdministrativeArea*. A valid composition satisfying the user request can consist of the services in the following order: **Init** $\rightarrow$ *getCountryFromLocation* $\rightarrow$ *getTransportCompany* $\rightarrow$ *getClosestCity* $\rightarrow$ *getLocalSubsidiary* $\rightarrow$ *getVehicle* $\rightarrow$ *makeArrangements* $\rightarrow$ **Goal**. The order is relevant, but not unique in this case. This can be verified by considering all resources added by each service and also by the use of the *isA* relation. For example, *LocalBusiness(email)* can be used as *Organization(email)*.

The Java implementation of the algorithm from Section 3 reads the OpenAPI specification document describing the above scenario and finds the highlighted composition based on schema.org conceptual model. We also validated the OpenAPI and JSON-LD files using available public software tools included in Swagger Editor[8] and JSON-LD Playground[9].

6 RELATED WORK

Several approaches [3,16,18] were considered to tackle the problem of Web service composition in the case of "classical" specification of Web services by using the old SOAP, WSDL (Web Service Description Language), and additional WS-* standards and initiatives [6]. Also, several semantic-based formal, conceptual, and software solutions are proposed by using DAML-S and its successor OWL-S[10] service ontologies (in the present, they are almost neglected by the current communities of practice) and language extensions to Web service descriptions.

Various initiatives, methods, and software solutions could be mentioned:

– A mathematical model of the semantic Web service composition problem considering AI planning and causal link matrices [12].
– A linear programming strategy for service selection scheme considering non-functional QoS (Quality of Service) attributes [4].
– Automated discovery, interaction, and composition of the Web services that are semantically described by DAML-S ontological constructs [23].
– A process ontology (called OWL-S) to describe various aspects of Web services using SOAP protocol [15].
– A service composition planner by using hybrid artificial intelligence techniques and OWL-S model [11].

[7] LocalBusiness, a particular physical business or branch of an organization – `https://schema.org/LocalBusiness`
[8] Swagger Editor – `http://editor.swagger.io/`
[9] JSON-LD Playground – `https://json-ld.org/playground/`
[10] OWL-S: Semantic Markup for Web Services – `https://www.w3.org/Submission/OWL-S/`

– A hybrid framework which achieves the semantic web service matchmaking by using fuzzy logic and OWL-S statements [7].
– An automated software tool using MDA (Model-Driven Architecture) techniques to generate OWL-S descriptions from UML models [24].
– Various efforts for annotating SOAP-based Web service descriptions by using semantic approaches – e.g., SAWSDL (Semantic Annotations for WSDL and XML Schema) – within the METEOR-S system [21].

In contrast, there are relatively few recent proposals focused on resolving the problem of automatic service composition in the context of the new pragmatic way of describing the public REST APIs by using OpenAPI specification. Several solutions and tools are covered in [9] and [13]. From the modeling service compositions perspective, our formal model presents several similarities to the solutions proposed by [2] and [5].

Concerning enhancing Web services with semantic descriptions [26], several recent initiatives considering OpenAPI-based approaches are focused on:

– Extending the OpenAPI specification with a meta-model giving developers proper abstractions to write reusable code and allowing support for design-time consistency verification [20]. This approach is not using any semantic descriptions and does not provide JSON-LD-based support for reasoning.
– Adopting fuzzy inference methods to match services considering QoS metrics [27], or denoted by OpenAPI specifications [17].
– Generating, in an automatic manner, the suitable GraphQL-based wrappers for REST APIs described by OpenAPI documents [28].
– Using the RESTdesc[11] for service composition and invocation processes in the context of the Internet of Things – e.g., smart sensors [25].
– Capturing the semantics and relationships of REST APIs by using a simplified description model (Linked REST APIs), to automatically discover, compose, and orchestrate Web services [19].

7 CONCLUSION AND FUTURE WORK

The paper focused on REST-based Web services composition by using a straightforward method that adopts a conceptual approach for modeling semantics of (micro)service parameters. Starting with a formal model described in Section 2, an automatic service composition algorithm was proposed and evaluated – see Section 3.

To prove the feasibility of the proposed formalism, we extended OpenAPI description of stateless services with JSON-LD constructs, in order to conceptually explain the involved entities (resources) of the service's interface. This new method was detailed in Section 4.

For practical reasons, we adopted the popular *schema.org* model to quickly determine taxonomic relationships between concepts according to the described

[11] RESTdesc – `http://restdesc.org/`

algorithm. This was exemplified by the case study presented in Section 5. Our approach is general enough to choose convenient ontologies for each specific set of composable (micro-)services. We consider that our proposal is also a suitable solution for service-based business Web applications deployed in various cloud computing platforms.

An alternative approach is to use the SHACL (Shapes Constraint Language) model[12] to specify certain restrictions on RDF and, equally, on JSON-LD data – this direction of research is to be investigated in the near future in order to provide support for accessing semantically enriched digital content [14].

We are aware that the proposed method presents several shortcomings – e.g., lack of support regarding service meta-data such as quality of service, various restrictions, work and data-flows, and others. These aspects will be considered, formalized, and implemented in the next stages of our research.

References

1. Allemang, D., Hendler, J.: Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL. Elsevier (2011)
2. Baccar, S., Rouached, M., Verborgh, R., Abid, M.: Declarative Web Services Composition using Proofs. Service Oriented Computing and Applications pp. 1–19 (2018)
3. Blake, M.B., Cheung, W., Jaeger, M.C., Wombacher, A.: Wsc-06: The web service challenge. In: The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06). pp. 62–62. IEEE (2006)
4. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L.: Flow-based service selection for Web service composition supporting multiple QoS classes. In: Web Services, 2007. ICWS 2007. IEEE International Conference on. pp. 743–750. IEEE (2007)
5. Cremaschi, Marco, D.P.F.: A Practical Approach to Services Composition Through Light Semantic Descriptions. In: European Conference on Service-Oriented and Cloud Computing. pp. 130–145. Springer (2018)
6. Erl, T.: SOA: Principles of Service Design. Prentice Hall (2007)
7. Fenza, G., Loia, V., Senatore, S.: A Hybrid Approach to Semantic Web Services Matchmaking. International Journal of Approximate Reasoning **48**(3), 808–828 (2008)
8. Fielding, R.T.: Chapter 5: REpresentational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures (Ph. D. Thesis) (2000)
9. Garriga, M., Mateos, C., Flores, A., Cechich, A., Zunino, A.: RESTful Service Composition at a Glance: A Survey. Journal of Network and Computer Applications **60**, 32–53 (2016)
10. Guha, R.V., Brickley, D., Macbeth, S.: Schema.org: Evolution of Structured Data on the Web. Communications of the ACM **59**(2), 44–51 (2016)
11. Klusch, M., Gerber, A., Schmidt, M.: Semantic Web Service Composition Planning with OWLS-xplan. In: Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web. pp. 55–62. sn (2005)

[12] Shapes Constraint Language (SHACL), W3C Recommendation, 2017 – `https://www.w3.org/TR/shacl/`

12. Lécué, F., Léger, A.: A Formal Model for Semantic Web Service Composition. In: International Semantic Web Conference. pp. 385–398. Springer (2006)
13. Lemos, A.L., Daniel, F., Benatallah, B.: Web Service Composition: a Survey of Techniques and Tools. ACM Computing Surveys (CSUR) **48**(3), 33 (2016)
14. Levina, O.: Towards a Platform Architecture for Digital Content. In: Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 – Volume 1: DCNET, ICE-B, OPTICS, SIGMAP and WINSYS. pp. 340–347. SciTePress (2018)
15. Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D.L., Sirin, E., Srinivasan, N.: Bringing Semantics to Web Services with OWL-S. World Wide Web **10**(3), 243–277 (2007)
16. Milanovic, N., Malek, M.: Current Solutions for Web Service Composition. IEEE Internet Computing **8**(6), 51–59 (2004)
17. Peng, C., Goswami, P., Bai, G.: Fuzzy Matching of OpenAPI Described REST Services. Procedia Computer Science **126**, 1313–1322 (2018)
18. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods. In: International Workshop on Semantic Web Services and Web Process Composition. pp. 43–54. Springer (2004)
19. Serrano Suarez, D.F.: Automated API Discovery, Composition, and Orchestration with Linked Metadata. Ph.D. thesis (2018)
20. Sferruzza, D., Rocheteau, J., Attiogbé, C., Lanoix, A.: Extending OpenAPI 3.0 to Build Web Services from their Specification. In: International Conference on Web Information Systems and Technologies (2018)
21. Sheth, A.P., Gomadam, K., Ranabahu, A.H.: Semantics enhanced services: Meteor-s, SAWSDL and SA-REST. Bulletin of the Technical Committee on Data Engineering **31**(3), 8 (2008)
22. Shvaiko, P., Euzenat, J.: Ontology Matching: State of the Art and Future Challenges. IEEE Transactions on knowledge and data engineering **25**(1), 158–176 (2013)
23. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated Discovery, Interaction and Composition of Semantic Web Services. Web Semantics: Science, Services and Agents on the World Wide Web **1**(1), 27–46 (2003)
24. Timm, J.T., Gannod, G.C.: A Model-driven Approach for Specifying Semantic Web Services. In: IEEE International Conference on Web Services. pp. 313–320. IEEE (2005)
25. Ventura, D., Verborgh, R., Catania, V., Mannens, E.: Autonomous Composition and Execution of REST APIs for Smart Sensors. In: SSN-TC/OrdRing@ ISWC. pp. 13–24 (2015)
26. Verborgh, R., Harth, A., Maleshkova, M., Stadtmüller, S., Steiner, T., Taheriyan, M., Van de Walle, R.: Survey of Semantic Description of REST APIs. In: REST: Advanced Research Topics and Practical Applications, pp. 69–89. Springer (2014)
27. Wang, P., Chao, K.M., Lo, C.C., Huang, C.L., Li, Y.: A fuzzy model for selection of qos-aware web services. In: 2006 IEEE International Conference on e-Business Engineering (ICEBE'06). pp. 585–593. IEEE (2006)
28. Wittern, E., Cha, A., Laredo, J.A.: Generating graphql-wrappers for rest (-like) apis. In: International Conference on Web Engineering. pp. 65–83. Springer (2018)