Received October 3, 2020, accepted October 26, 2020, date of publication October 30, 2020, date of current version November 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3034937

JSON-LD Based Web API Semantic Annotation Considering Distributed Knowledge

XIANGHUI WANG[1], (Member, IEEE), QIAN SUN[1], AND JINLONG LIANG[2]
[1]Department of Computer Science and Technology, Shandong Jianzhu University, Jinan 250101, China
[2]Information Center, Shandong Provincial Qianfoshan Hospital, First Affiliated Hospital of Shandong First Medical University, Jinan 250100, China

Corresponding author: Xianghui Wang (wxh_225@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61902221, in part by the Project of Shandong Province Higher Educational Science and Technology Program under Grant J18KA364, in part by the Doctoral Fund of Shandong Jianzhu University under Grant X19045Z, and in part by the Shandong Provincial Natural Science Foundation under Grant ZR2018MF012.

ABSTRACT Based on semantically annotated Web APIs, automatic Web API composition can be implemented easily. The operation can greatly improve efficiency of building a software system. However, in real world, semantic annotation for Web APIs will encounter various difficulties, because of their distribution and function diversity, such as disunited API description formats, response result with complex structure, shortage of business domain ontologies, semantic conflicts among distributed knowledge, and so on. To solve these difficulties, we propose a JSON-LD based Web API semantic annotation approach (JWASA). JWASA can assist professional developers to semi-automatically complete semantic annotation of Web APIs. A common Web API description ontology is firstly defined, including necessary vocabularies about invocation information, functional semantics, and non-functional features. Then, JWASA automatically converts a Web API description into a document in an united JSON format, and assist developers to semi-automatically embed semantic information of crucial elements of the API by means of a lightweight Linked Data format JSON-LD. Meanwhile, a semantic annotation specification is proposed to deal with various complex situations in Web API description, e.g: too many response parameters, no request parameters, etc. In addition, to improve efficiency of annotation, JWASA provides some extra operations, including automatic new ontology or vocabulary creation, automatic functional semantics extraction etc. Also, JWASA provides semi-automatically bridge rule generation algorithm, which can infer implied relationships among vocabularies (e.g: sub-class, super-class, equality). JWASA focuses on the semantic annotation of functionality of Web APIs, and can create effective semantic Web APIs for future API automatic composition. We implement a prototype system and carry out a series of experiments to evaluate JWASA on real Web APIs crawled from Internet. Experiments show that our approach is effective and efficient.

INDEX TERMS Web API, semantic annotation, JSON-LD, ontology creation, distributed knowledge.

I. INTRODUCTION

Under micro service architecture [1], a software system can be quickly constructed by integrating various existing Web APIs from different providers. Some of them are from open Web API platforms on Internet, such as programmableweb,[1] juhe,[2] jisuapi.[3] Others are developed by internal developers. Providers generally give descriptions about how to invoke Web APIs according to their own custom formats, because there is no widely accepted description language for Web APIs. With the increase of Web APIs, it is tedious and time-consuming for software system developers to manually find and integrate suitable APIs.

To lighten burden of developers, researchers provided some semantic annotation approaches for Web APIs [2], [3]. Based on semantically annotated Web APIs, developers can rapidly find suitable APIs and automatically integrate them with the help of service composition technologies [4]. In some approaches, specific semantic annotation languages are utilized to annotate semantics of crucial elements in HTML-based Web API description documents, such as

The associate editor coordinating the review of this manuscript and approving it for publication was Francesco Tedesco.

[1]https://www.programmableweb.com 2020-08-27
[2]https://www.juhe.cn/ 2020-08-27
[3]https://www.jisuapi.com/ 2020-08-27

SA-REST, Microdata, RDFa [5]–[7]. In other approaches, Web APIs are described by existing API specifications with a concise format, e.g: JSON, and custom semantic annotation formats are proposed to embed semantic information in original description documents [8]–[10]. However, in practice, existing approaches would encounter five realistic problems.

Firstly, multiple types of request parameters make it difficult to decide which parameters need to be annotated. From the perspective of necessity, the parameters may be required, optional, or having default value. Specially, some APIs have no request parameters, and they only provide all information about specific data. Semantically, some are business parameters, for instance, *City* and *Date* in *Query History Weather* API; others are result format parameters, e.g: *Page No.*, *Total pages*, etc. Secondly, response parameter descriptions from providers are incomplete, and some details still need to be parsed out of response result example in a text-based description document. Generally, the response results are in JSON format, and data types of parsed parameters are multiple, such as array, value, object, etc. Also, a response parameter may be parsed out of a leaf node in a deep level of the response results. All these factors determine that the parsing operation is complex. Thirdly, a common Web API semantic ontology, that mainly describes its invocation, function, QoS, faults etc., is required to implement automatic invocation, discovery and composition. Although, existing semantic annotation approaches provide some ontologies for Web APIs, they usually aren't incomplete. Some of them mainly describe request/response parameters [5], [6], and are short of description about other details, such as precondition, effects etc. Others are used to annotate traditional WSDL-based web services and aren't for Web APIs [11]. Fourthly, there are few available ontologies for specific domains. Lastly, semantic conflicts among distributed knowledge from multiple used ontologies can decrease recall ratio of discovery [12], such as different name synonymous.

To solve the problems above, we propose a JSON-LD [13] based Web API semantic annotation approach (JWASA). It converts a Web API description document in informal natural language into a document in an united JSON format, and can assist developers in semantically annotating elements in the JSON document according to JSON-LD format. JSON-LD is a lightweight Linked Data format, and is widely accepted by developers for semantically annotating JSON data. Semantic information for JSON data can be easily embedded in corresponding JSON document, and can be parsed out of the document by means of ready-made JSON-LD APIs. Therefore, JSON-LD is very suitable as a semantic annotation format of Web API documents in JSON format.

Furthermore, JWASA proposes a semantic annotation specification to deal with various real complex situations for Web API descriptions. During annotation, JWASA can automatically extract functional semantic information from semantically annotated JSON-LD documents, and automatically generate or modify relevant ontologies according to new vocabularies used by developers. Also, JWASA provides semi-automatic bridge rule generation algorithm to efficiently solve semantic conflicts among distributed knowledge from different ontologies.

The main contributions of this article are in the following folds:

1) A common Web API description ontology is created to provide various semantic vocabularies about Web APIs, and a novel lightweight JSON-based Web API description format is designed.
2) A semi-automatic JSON-LD based Web API semantic annotation approach is proposed, where a semantic annotation specification is designed and some efficient assist operations are provided including automatic ontology generation, bridge rule generation and functional semantics extraction.
3) A prototype system is implemented, and, on real Web APIs from Internet, a series of experiments are carried out to evaluate effectiveness and efficiency of our approach.

The remainder of this article is organized as follows. Section II introduces related works. Section III describes an overview of our semantic annotation approach. Section IV presents formal definitions of Web API and Common Web API Ontology. Section V presents implementation details of the approach. Section VI reports the empirical results. Section VII concludes the paper.

II. RELATED WORKS

In recent years, researchers proposed some semantic annotation approaches for Web APIs (or Web services). According to difference of Web API description formats, these approaches mainly are divided into three categories: WSDL-based, HTML-based, and API specification based.

A. WSDL-BASED APPROACH

Currently, available Web APIs most are restful style which is resource-oriented [14]. Traditional web service description language (WSDL) provides support for restful APIs in its version 2.0 [15]. Specific semantic annotation languages were used to annotate WSDL-based Web APIs, e.g, SA-WSDL [16], OWL-S [11], etc.

SA-WSDL implemented semantic annotation through adding extra XML attributes to current WSDL document and associated XML schema document. Literature [17] designed a semantic annotation tool for Web services based on WSDL2.0 and SA-WSDL. The tool could assist users to search semantic vocabularies and embed semantic information in current documents.

Different from SA-WSDL, OWL-S described semantics of every Web service through an independent XML document in OWL-S format. The document not only described functional semantics of a web service from four aspects: input, output, precondition, and effect (called IOPE), but also described other feathers, for instance, QoS (reliable, response time etc.)

and the category of a given service. These information were enough for automatic service composition.

In these WSDL-based approaches, Web API description documents and semantically annotated documents all are in XML format, and their syntax is more complex for providers and annotators than other popular formats, e.g: HTML and JSON. Therefore, in practice, WSDL format rarely is used to describe Web API by providers.

B. HTML-BASED APPROACH

Generally, providers offer human-readable Web API descriptions according to custom formats on their official websites. Therefore, descriptions for Web APIs are embedded in HTML documents.

Based on HTML documents, some lightweight semantic annotation languages were adopted to directly annotate crucial elements of Web APIs in these documents (e.g: access url, request method, request/response parameter etc.), such as Microdata, hREST, SA-REST, MicroWSMO and so on [6], [18], [19].

Literature [20] provided a meta model of restful API description, and implemented semantic annotation by means of Microdata. The model defined some vocabularies describing web services, including *WebService*, *WSResource*, *WSAction*, *WSParam* etc. Microdata is used to add additional semantics of existing data in a HTML document on the basis of the model. Literature [2] proposed a semantic annotation tool SWEET, which used MicroWSMO and hRESTs to implement annotation of Web APIs. The hRESTs was adopted to identifying service properties (e.g, service, method, input, output) by insert hREST tags in given HTML document; the MicroWSMO was used to annotate service properties with semantic information by means of existing domain ontologies. Identifying service properties from a HTML document is tedious, because they are put together with other irrelevant HTML elements. Hence, literature [3] gave an automated semantic annotation approach, which could automatically identify service properties in a HTML document, and then annotate them with hRESTs or SA-REST.

However, these approaches are limited when HTML documents for describing Web APIs are complex. For example, multiple Web APIs are described in a HTML page; Web API description contents aren't embedded directly into a HTML page, while they are obtained by asynchronous requests. In these situations, extra operation is expected to extract a valid HTML document for every Web API. Furthermore, most of them focused on annotation for request/response parameters, and ignored other functional semantics, especially precondition, effect and fault semantics of Web APIs.

C. API SPECIFICATION BASED APPROACH

In practice, various custom Web API description formats from different providers make it difficult for API automatic invocation and semantic annotation. Therefore, some concise and easy-to-use metadata formats to describe Web API are proposed, such as OpenAPI Specification (OAS),[4] RESTful API Modeling Language (RAML),[5] API Blueprint,[6] and so on.

In recent years, on description documents of Web APIs conforming to OAS, some semantic annotation approaches have been proposed. OAS defines a standard, programming language-agnostic interface description for REST APIs. An OAS description is a YAML or JSON document that consists in various elements for describing Web APIs and a list of tags used by the specification with additional metadata.

Literature [10] extended OAS description to add semantic annotation. New elements (*classAnnotation* and *propertyAnnotation*) were added in the description documents of Web APIs, and a specific process was designed to identify these annotation elements. Literature [8] and [9] utilized tag elements in OAS to add custom semantic information, and annotated description documents still conformed to OAS. These approaches semantically annotated Web APIs in custom formats, thus, extra parsing processes are necessary to turn semantic API description to semantic resource graph in RDF.

Considering that an OAS description document can be represented in JSON format, Literature [21] embedded semantic information in OAS description documents for Web APIs by mean of JSON-LD format [13]. JSON-LD is a lightweight Linked data format, and provides a series of special elements to semantically annotate a JSON document. Compared with previous custom semantic annotation formats, JSON-LD format is more mature because ready-made APIs can be used to parse semantics out of JSON documents.

These approaches only presented semantic annotation formats, but didn't illustrate concrete annotation specification. For instance, which parameters can be annotated when a lot of parameters are answered; how to annotate when a Web API can have multiple request alternatives.

D. OUR APPROACH

All the previous approaches default that the used domain ontologies always are available. However, in practice, two main problems may occur for domain ontologies. The first is that no suitable domain ontologies are available for some Web APIs, and the second is that some semantic conflicts among distributed knowledge in domain ontologies may occur. No solution about the two problems is mentioned in these approaches.

Our approach proposes a whole solution for Web API semantic annotation. It learns from the advantages and overcomes the shortcomings of the previous approaches. The approach newly describes a Web API with a custom united JSON format according to its original description document, and uses JSON-LD to embed semantic information in the JSON document of the Web API. The semantic information

[4]http://spec.openapis.org/oas/v3.0.3 2020-10-02
[5]https://raml.org/ 2020-10-02
[6]https://apiblueprint.org/ 2020-10-02

TABLE 1. Comparison among existing approaches and our approach.

Literature	API format	Serialization	Annotation format	Annotation mode	Annotation elements	Annotation specification	Ontology	Distributed knowledge
[17]	WSDL	XML	SA-WSDL	embedded	IO	no	available	no solution
[11]	WSDL	XML	OWL-S	independent	IOPE	no	available	no solution
[20]	HTML	XML	Microdata	embedded	IO	no	available	no solution
[2], [3]	HTML	XML	hRESTs& MicroWSMO	embedded	IO	no	available	no solution
[8]–[10]	OAS	YAML/JSON	OAS(extended)	embedded	IO	no	available	no solution
[21]	OAS	JSON	JSON-LD	embedded	IO	no	available	no solution
this paper	custom	JSON	JSON-LD	embedded	IOPE	specific	available& automatic generation	bridge rules&automatic inference

includes IOPE (just like in OWL-S), and other feathers (e.g: QoS and faults). Also, a semantic annotation specification is designed to effectively handle various complex annotation situations.

To solve shortage of domain ontologies and efficiently eliminate semantic conflicts, our approach supports the automatic creation of new ontologies and vocabularies during semantic annotation, and semi-automatic bridge rule inference among multiple domain ontologies. It uses six types of bridge rules defined in our previous work [22] to assist inference among domain ontologies. These types of bridge rules respectively are intoc, *ontoc*, *equalc*, *intor*, *ontor*, *equalr*. The first three types are used to define sub-class, super-class, or equality relationship among two concepts from different ontologies. And the last three types are for sub-property, super-property, or equality relationship among two properties (or predicates) from different ontologies. Generated ontologies and bridge rules are crucial elements for future composition of Web APIs. Table 1 compares existing main approaches and our approach from various perspectives.

Furthermore, our approach can semantically annotate various faults in Web API descriptions by means of three defined exceptions in our previous work [23]: *UnPre* (precondition unsatisfied), *UnExe* (execution failure), and *UnEff* (unexpected execution effects). For example, *no weather is return* is a fault description for Web API *Weather Query*, and then the fault can be annotated as *UnEff*.

III. OVERVIEW OF JWASA

In this section, an illustrative example is described to show semantic annotation process in JWASA, and then an architecture to implement JWASA is presented.

A. AN ILLUSTRATIVE EXAMPLE

In JWASA, the semantic annotation for a Web API follows five steps: collect description information of the API, newly describe the API in an united format, annotate semantics, update ontology, and generate bridge rules. The detail description in each step is shown in the following.

1) COLLECT DESCRIPTION INFORMATION OF THE API

On an API open platform (https://www.jisuapi.com/), there are about *130* types of APIs, and the number of APIs is about *300*. All APIs have their own description documents. The description information of *Weather Query* API on the platform is shown in Table 2.

TABLE 2. Description information of *Weather Query* API.

Item name	Item Content
Interface URL	https://api.jisuapi.com/weather/query
Return format	JSON,JSONP
Request method	GET POST
Request example	https://api.jisuapi.com/weather/query?appkey= yourappkey&city=real city name
Request parameters	*5* business parameters are shown: *city*, *cityid*, *citycode*, *location*, *ip*. The API can succeed to run when one of them is input. Each parameter includes four aspects: name, data type, isrequired, and meanings.
Response parameters	*71* business parameters are shown. For instance, weather base information (high/low temperature, weed speed, etc.), air quality index (*PM2_5*, *aqi* value, etc.), daily weather, hourly weather, etc. Each parameter item includes three aspects: name, data type, and meaning.
Response example	A JSON data shown in Fig. 1 where most parameters are omitted due to limited space.
Exceptions	*4* fault items on API level are shown. Each item presents exception code and related meaning. For example, exception code *202* means entered *city* doesn′ t exist.
Other features	collection number: *1915*, used number: *14060*

Specially, there are two types of request/response parameters in the API: format parameter and business parameter. Format parameters are used to specify invocation authorization and returned result format. For example, parameter *appkey* in the request example is a format parameter, and its value can be applied by users. The parameter is required for every invocation of the API. Furthermore, *status*, *msg*, *result* also are format parameters, and respectively represent execution status code, status text description, and response business data. The three parameters are all returned after every invocation of the API. Business parameters can really reflect business function of the API, for instance, *city*, *cityid* in request, *weather*, *winddirect*, *aqi* in response. The meaning of every business parameter can be found from request/response parameter list in the description document. However, the relationships among parameters at structure level only are shown in the response example (Fig. 1). For example, *pm2_5* is a property of *aqi*.

```
{
        "status": 0,
        "msg": "ok",
        "result": {
                "temp": "16",
                "windspeed": "14.0",
                ......
                "aqi": {   ......
                        "pm2_5": "23",
                        "aqiinfo": { ......
                                "color": "#00e400",
                        }
                    },
                "daily": [{    ......
                                "sunrise": "07:39",
                                "sunset": "18:09",
                                "night": { ......
                                      "templow": "9",
                                      "img": "1",
                                },
                                "day": { ......
                                     "temphigh": "18",
                                     "img": "1",
                                }
                        }],
                "hourly": [ {......}, {......}]
        }
}
```

FIGURE 1. A response example of *Weather Query* API.

2) NEWLY DESCRIBE THE API IN AN UNITED FORMAT

In JWASA, through parsing the JSON data in response example, the names of all parameters are described newly. New names can reflect structure information of relevant parameters, such as *result-weather*, *result-winddirect*, *result-aqi*, *result-aqi-pm2_5*, etc. In addition, complex data types are introduced to further illustrate the structure of relevant parameter, including Object, Array of value, and Array of object. For example, in Fig. 1, the data types of *result-aqi* and *result-daily* respectively are *Object* and *Array of object*.

It is noticed that description contents of Web APIs on other platforms are similar to that information in Table 2, except some individual details. For example, on Juhe platform, parameter *appkey* is named *key*, and parameter *status*, *msg* are respectively named *error_code*, and *reason*. Also, description details of request and response parameter lists are a little different.

To automatically invoke a Web API, it is necessary to provide an united description document format. In JWASA, a Web API is described in an united JSON format, where various well-defined common attributes are set. Fig. 2 shows the description document with the JSON format of *Weather Query* API. Here, *apiID* represents the unique identity given by current user; *reqexam*, *respexam*, *qoS* respectively represent request example list, response example list, and other features in Table 2. Meanings of other attributes are as their names suggest. Every parameter is expressed a JSON object mainly including three aspects: name (*paramname*), data type (*datatype*), and meaning (*pararemark*). In addition, every response parameter has a value example attribute (*valexmp*), which value is obtained from response example. The attribute can assist annotators to comprehend semantics of current parameter. Meanwhile, every invocation exception also is expressed a JSON object, and is described from three aspects: *errortype*, *errorcode*, and *errorcontent*. Specially, given description information in Table 2, the JSON document can be automatically generated by JWASA.

3) ANNOTATE SEMANTICS

The semantics of *Weather Query* API are directly annotated in its extended JSON document, shown in Fig. 3. Some JSON-LD attributes (*@context*, *@id*, *@type*) and other attributes related to semantics (e.g: *inputs*, *outputs*, *effects*, *preconditions*) are added in the document.

Here, The *@context* is used to introduce well-defined semantics mapping files (*jsonld* as suffix) and declare various name space prefixes. The *@id* is used to specify the URI of current API or request/response parameter. The *@type* is used to annotate semantics of current API or parameter. Specially, in JWASA, *@type* of all format parameters need to be specified, because the information is key for automatically invoking the API and parsing its response result. Only *@id* and *@type* of those crucial parameters, that may interact with other APIs, are specified.

Attribute *domains* and *function* respectively specifies the business domains and function description details of current API, and they can facilitate API classification. Attribute *inputs*, *outputs*, *preconditions* and *effects* provide IOPE features in functional semantics of current API. Based on these information, automatic composition of Web APIs can be implemented easily.

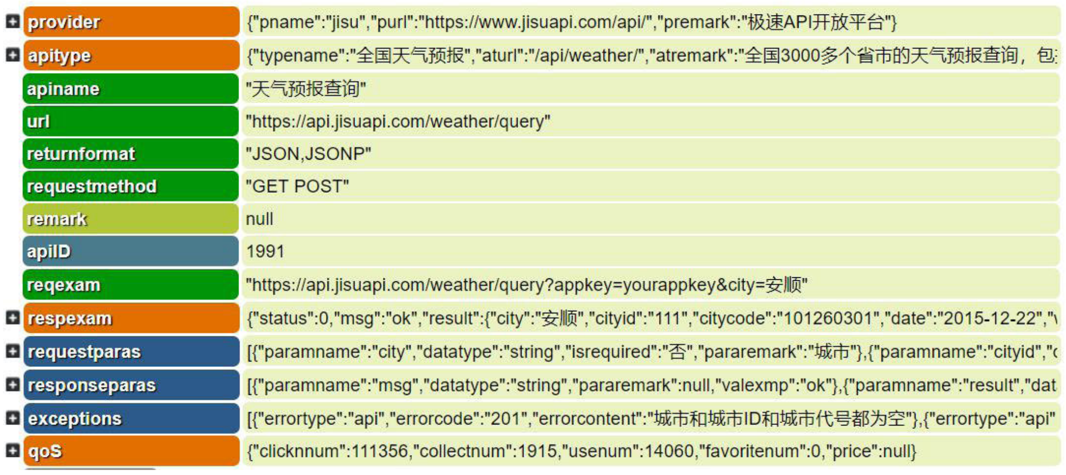
Furthermore, the exception with *api* type means the fault at business level, and it is necessary for automatic fault recognition to distinguish different exceptions by annotation. In JWASA, every exception with *api* type has attribute *@type*, and the value may be *UnPre*, *UnEff*, or *UnExe* [23] according to its *errorcontent* value.

In this step, an annotator needs to manually specify semantics about crucial request/response parameters, every business fault in exceptions, and to set preconditions and effects. The inputs and outputs will be automatically extracted by JWASA according to annotated information.

All in all, JWASA proposes a series of annotation specifications for Web APIs to make their semantics more complete and effective.

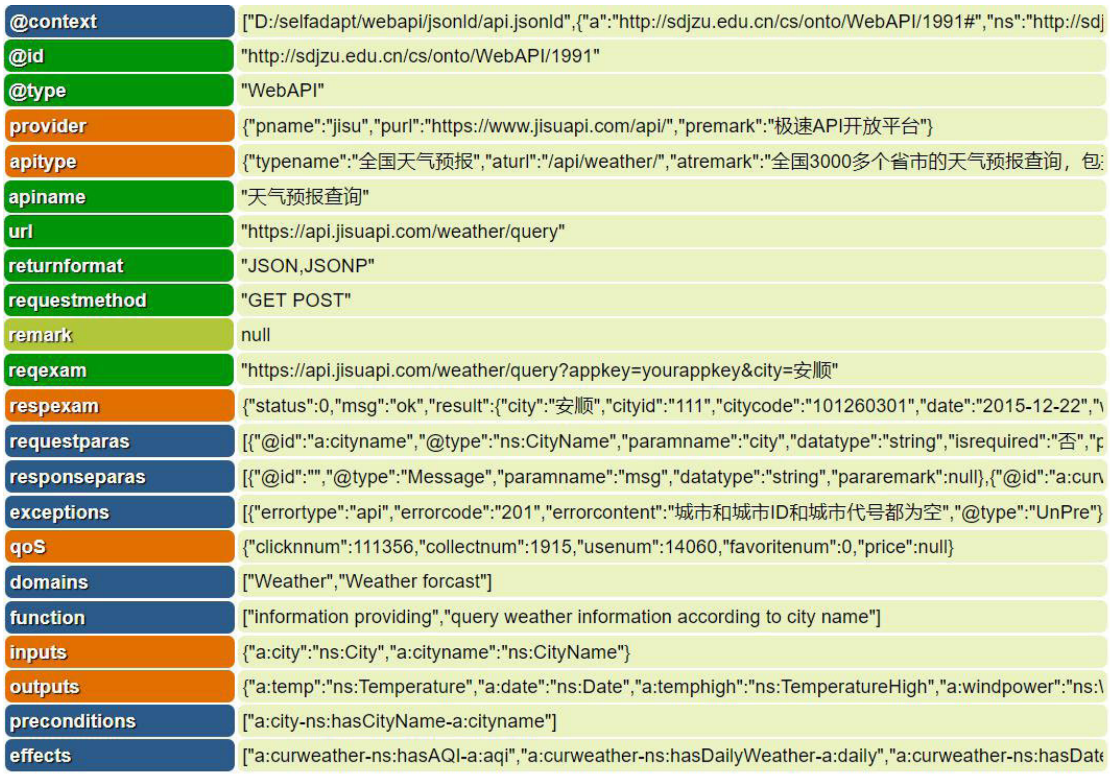
4) UPDATE ONTOLOGY

JWASA not only supports sharing concepts in existing ontologies, but also can dynamically create new ontologies and vocabularies during annotation. For example, the ontology for weather doesn't exist when an annotator annotates *Weather Query* API. He can declare a prefix *ns* for a new name space (http://sdjzu.edu.cn/cs/onto/weather), and then use a series of new vocabularies to annotate the API.



⊞ provider	{"pname":"jisu","purl":"https://www.jisuapi.com/api/","premark":"极速API开放平台"}
⊞ apitype	{"typename":"全国天气预报","aturl":"/api/weather/","atremark":"全国3000多个省市的天气预报查询，包
apiname	"天气预报查询"
url	"https://api.jisuapi.com/weather/query"
returnformat	"JSON,JSONP"
requestmethod	"GET POST"
remark	null
apiID	1991
reqexam	"https://api.jisuapi.com/weather/query?appkey=yourappkey&city=安顺"
⊞ respexam	{"status":0,"msg":"ok","result":{"city":"安顺","cityid":"111","citycode":"101260301","date":"2015-12-22","
⊞ requestparas	[{"paramname":"city","datatype":"string","isrequired":"否","pararemark":"城市"},{"paramname":"cityid","
⊞ responseparas	[{"paramname":"msg","datatype":"string","pararemark":null,"valexmp":"ok"},{"paramname":"result","dat
⊞ exceptions	[{"errortype":"api","errorcode":"201","errorcontent":"城市和城市ID和城市代号都为空"},{"errortype":"api"
⊞ qoS	{"clicknnum":111356,"collectnum":1915,"usenum":14060,"favoritenum":0,"price":null}

FIGURE 2. Description document of *Weather Query* API in JSON format.



@context	["D:/selfadapt/webapi/jsonld/api.jsonld",{"a":"http://sdjzu.edu.cn/cs/onto/WebAPI/1991#","ns":"http://sdj
@id	"http://sdjzu.edu.cn/cs/onto/WebAPI/1991"
@type	"WebAPI"
provider	{"pname":"jisu","purl":"https://www.jisuapi.com/api/","premark":"极速API开放平台"}
apitype	{"typename":"全国天气预报","aturl":"/api/weather/","atremark":"全国3000多个省市的天气预报查询，包
apiname	"天气预报查询"
url	"https://api.jisuapi.com/weather/query"
returnformat	"JSON,JSONP"
requestmethod	"GET POST"
remark	null
reqexam	"https://api.jisuapi.com/weather/query?appkey=yourappkey&city=安顺"
respexam	{"status":0,"msg":"ok","result":{"city":"安顺","cityid":"111","citycode":"101260301","date":"2015-12-22","
requestparas	[{"@id":"a:cityname","@type":"ns:CityName","paramname":"city","datatype":"string","isrequired":"否","p
responseparas	[{"@id":"","@type":"Message","paramname":"msg","datatype":"string","pararemark":null},{"@id":"a:curv
exceptions	[{"errortype":"api","errorcode":"201","errorcontent":"城市和城市ID和城市代号都为空","@type":"UnPre"}
qoS	{"clicknnum":111356,"collectnum":1915,"usenum":14060,"favoritenum":0,"price":null}
domains	["Weather","Weather forcast"]
function	["information providing","query weather information according to city name"]
inputs	{"a:city":"ns:City","a:cityname":"ns:CityName"}
outputs	{"a:temp":"ns:Temperature","a:date":"ns:Date","a:temphigh":"ns:TemperatureHigh","a:windpower":"ns:\
preconditions	["a:city-ns:hasCityName-a:cityname"]
effects	["a:curweather-ns:hasAQI-a:aqi","a:curweather-ns:hasDailyWeather-a:daily","a:curweather-ns:hasDate

FIGURE 3. Well annotated *Weather Query* API in JSON-LD format.

When the well-annotated JSON-LD file is saved, an new ontology for weather will automatically be generated and its name space is http://sdjzu.edu.cn/cs/onto/weather. Other Web APIs related to weather can use the ontology or add new vocabularies into this ontology during annotation.

5) GENERATE BRIDGE RULES

Due to diversity of business domains, multiple domain ontologies may be used in well-annotated Web APIs, such as weather ontology, vehicle ontology, book ontology, etc. Some semantic conflicts may occur for these ontologies. For example, *City*, *CityName* in ontology *o1* respectively represent a city object, and a city's name; *City* in ontology *o2* represents a city's name. Here, *City* in *o1* and *City* in *o2* use the same vocabulary, but they have different meaning; *CityName* in *o1* and *City* in *o2* are different vocabularies, but they have the same meaning. In JWASA, we employ bridge rules to solve these semantic conflicts. These rules specify parent-child and equality relationships among different vocabularies. For instance, rule ⟨*o*1, *CityName*, *equalc*, *o*2, *City*⟩ represents *CityName* in *o1* and *City* in *o2* are the equivalent classes. It is a tedious work for developers to manually create all bridge rules. To lighten their workload, our approach can semi-automatically generate all bridge rules among all ontologies.

After annotation, three types of semantic annotation results are generated: well-annotated JSON-LD files for Web APIs, all ontology files used during annotation, and all bridge rules. They are the crucial information for subsequent automatic service composition operations.

B. ARCHITECTURE OF JWASA

To efficiently complete various semantic annotation tasks above, a semantic annotation tool supporting JWASA is designed, and its architecture is shown in Fig. 4. The tool includes four main function modules: Original Information Collection, United Format Conversion, Semantic Annotation Core, and Ontology Management.

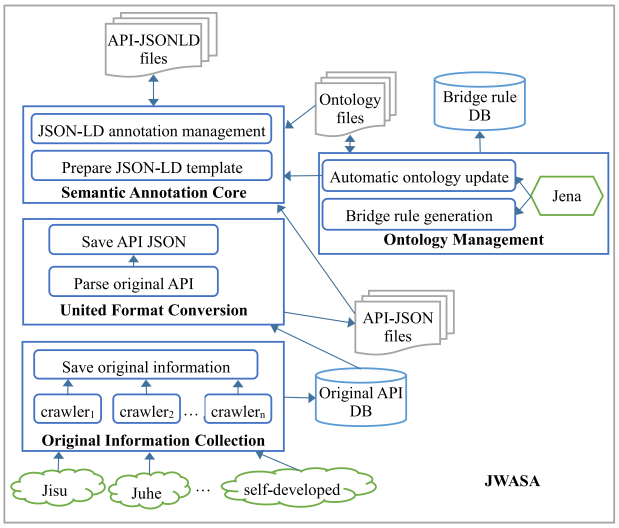


FIGURE 4. Architecture of a semantic annotation tool for JWASA.

1) ORIGINAL INFORMATION COLLECTION MODULE

This is used to collect available Web APIs from various API open platforms on Internet or self-developed Web APIs. For different API sources, the module provides different crawler agents. Also, developers can develop own crawler agents for special API sources. The caught original Web API description information will be stored in **Original API DB**.

2) UNITED FORMAT CONVERSION MODULE

This can read original information of APIs from **Original API DB**, and parse that information, especially parse response parameters out of corresponding response result data. Then, for every Web API, it assembles parsed information into an united JSON format, and stores that information in a JSON file. The file is called API-JSON file in the following.

3) SEMANTIC ANNOTATION CORE MODULE

This firstly uses **Prepare JSON-LD template function** to automatically insert JSON-LD elements and extra attributes about functional semantics into all API-JSON files, and then adopts **JSON-LD annotation management function** to assist developers specify semantics for crucial elements and to store annotation results in API-JSON files. In the following, a well-annotated API-JSON file is called API-JSONLD file. Meanwhile, **JSON-LD annotation management** also provides ontology search operations to help developers rapidly find suitable vocabularies from available ontology files. In addition, when new name spaces or new vocabularies are introduced during annotation, the function would invoke **Automatic ontology update function** in **Ontology Management module** to add or update ontology files.

4) ONTOLOGY MANAGEMENT MODULE

This includes two main functions based on **Jena** ontology tool kit.[7] **Jena** can support ontology CRUD and reasoning operations. **Automatic ontology update function** can automatically extract all semantic information from API-JSONLD files and add/update relevant ontology files. **Bridge rule generation function** can semi-automatically generate bridge rules among vocabularies from multiple ontology files, and store all bridge rules in **Bridge rule DB**.

Implementation details of main functions in JWASA will be shown in section IV and section V.

IV. WEB API AND COMMON WEB API ONTOLOGY

In this section, we will present formal definitions of Web API, semantic Web API and common Web API ontology. They are the basis of implementing JWASA.

A. WEB API AND SEMANTIC WEB API

In this article, a Web API means a web service that can receive HTTP protocol based requests and respond by HTTP protocol based answers. Its provider can provide complete description information about features of the API, including general features (e.g: provider, business category, quality of service), invocation crucial features, and invocation illustration features. Although Web APIs from different providers may have description documents with different formats, common features of Web APIs can be abstracted from their description documents. Here, based on abstracted common features, a Web API is formally defined.

Definition 1: Web API: A Web API is a tuple ⟨*apiname*, *Genf*, *InvCru*, *InvIll*⟩, where,

- *apiname* is the unique identifier of a Web API, and can be used to differentiate from other Web APIs from the same provider;

[7]https://jena.apache.org/ 2020-10-02

- *Genf* is a set of elements describing general features of the API, including *provider*, *apitype* (business category), *remark* (functional description text), and *QoS* (quality of service).
- *InvCru* is a set of elements describing crucial features about invocation details, including *url* (access URL), *returnformat* (return format), *requestmethod* (request method), *requestparas* (request parameters), *responseparas* (response parameters), and *exceptions*;
- *InvIll* is a set of elements illustrating how to invoke the API, including *reqexam* (request example) and *respexam* (response example).

In this definition, *provider* in *Genf* is a triple $\langle pname, purl, premark\rangle$, where components respectively represent provider's unique name, URL, and description text; *apitype* in *Genf* is also a triple $\langle typename, aturl, atremark\rangle$, where components respectively represent name, URL, and description text of API business category; *QoS* in *Genf* is a mutable tuple $\langle q_1, q_2, \ldots, q_n\rangle$, where $q_i (1 \leqslant i \leqslant n)$ is an QoS index, e.g: price, use number, etc. These indexes are provided by current Web API's provider, and different providers may provide different indexes.

Furthermore, *requestparas* and *responseparas* in *InvCru* both are sets including multiple parameters, and each parameter is modelled as a tuple $\langle paramname, datatype, isrequired, pararemark\rangle$, where components respectively represent parameter's unique identifier in current API, data type (e.g: string, integer, etc.), required or not, and meaning; *exceptions* in *InvCru* represents a set including various exceptions that may occur during invocation, and each exception is modelled as a triple $\langle errortype, errorcode, errorcontent\rangle$, where components respectively represent error type, error code returned, error message returned. Here, the value of error type has two kinds: *api* (application) and *sys* (system). The *api* means exceptions about business, e.g: some request parameter is invalid or no result is returned; the *sys* means exceptions about invocation permission of the API, e.g: application key is overdue or invalid.

In the following, given a Web API *wa*, we use *wa.XXX* to represent the value of its feature element *XXX*. For instance, *wa.requestparas* responses request parameter set of *wa*.

A semantic Web API is an extension of corresponding Web API through adding extra semantic features. These semantics features make the API easily be discovered, interactive and automatically invoked.

Definition 2 Semantic Web API: A semantic Web API is a semantically well-annotated Web API, and is expressed as a tuple $\langle webapi, semVs, funcSem, exSemV, exSem\rangle$, where,

- *webapi* is a Web API conforming to Definition 1, and provides original information from its provider;
- *semVs* is a set of vocabularies from business domains related to the API;
- *funcSem* describes functional semantics, and is expresses as a tuple $\langle inputs, outputs, preconditions, effects\rangle$. Here, assume that *inparams* and *outparams* respectively are sets of crucial parameters in *webapi.requestparas* and *webapi.responseparas*, then,
  - *inputs* represents input parameters with semantics, and is a mapping: $inparams \rightarrow semVs$;
  - *outputs* represents output parameters with semantics, and is a mapping: $outparams \rightarrow semVs$;
  - *preconditions* represents preconditions before invoking the API, and is a set of literals that describe relationships among parameters in *inparams* by means of vocabularies in *semVs*;
  - *effects* represents execution effects after invoking the API, and is a set of literals that describe relationships among parameters in *inparams* and *outparams* by means of vocabularies in *semVs*.
- *exSemVs* is a set of vocabularies related to exception semantics;
- *exSem* represent exceptions semantics, and is a mapping: $webapi.exceptions \rightarrow exSemV$.

Specially, given a semantic Web API, its component *funcSem* describes IOPEs features just like in OWL-S. Meanwhile, extra exception semantics are introduced to facilitate self-adaptation running of the API.

B. COMMON WEB API ONTOLOGY

Common features of Web APIs from different providers are defined in Definition 1. However, in practice, these features may have different names and presentation formats because of different providers. This hampers automatic discovery and interoperability among different Web APIs. To solve this problem, we design a common Web API ontology: ***WebAPIOnto***. It not only provides requisite vocabularies to describe common features of Web APIs in Definition 1, but also provides extra semantic vocabularies to describe function and exception semantics in Definition 2. The formal definition of ***WebAPIOnto*** is shown in Definition 2.

Definition 3 WebAPIOnto: ***WebAPIOnto*** is an owl ontology specially for Web API domain, and is expressed as a tuple $\langle WebAPI, Genc, Grdc, FuncSem, R\rangle$, where,

- *WebAPI* is an owl class that represents a Web API description document;
- *Genc* is a set of owl classes that represent concepts related to general information of a Web API, and describes what the API is;
- *Grdc* is a set of owl classes that represent concepts related to invocation detail of a Web API, and describes how to invoke the API;
- *FuncSem* is a set of owl classes that represent concepts related to functional semantics of a Web API, and describes what the API does;
- *R* is a set of owl object properties with one domain and one range, where every domain or range is an owl class in set $\{WebAPI\} \cup Genc \cup Grdc \cup FuncSem$, and each object property describes a relationship between concept for its domain and concept for its range.

In this ontology, the most core concept is *WebAPI*, and other concepts are used to illustrate various properties

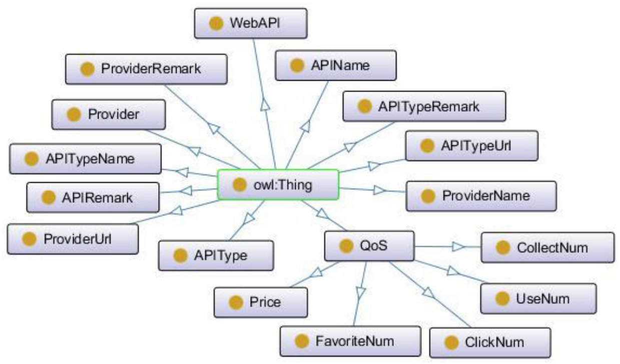


FIGURE 5. Concepts in component *Genc* of *WebAPIOnto*.

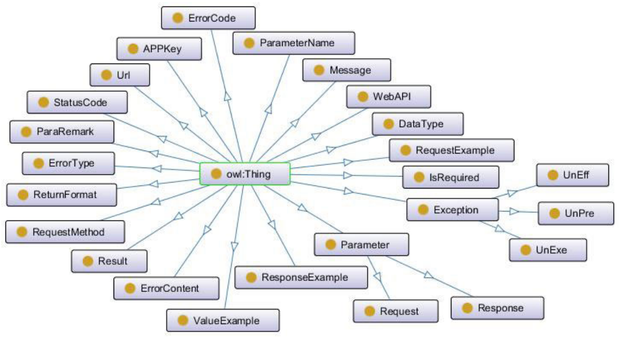


FIGURE 6. Concepts in component *Grdc* of *WebAPIOnto*.

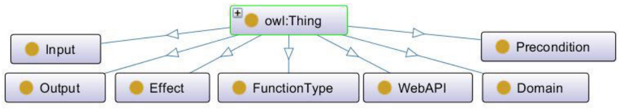


FIGURE 7. Concepts in component *FuncSem* of *WebAPIOnto*.

of *WebAPI*. Component *Genc*, *Grdc*, and *FuncSem* respectively provide concepts related to three types of features of a Web API. These concepts are respectively shown in Fig. 5, Fig. 6, and Fig. 7. Here, an arrow in these figures represents the concept at its tail is super-class of other concept at its head.

Concepts in *Genc* mainly describe a Web API's features in component *Genf*. Specially, common QoS indexes also are included in *Genc*, such as *CollectNum* (the number of collected by users),*ClickNum* (the number of clicked) and so on. Concepts in *Grdc* describe a Web API's features in component *InvCru* and *InvIll*. Concepts in *FuncSem* are mainly describe necessary IOPE components, business domain, and function category, and they are extra functional semantic vocabularies.

Meanwhile, all object properties in component *R* have unique names, which can directly illustrate relationship meanings. Object properties in ***WebAPIOnto*** are shown in Table 3, Table 4, and Table 5. Specially, meanings of concepts and properties are suggested by their names.

WebAPIOnto is a common Web API ontology, and can be used by any Web API provider to annotate their own description documents. In JWASA, it is used to semantically annotate various Web APIs crawled from Internet.

Specially, ontology ***WebAPIOnto*** provides vocabularies related to currently popular Web APIs as complete as possible. From the RESTful view, maturity of these Web APIs generally is at a lower maturity level [24]. Therefore, developers can extended this ontology as the maturity of Web APIs is raised in future.

TABLE 3. Object properties about concepts in component *Genc*.

NO.	Object Property	Domain	Range
1	hasAPIName	WebAPI	APIName
2	hasAPIDesc	WebAPI	APIRemark
3	hasProvider	WebAPI	Provider
4	hasAPIType	WebAPI	APIType
5	hasQoS	WebAPI	QoS
6	hasProviderName	Provider	ProviderName
7	hasProviderUrl	Provider	ProviderUrl
8	hasProviderDesc	Provider	ProviderRemark
9	hasAPITypeName	APIType	APITypeName
10	hasAPITypeUrl	APIType	APITypeUrl
11	hasAPITypeDesc	APIType	APITypeRemark

TABLE 4. Object properties about concepts in component *Grdc*.

NO.	Object Property	Domain	Range
1	hasUrl	WebAPI	Url
2	hasReturnFormat	WebAPI	ReturnFormat
3	hasRequestMethod	WebAPI	RequestMethod
4	hasRequestExample	WebAPI	RequestExample
5	hasResponseExample	WebAPI	ResponseExample
6	hasRequestParameter	WebAPI	Request
7	hasResponseParameter	WebAPI	Response
8	hasException	WebAPI	Exception
9	hasErrorType	Exception	ErrorType
10	hasErrorCode	Exception	ErrorCode
11	hasErrorContent	Exception	ErrorContent
12	hasAPPKey	WebAPI	APPKey
13	hasStatusCode	WebAPI	StatusCode
14	hasMessage	WebAPI	Message
15	hasResult	WebAPI	Result
16	hasParameterName	Parameter	ParameterName
17	hasDataType	Parameter	DataType
18	hasIsRequired	Parameter	IsRequired
19	hasRemark	Parameter	ParaRemark
20	hasValueExample	Parameter	ValueExample

TABLE 5. Object properties about concepts in component *FuncSem*.

NO.	Object Property	Domain	Range
1	hasDomain	WebAPI	Domain
2	hasFunctionType	WebAPI	FunctionType
3	hasInput	WebAPI	Input
4	hasOutput	WebAPI	Output
5	hasPrecondition	WebAPI	Precondition
6	hasEffect	WebAPI	Effect

V. IMPLEMENTATION OF JWASA

Given original information of Web APIs crawled from Internet, JWASA can generate corresponding semantic Web APIs. During generation, a series of manual and automatic operations need to be completed according to the architecture of JWASA (Fig. 4). To ensure the accuracy of semantics, developers need to manually identify functional semantics and exception semantics of these APIs. Except this, other operations can be completed automatically by means of designed algorithms. Therefore, JWASA is a semi-automatic semantic annotation approach. In this section, implementation details of JWASA will be presented.

A. AN UNITED JSON FORMAT FOR WEB API

In JWASA, Web APIs crawled from Internet are reorganized into documents in an united JSON format. The format is called API-JSON, and document in API-JSON format is described in Definition 3.

Definition 4 API-JSON Document: Given a Web API, its *API-JSON* document is in JSON format, where all features of the API are modelled as attributes, and each attribute is expressed a key-value pair. Here, except for features in all components of the API, it adds an extra attribute *apiID* as the unique identifier of current API.

Specially, in an API-JSON document, those features modelled as tuples are converted into attributes with JSON object values, and each component in a tuple is converted into one attribute of corresponding JSON object.

In practice, for a Web API, except *apiID* and *responseparas*, other features can be crawled from its official description document. An concrete example for an complete API-JSON document is shown in Fig. 2 (section III).

B. PARSE RESPONSE PARAMETER

A provider generally provides two parameter description lists: request parameter list and response parameter list. And parameters in the lists are described from four aspects: parameter name, data type, meaning, required. A real example for those parameter descriptions is shown in Table 6.

TABLE 6. An example for real parameter descriptions fragment.

List type	Pname	Data type	Meaning	required
request list	city	string	city name	no
	cityid	int	city ID	no
	. . .	. . .	. . .	. . .
response list	temphigh	string	high temperature	
	windspeed	string	wind speed	
	aqi	string	AQI index	
	pm2_5	string	PM2.5	
	. . .	. . .	. . .	

Generally, parameters in the two lists are described in a flatted format, that is, no structural relationships (e.g: object-property) among different two parameters are presented. In practice, request parameters of a Web API usually are flatted. Thus, users can directly specify value for every parameters in the request list to invoke a Web API. In API-JSON document, the request list is assigned to attribute *requestparas*. However, in most cases, there are some structural relationships between two parameters in the response list. For example, *aqi* is an object where multiple properties are included, here, *pm2_5* is one property of *aqi*. This kind of information is important for automatic interaction among multiple Web APIs.

```
Algorithm 1 ParseRespara
Input: examobj: a JSON object, resparas: an array including parsed parameters,
parentname: attribute name with value examobj
Output: resparas: updated resparas after parsing examobj
01. para ← a parameter called parentname from resparas, null is return if no
parameter is found
02. valuetype← the node type of examobj
03. IF valuetype is a value THEN
04.   IF para.datatype is null THEN
05.       realtype←real data type of examobj (string or integer)
06.       para.datatype← realtype
07.       para.valexmp← text value of examobj
08.   END IF
09. END IF
10.  IF valuetype is an object THEN
11.     IF para is not null THEN para.datatype← "object"
12.     FOR each attribute aname of examobj THEN
13.       avalue← value for attribute aname
14.       paraname← aname
15.       IF para is not null THEN paraname←parentname+"-"+paraname
16.       newpara ← a new parameter object
17.       newpara.paramname←paraname
18.       add newpara into resparas
19.       call ParseRespara(avalue,resparas,paraname)
20.     END FOR
21. END IF
22. IF valuetype is an array THEN // default that all elements in an array have
the same data type.
23.     enode← the first element in examobj
24.     evaluetype← the node type of enode
25.     IF evaluetype is a value THEN
26.       para.datatype← "array.value"
27.       para.valexmp← text value of enode
28.       RETURN
29.     END IF
30.     IF evaluetype is an object THEN
31.       para.datatype← "array.object"
32.       call ParseRespara(enode,resparas,parentname)
33.     END IF
34.      IF evaluetype is an array THEN// if an element in an array is also an
array, no further parsing continues.
35.       para.datatype← "array.array"
36.       para.valexmp← text value of enode
37.       RETURN
38.     END IF
39. END IF
```

FIGURE 8. Algorithm for parsing response parameter.

It is noticed that this information isn't shown in the response list. Therefore, extra operation is needed to obtain response parameters with structural relationship. Fortunately, attribute *respexam* records a response result in JSON object format, through analysing internal structure of the result, response parameters with structural relationship can be obtained. Specially, attribute *responseparas* in an API-JSON document will record all response parameters with structural relationships.

Here, we design an algorithm to automatic parse all response parameters out of attribute *respexam*, and its procedure is shown in Fig. 8. It deeply traverses an given JSON object by means of a recursive call. Except for the JSON object to be parsed(*examobj*), it also receives two input parameters: current parsed parameters (*resparas*), and attribute name with value *examobj* (parentname). Finally, it will return updated *resparas* with all parsed parameters out of *examobj*.

On the first invocation of ***ParseRespara***, *examobj* is the value of attribute *respexam* in current Web API, *resparas* has no element, and *parentname* is null. The algorithm firstly judges value type of *examobj*, and gives different parsing strategies for different types. When *examobj* is a value node (e.g: string or integer value), data type and value example attributes of the parameter called *parentname* will be assigned (lines *03-09*). When *examobj* is an object, data type of the parameter called *parentname* firstly is assigned *object*, then, for each attribute in this object, *ParseRespara* is called again to continue to parse value of the attribute (lines *10-21*). Specially, during each loop, a new parameter with structure relationship is created (lines *15-18*). When *examobj* is an array, its first element is parsed, and data type of the parameter called *parentname* is respectively assigned *array.value*, *array.object* or *array.array* according value type of this element (lines *22-39*). When the value type is an object, *ParseRespara* is called again to further parse the element (line *32*).

```
{"paramname":"msg","datatype":"string","pararemark":null,"valexmp":"ok"}
{"paramname":"result","datatype":"object","pararemark":null,"valexmp":null}
{"paramname":"status","datatype":"int","pararemark":null,"valexmp":null}
{"paramname":"result-aqi","datatype":"object","pararemark":"AQI index","valexmp":null}
{"paramname":"result-temp","datatype":"string","pararemark":"°C","valexmp":"16"}
{"paramname":"result-windspeed","datatype":"string","pararemark":"wind
speed","valexmp":"14.0"}
{"paramname":"result-daily","datatype":"array.object","pararemark":"daily","valexmp":null}
{"paramname":"result-daily-date","datatype":"string","pararemark":"date","valexmp":"2015-
12-22"}
{"paramname":"result-daily-day-img","datatype":"string","pararemark":"image
figure","valexmp":"1"}
{"paramname":"result-hourly","datatype":"array.object","pararemark":"hourly","valexmp":nu
ll}
{"paramname":"result-hourly-temp","datatype":"string","pararemark":"°C","valexmp":"14"}
```

FIGURE 9. **Some parsed response parameters for *Weather Query* API.**

According to this algorithm, some parsed parameters for the response example in Fig. 1 are shown in Fig. 9. Here, *paramname* and *datatype* can reflect current parameter's structure in the response result. The information can help users to directly obtain value of this parameter from corresponding response result, and to facilitate interaction with other Web APIs. In addition, after parsing, the attribute *pararemark* of every parameter also is set according to its meaning description in the response list. This process is easy, and aren't shown here.

C. SEMANTIC ANNOTATION SPECIFICATION

Given an API-JSON document, JWASA can semantically annotate the document and finally generate an extension version of this document: API-JSONLD document. Compared with the API-JSON document, the API-JSONLD document adds *3* JSON-LD elements to annotate semantic information (*@context*, *@id*, *@type*), and introduces *6* new attributes related to semantic features (*domains*, *function*, *inputs*, *outputs*, *preconditions*, and *effects*).

Element *@context* is an array including some information related to semantics, eg: defined mapping from attributes to semantic concepts, used name spaces etc.; element *@id* specifies the URI of current Web API or every parameter in request/response; element *@type* specifies semantics of current JSON document, parameters in *requestparas* and *responseparas*, or exceptions in *exceptions*.

Attribute *inputs*, *outputs*, *preconditions*, and *effects* are used to describe functional semantics of a semantic Web API. Attribute *domains* and *function* are added to identify concrete business category. Here, attribute *domains* includes vocabularies about business domain of current API; attribute *function* includes two elements: the first specifies function type of current API, and the second is a detailed text description about function of current API. Definition 5 shows the details about API-JSONLD document.

Definition 5 API-JSONLD Document: Given a semantic Web API *swa*, ***WebAPIOnto***, and a set of business domain ontologies *ontos*, and *ad* is the API-JSON document of *swa.webapi*. An API-JSONLD document for *swa* is an extension of *ad* through adding other components of *swa* and semantic information conforming to JSON-LD specification, where,

- ***WebAPIOnto*** provides semantic vocabularies about attributes related to features of *swa* and exception semantic vocabularies in *swa.exSemVs*, and is introduced by *@context* element;
- *ontos* provides functional semantic vocabularies in *swa.funcsemVs*, and is introduced by *@context* element;
- Mappings in *swa.funcSem* and *swa.exSem* are specified through introducing *@type* element in corresponding parameters or exceptions.
- Each parameter in *inputs* and *outputs* of *swa.funcSem* is uniquely identified through introducing *@id* element.

Specially, in an API-JSONLD document, both attribute *inputs* and *outputs* are JSON objects where each attribute represents a parameter. Each parameter is expressed as a attribute-value pair *pid:psem*, where *pid* and *psem* respectively represent parameter ID and its semantics.

During annotation, except for attribute *inputs* and *outputs*, values of other attributes need to be manually assigned by annotators. Values of the two attributes can be automatically generated according to annotated request/response parameters. An concrete example for an API-JSONLD document is shown in Fig. 3.

In every API-JSONLD document, except for JSON-LD elements, semantics of other attributes are declared by a mapping from attributes to concepts in ontology ***WebAPIOnto***. The mapping is saved to a JSON-LD file, and is introduced into the document by attribute *@context* (shown in Fig. 10(a)). Here, the first element in *@context* specifies the mapping file's location: *D:/selfadapt/webapi/jsonld/api.jsonld*, and Fig. 10(b) shows a fragment in this file. In Fig. 10(b), *@vocab* is used to declare default name space of semantic vocabularies in current document, and in the following annotation, no prefixes are required for vocabularies in the default name space. For example, *pname:ProviderName* means semantics of attribute *pname*

```
@context: [
        "D:/selfadapt/webapi/jsonld/api.jsonld",
        {
        "a":"http://sdjzu.edu.cn/cs/onto/WebAPI/1969#",
        "ns":"http://sdjzu.edu.cn/cs/onto/train#",
        "com":"http://sdjzu.edu.cn/cs/onto/jisucom#"
        }
]
```

(a) An example of @context in an API-JSONLD document

```
{
    "@context": {
            "@vocab": "http://sdjzu.edu.cn/cs/onto/WebAPI#",
            "provider": "Provider",
            "pname": "ProviderName",
            ......
            "requestparas": "Request",
            "responseparas": "Response",
            "paramname": "ParameterName",
            ......
            "errorcode": "ErrorCode",
            "errorcontent": "ErrorContent",
            ......
    }
}
```

(b) A mapping fragment from attributes to semantic concepts

FIGURE 10. Some fragments in an API-JSONLD document.

is *ProviderName* from *http://sdjzu.edu.cn/cs/onto/WebAPI#* (ontology ***WebAPIOnto***).

The second element in *@context* is an object which declares source paths of resources ("*a:*" as default prefix) and ontologies to be used in current document. Semantic annotators can modify these prefix names and their source paths or add new prefix definitions to introduce new ontologies. In Fig. 10(a), three source paths are declared, and respectively specify path of resources in current document (e.g: request/response parameters) and paths of two introduced ontologies.

Values for attribute *domain* and *function* are easily specified according to some text description from providers. Specially, for a Web API, its function type in attribute *function* may be *information providing*, *world altering*, or *dictionary service* according to concrete function effect. For example, *Weather Query* API is an *information providing* API, because it only query some relevant data; *Order Train Ticket* API is a *world altering* API, because the world status will be changed after this API runs; *Weather Query cities* API is a *dictionary service* API, because it can run without any request parameters and return all cities related to weather query. Based on response result of *Weather Query cities* API, according to a given city name, a user can find the city's ID, code etc. This is just like a dictionary.

After attribute *@context*, *domain*, and *function* of a template are specified, the core semantic annotation operation can start. The core of semantic annotation is to specify semantics of parameters in attribute *requestparas* and *responseparas*, and to declare literals in preconditions and effects. That information can directly reflect IOPE features of current Web API, and is important for following automatic composition. However, in real world, this annotation task will encounter the following five main problems, because of diversity of Web APIs.

(1) A lot of response parameters make annotation tedious and time-consuming. For example, *Weather Query* API has *3* format parameters (*msg*, *result*, *status*) and *71* business parameters. These *71* parameters describe weather information of today including various air quality indexes, hourly weather details, and daily weather details.

(2) Some Web APIs can receive different request parameters and return similar response. For example, *Weather Query* API can receive any one of *city*, *cityid*, *citycode*, *location*, and all return the weather information.

(3) Multiple parameters describing a business object aren't specially declared. This makes the expression of P/E (precondition/effect) difficult. For example, a Web API can obtain weather information according to two given GPS coordinate parameters: *lon*, *lat*. They respectively represent longitude and latitude of a location, but this relationship of the two parameters isn't clearly declared in response parameters of the API.

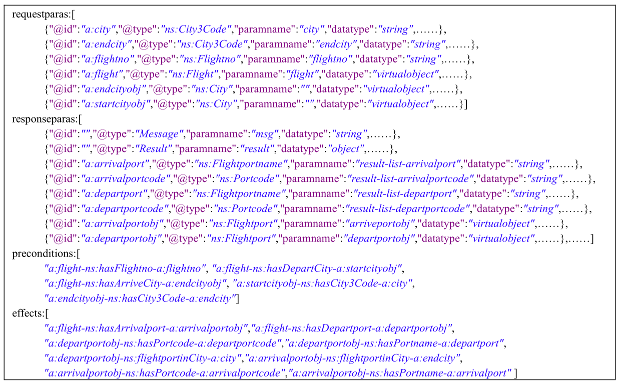
(4) Business parameters are crucial elements reflecting I/O features of a Web API, but, they always are mixed with format parameters in request and response. For instance, return data format, request data number, application key are format parameters in request; execution status code and execution message both are format parameters in the response.

(5) Some Web APIs don't have request, and only have response. For example, *Cities for weather Query* API can return all cities used for weather query APIs, and users don't need to input any request parameters.

Considering those problems above, in our semantic annotation, we design a semantic annotation specification from the perspective of developers, shown in the following. The specification make IOPE and other features of a Web API clearly and easily integrate with other Web APIs.

(1) General JSON-LD element *@id* and *@type* are used to respectively specify unique identify and semantics of a parameter. For format parameters (e.g. application key, result, execution message, status code), only their @type attributes are specified, because format parameters with the same semantics are unique in a Web API. Some business parameters, that may interoperate with other Web APIs, are picked. And both *@id* and *@type* attributes are specified for those picked parameters. An annotation fragment following this specification is shown in Fig. 11, where parameters without *@id* and *@type* aren't shown.

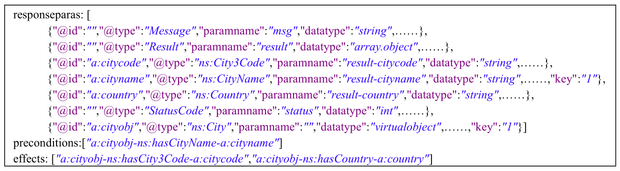
(2) For a Web API with multiple alternative request parameters, users can create multiple API-JSONLD documents. In each document, only one alternative parameter is picked, and other components are the same. For example, for *Weather Query* API, its request parameters may be *city*, *cityid*, *citycode*, or *location*. According to these request parameters, a user can create four similar API-JSONLD documents for this Web API.



```
requestparas:[
        {"@id":"a:city","@type":"ns:City3Code","paramname":"city","datatype":"string",……},
        {"@id":"a:endcity","@type":"ns:City3Code","paramname":"endcity","datatype":"string",……},
        {"@id":"a:flightno","@type":"ns:Flightno","paramname":"flightno","datatype":"string",……},
        {"@id":"a:flight","@type":"ns:Flight","paramname":"flight","datatype":"virtualobject",……},
        {"@id":"a:endcityobj","@type":"ns:City","paramname":"","datatype":"virtualobject",……},
        {"@id":"a:startcityobj","@type":"ns:City","paramname":"","datatype":"virtualobject",……}]
responseparas:[
        {"@id":"","@type":"Message","paramname":"msg","datatype":"string",……},
        {"@id":"","@type":"Result","paramname":"result","datatype":"object",……},
        {"@id":"a:arrivalport","@type":"ns:Flightportname","paramname":"result-list-arrivalport","datatype":"string",……},
        {"@id":"a:arrivalportcode","@type":"ns:Portcode","paramname":"result-list-arrivalportcode","datatype":"string",……},
        {"@id":"a:departport","@type":"ns:Flightportname","paramname":"result-list-departport","datatype":"string",……},
        {"@id":"a:departportcode","@type":"ns:Portcode","paramname":"result-list-departportcode","datatype":"string",……},
        {"@id":"a:arrivalportobj","@type":"ns:Flightport","paramname":"arriveportobj","datatype":"virtualobject",……},
        {"@id":"a:departportobj","@type":"ns:Flightport","paramname":"departportobj","datatype":"virtualobject",……},……]
preconditions:[
        "a:flight-ns:hasFlightno-a:flightno", "a:flight-ns:hasDepartCity-a:startcityobj",
        "a:flight-ns:hasArriveCity-a:endcityobj", "a:startcityobj-ns:hasCity3Code-a:city",
        "a:endcityobj-ns:hasCity3Code-a:endcity"]
effects:[
        "a:flight-ns:hasArrivalport-a:arrivalportobj","a:flight-ns:hasDepartport-a:departportobj",
        "a:departportobj-ns:hasPortcode-a:departportcode","a:departportobj-ns:hasPortname-a:departport",
        "a:departportobj-ns:flightportinCity-a:city","a:arrivalportobj-ns:flightportinCity-a:endcity",
        "a:arrivalportobj-ns:hasPortcode-a:arrivalportcode","a:arrivalportobj-ns:hasPortname-a:arrivalport" ]
```

FIGURE 11. An annotation fragment for *Flight Query* API.

(3) To gather relevant parameters together, various virtual parameters representing business objects are introduced. Their data types are assigned to *virtualobject*. These parameters can exist in request or response. For example, in annotated *Flight Query* API (Fig 11), there are three virtual parameters in request (*requestparas*): *a:flight*, *a:endcityobj*, *a:startcityobj*, and two virtual parameters in response (*responseparas*): *a:arrivalportobj*, *a:departportobj*. Relationships between virtual and real parameters are presented in preconditions or effects. Specially, each literal in preconditions or effects is a string including three components (split by "-"). The middle component represents an object property from an ontology, and the first and last components respectively represent domain and range values of this property.

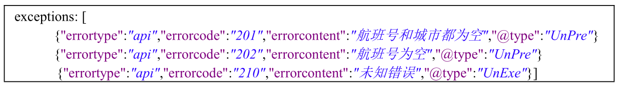


```
responseparas: [
        {"@id":"","@type":"Message","paramname":"msg","datatype":"string",……},
        {"@id":"","@type":"Result","paramname":"result","datatype":"array.object",……},
        {"@id":"a:citycode","@type":"ns:City3Code","paramname":"result-citycode","datatype":"string",……},
        {"@id":"a:cityname","@type":"ns:CityName","paramname":"result-cityname","datatype":"string",……,"key":"1"},
        {"@id":"a:country","@type":"ns:Country","paramname":"result-country","datatype":"string",……},
        {"@id":"","@type":"StatusCode","paramname":"status","datatype":"int",……},
        {"@id":"a:cityobj","@type":"ns:City","paramname":"","datatype":"virtualobject",……,"key":"1"}]
preconditions:["a:cityobj-ns:hasCityName-a:cityname"]
effects: ["a:cityobj-ns:hasCity3Code-a:citycode","a:cityobj-ns:hasCountry-a:country"]
```

FIGURE 12. An annotation fragment for *Flight Cities Query* API.

(4) Function type of a Web API without request will be assigned to *dctionary service* (first element of attribute function). In the API, some parameters are picked as query keys, and an example is shown in Fig. 12. Here, attribute *key* of each picked parameter is assigned to *1*, e.g: *a:cityname* and *a:cityobj*. The two parameters will be considered as input parameters of the API, and other parameters with @*id* will be as output parameters. Specially, a virtual parameter *a:cityobj* is introduced to gather all real parameters for city features together.

(5) Business exceptions (value of attribute *errortype* is *api*) in attribute *exceptions* also are annotated. This can help users to automatically handle business exceptions during invocation. An exception annotation fragment is shown in Fig. 13. Here, @*type* is used to specify exception semantics of each exception.



```
exceptions: [
        {"errortype":"api","errorcode":"201","errorcontent":"航班号和城市都为空","@type":"UnPre"}
        {"errortype":"api","errorcode":"202","errorcontent":"航班号为空","@type":"UnPre"}
        {"errortype":"api","errorcode":"210","errorcontent":"未知错误","@type":"UnExe"}]
```

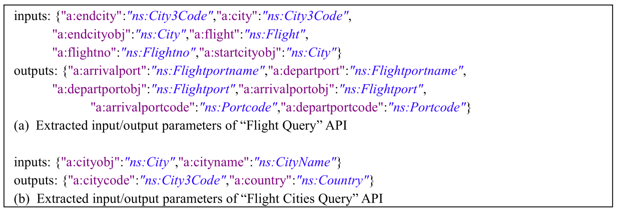
FIGURE 13. An exception annotation fragment of *Flight Cities Query* API.

D. FUNCTIONAL SEMANTICS EXTRACTION

According to previous annotation specifications, except attribute *inputs* and *outputs*, other attributes in an API-JSONLD document can be specified by current user. Thus, P/E components in IOPE features of a Web API can be directly obtained from attribute *preconditions* and *effects*. Meanwhile, I/O components can be automatically extracted from annotated request/response parameters. The generation process is shown in Fig. 14.

```
Algorithm 2 I/OExtraction
Input: apijsonld: a API-JSONLD document.
Output: updated apijsonld
01. extracting following data from apijsonld
        reqparams ← all parameters with attribute @id and @type in attribute requestparas
        resparams ← all parameters with attribute @id and @type in attribute responseparas
02. IF reqparams is null THEN // Dictionary Service
03.     reqparams ← all parameters with key in resparams
04.     resparams ← resparams - reqparams
05. END IF
06. FOR each parameter p in reqparams THEN
07.     pid ← value of attribute @id of p, pidsem ← value of attribute @type of p
08.     newp ← a new parameter is created by pid and sem
09.     add newp into attribute inputs of apijsonld
10. END FOR
11. FOR each parameter p in resparams THEN
12.     pid ← value of attribute @id of p, pidsem ← value of attribute @type of p
13.     newp ← a new parameter is created by pid and sem
14.     add newp into attribute outputs of apijsonld
15. END FOR
```

FIGURE 14. Algorithm for extracting I/O parameters.



```
inputs: {"a:endcity":"ns:City3Code","a:city":"ns:City3Code",
        "a:endcityobj":"ns:City","a:flight":"ns:Flight",
        "a:flightno":"ns:Flightno","a:startcityobj":"ns:City"}
outputs: {"a:arrivalport":"ns:Flightportname","a:departport":"ns:Flightportname",
        "a:departportobj":"ns:Flightport","a:arrivalportobj":"ns:Flightport",
                "a:arrivalportcode":"ns:Portcode","a:departportcode":"ns:Portcode"}
(a)  Extracted input/output parameters of "Flight Query" API

inputs: {"a:cityobj":"ns:City","a:cityname":"ns:CityName"}
outputs: {"a:citycode":"ns:City3Code","a:country":"ns:Country"}
(b)  Extracted input/output parameters of "Flight Cities Query" API
```

FIGURE 15. An example for extracted input and output parameters.

Firstly, the algorithm extracts request/response parameters from a given API-JSONLD document (line *01*). Only those parameters with attribute @*id* and @*type* are picked. Secondly, if current API is *dictionary service*, and then those response parameters with attribute *key* will be considered as request parameters (lines *02-05*). Thirdly, it respectively creates input and output parameters according to extracted request/response parameters (lines *06-15*). An example for extracted input and output parameters is shown in Fig. 15.

It is noticed that it is easily to convert IOPE data of an annotated API-JSONLD document into specific formats required by existing discovery and composition tools.

E. AUTOMATIC ONTOLOGY GENERATION

In practices, Web APIs are from various business domains. Generally, few of suitable existing ontologies can be used to provide semantics for all Web APIs. Therefore, when no vocabularies from existing ontologies are suitable, new vocabularies may be used during annotating a Web API. To raise reuse rate of ontologies, it is necessary to generate new ontologies or modify existing ontologies. Here, an automatic ontology generation algorithm is designed, and is shown in Fig. 16. It will extract semantic vocabularies from an API-JSONLD document, and then create new ontologies or add new vocabularies into existing ontologies.

```
Algorithm 3 OntologyGeneration
Input: apijsonld: a API-JSONLD document
Output: ontologies that are created and updated
01. extracting following data from apijsonld
      nsprefixmap ← a mapping from name space prefixes to their name spaces, that are parsed from @context.
      params ← all parameters with @id and @type in requestparas and responseparas.
      literals ← all literals in preconditions and effects.
02. nsontomap ← a mapping from name spaces in nsprefixmap and their ontologies, and ontologies can be obtained from specific ontology DB.
03. FOR each parameter p in params THEN
04.     semconcept ← value of @type in p
05.     nspre ← name space prefix of semconcept
06.     onto ← nsontomap.get(nsprefixmap.get(nspre))//obtain ontology for nspre
07.     IF onto is null THEN
08.        onto ← a new ontology model
09.            createNewClass(semconcept, p.datatype, p.pararemark, onto)//create a new class for semconcept where p.datatype is its data type, and p.pararemark is its comment, and add it into onto.
10.        nsontomap.put(nsprefixmap.get(nspre),onto)
11.     ELSE
12.        IF semconcept exists in onto THEN CONTINUE
13.        createNewClass(semconcept, p.datatype, p.pararemark, onto)
14.     END IF
15. END FOR
16. FOR each literal l in literals THEN
17.     (p1, semproperty, p2) ← three elements in l (split by "-")
18.     domain ← getSemConcept(p1)//obtain semantic concept of parameter p1
19.     range ← getSemConcept(p2)
20.     nspre ← name space prefix of semproperty
21.     onto ← nsontomap.get(nsprefixmap.get(nspre))//obtain ontology for nspre
22.     IF onto is null THEN
23.        onto ← a new ontology model
24.         createNewObjProp(semproperty, domain, range, onto)//create a new object property where semproperty is its name, domain and range respectively are its domain and range; and then add it into onto.
25.        nsontomap.put(nsprefixmap.get(nspre),onto)
26.     ELSE
27.        IF semconcept exists in onto THEN CONTINUE
28.        createNewObjProp(semproperty ,domain, range, onto)
29.     END IF
30. END FOR
```

FIGURE 16. Algorithm for automatically generating ontology during annotation.

The algorithm firstly parses ontologies, annotated parameters in request/response, and literals in preconditions/effects out of an API-JSONLD document (line *01*). Then, it loads all introduced ontologies in this document (line *02*). Thirdly, it parses semantic concepts out of parameters, and creates new ontologies or adds new classes into existing ontologies according to whether ontologies and semantic concepts for parameters are new or not (lines *03-15*). Lastly, it parses object properties out of literals, and carries out the similar process with the semantic concepts (lines *16-30*).

The algorithm will be invoked when an API-JSONLD document is saved. Thus, during annotation of following Web APIs, these newly generated ontologies can be reused.

F. SEMI-AUTOMATIC BRIDGE RULE GENERATION

During annotation, a user can pick vocabularies from existing ontologies, and also can create new vocabularies. Thus, three main semantic conflicts among two vocabularies from different ontologies may occur. The first is that the two vocabularies from different ontologies have the same name and meaning. However, they can't be considered as the same vocabulary, because their name spaces are different. The second is that the two vocabularies have the same name, but different meanings. This means any two vocabularies with the same name shouldn't be directly considered as the same meaning. The third is that the two vocabularies have different names, but the similar meaning. Judgement of the second and third conflicts needs users' assistance, that is, users need manually to specify which two vocabularies the conflict occurs between. Also, for the third conflict, further relationship between the two vocabularies should be specified, such as sub-class, super-class, or equality. In the following, the three semantic conflicts respectively are called ***semconflict1***, ***semconflict2***, and ***semconflict3***.

To solve the conflicts above, we use bridge rule technology in our previous work [22] to declare real relationship between two vocabularies. A bridge rule can define relationship between two vocabularies, and different rule types mean different relationship meanings. For instance, rule ⟨*ns1:Train, intoc, ns2:Vehicle*⟩ means concept *Train* in ontology *ns1* is a sub-class of *Vehicle* in ontology *ns2*. Other rule types have been introduced in section 2. Here, an automatic bridge rule generation algorithm is designed to improve efficiency of bridge rule creation, and is shown in Fig. 2. Based on given vocabulary information with ***semconflict2*** and ***semconflict3*** (parameter *vacabwith2* and *manualbrs*), the algorithm can automatically infer various new bridge rules implied by given ontologies (parameter *ontos*) with the help of ***Jena*** ontology tool kit.

The algorithm firstly loads all ontologies into an ontology model, and extracts all concepts and properties from the model (lines *01-03*). Then, it automatically generates basic bridge rules according to relationship declarations in existing ontologies and situation in ***semconflict1*** (lines *04-19*). Lastly, based on the basic bridge rules and manual rules given by users, all implied rules are obtained (line *20*). Specially, in this algorithm, three new operations are invoked: *isEqualc* (line *06*), *isEqualr* (line *14*), *generateImpBR* (line *20*).

Operation *isEqualc/isEqualr* is used to determine whether two concepts/properties are equal or not. In *isEqualc*, two

```
Algorithm 4 BridgeRuleGeneration
Input: vacabwith2: vocabularies that have different meanings from vocabularies with the same
name; manualbrs: bridge rules between two vocabularies with different names or in vacabwith2;
ontos: ontologies used by all API-JSONLD documents
Output: brs: all bridge rules among all ontologies in ontos
01. ontomodel ← an ontology model where all ontologies in ontos are loaded.
02. concepts ← all classes in ontomodel.
03. props ← all object properties in ontomodel.
04. FOR any c1, c2 in concepts THEN //generate bridge rules to solve semconflict1 considering
    semconflict2
05.     IF c1==c2 or c1 in vacabwith2 or c2 in vacabwith2 THEN CONTINUE
06.     IF isEqualc(c1,c2) THEN add rule <c1, equalc, c2> into brs
07. END FOR
08. FOR c in concepts THEN
09.     subs ← direct and indirect sub-classes of c, that are obtained through reasoning on
        ontomodel (Jena is employed)
10.     FOR s in subs THEN add rule <c, ontoc, s>, <s, intoc, c> into brs
11. END FOR
12. FOR any p1, p2 in props THEN //generate bridge rules to solve semconflict1 considering
    semconflict2
13.     IF p1==p2 or p1 in vacabwith2 or p2 in vacabwith2 THEN CONTINUE
14.     IF isEqualr(p1,p2) THEN add rule <p1, equalr, p2> into brs
15. END FOR
16. FOR p in props THEN
17.     subs ← direct and indirect sub-properties of c, that are obtained through reasoning on
        ontomodel (Jena is employed)
18.     FOR s in subs THEN add rule <p, ontor, s>, <s, intor, p> into brs
19. END FOR
20. brs ← generateImpBR(brs, manualbrs)// generate all bridge rules that are implied by brs and
    manualbrs
21. RETURN brs
```

FIGURE 17. Algorithm for generating bridge rules among multiple ontologies.

concepts are equal when one of two conditions in the following is satisfied:

- They have the same local name and data type. Here, local name means a name without name space.
- The equivalent relationship between them is declared explicitly in ontologies.

In *isEqualr*, two properties are equal when three conditions in the following all are satisfied:

- They have the same local name, or the equivalent relationship between them is declared explicitly in ontologies.
- Their domains *d1*, *d2* make *isEqualc(d1, d2)* true or rule ⟨*d1, equalc, d2*⟩ exists.
- Their ranges *r1*, *r2* make *isEqualc(r1, r2)* true or rule ⟨*r1, equalc, r2*⟩ exists.

Operation *generateImpBR* is used to infer implied rules according to the previous basic bridge rules and manual bridge rules. The following steps show its inference process:

(1) Generate new bridge rules according to symmetry. If ⟨*c1, rt, c2*⟩ exists, then ⟨*c2, inverseop(rt), c1*⟩ is created. Here, *rt* means one of six rule types *(intoc, ontoc, equalc, intor, ontor, equalr)*, and operation *inverseop* is used to convert *rt* into its inverse type. Specially the inverse type of *intoc/r* is *ontoc/r*, and the inverse type of *equalc/r* is itself.

(2) Generate new bridge rules according to transitivity. If ⟨*c1, rt, c2*⟩ and ⟨*c1', rt, c1*⟩ exist, then ⟨*c1', rt, c2*⟩ and ⟨*c2, inverseop(rt), c1'*⟩ are created. And, if ⟨*c1, rt, c2*⟩ and ⟨*c2, rt, c2'*⟩ exist, then ⟨*c1, rt, c2'*⟩ and ⟨*c2',inverseop(rt),c1*⟩ exist.

(3)Infer new bridge rules among rules with different rule types. Here, the following four inference situations are considered, where *rt* only represents *intoc/r* or *ontoc/r*:

- If ⟨*c1, rt, c2*⟩ and ⟨*c2, equalc/r, c2'*⟩ exist, then ⟨*c1, rt, c2'*⟩ and ⟨*c2', inverseop(rt), c1*⟩ are created.
- If ⟨*c1, rt, c2*⟩ and ⟨*c1,equalc/r,c1'*⟩ exist, then ⟨*c1', rt, c2*⟩ and ⟨*c2, inverseop(rt), c1'*⟩ are created.
- If ⟨*c1, equalc/r, c2*⟩ and ⟨*c1, rt, c1'*⟩ exist, then ⟨*c2, rt, c1'*⟩ and ⟨*c1', inverseop(rt), c2*⟩ are created.
- If ⟨*c1, equalc/r, c2*⟩ and ⟨*c2, rt, c2'*⟩ exist, then ⟨*c1, rt, c2'*⟩ and ⟨*c2', inverseop(rt),c1*⟩ are created.

It is noticed that generated bridge rules by this algorithm declare various relationships between different two vocabularies in given ontologies. These rules can effectively eliminate semantic conflicts among distributed knowledge.

VI. EXPERIMENT EVALUATION

In this section, a series of experiments are designed to evaluate effectiveness and efficiency of our semantic annotation approach.

A. TEST CASE ILLUSTRATION

We crawl *183* Web API descriptions from two open API platforms (Jisu & Juhe), and *23* business domains are involved. Main business domains and API numbers in them is shown in Table 7. Four function types of APIs are involved: *information providing* (*134* APIs), *world altering* (*1* APIs), *dictionary service* (*45* APIs), and *value providing* (*3* APIs). It can be seen that most APIs are *information providing*. Furthermore, we store all interface description information into a database including provider name, API type, API name, access URL, return format, request method, request example, response example, etc. Also, we assign an unique ID to every API.

TABLE 7. Main business domain details.

No.	Domain	API num	No.	Domain	API num
1	calendar	2	13	express	2
2	QR code	3	14	literature	2
3	traffic	15	15	train	3
4	face recognition	1	16	movie	5
5	hospital	7	17	TV	2
6	unit conversion	1	18	network	1
7	divine	2	19	translation	1
8	history	1	20	administrative area	5
9	card recognition	1	21	poetry	9
10	book	88	22	vehicle	7
11	weather	13	23	flight	5
12	business	7			

B. EXPERIMENT ENVIRONMENT

Based on JWASA, a semantic annotation tool is implemented by means of JavaEE platform, and it adopts B/S architecture to support simultaneous annotation by multiple developers. Application server *Tomcat8.0* is used to deploy the tool. *Jena3.8* is imported to implement reasoning and basic management operations about ontologies. *Mysql5.1* is picked as the tool's database to store original API description information, generated bridge rules, and other information about annotation. And the tool is installed on *ThinkPad X1 (1.80GHz,1.99GHz, 16GRAM, Win10)*.

Based on JWASA, we design an experiment to complete semantic annotation of all Web APIs in the previous test case. It includes the following five steps:

- **Step 1**: Generate an API-JSON document for every Web API according to its description crawled from Internet.
- **Step 2**: Generate an API-JSONLD template document for every API-JSON document.
- **Step 3**: Manually add semantic vocabularies in every template document, and save semantically annotated API-JSONLD document.
- **Step 4**: Correct and improve ontologies generated in Step 3.
- **Step 5**: Generate bridge rules among all used business domain ontologies in annotation.

During annotation, three types of files are generated: API-JSON, API-JSONLD, ontology. The first two types of documents all are saved in JSON format in files with suffix *.json* and *.jsonld*. And the last are saved in OWL [25] format in files with *.owl* suffix.

Furthermore, in following experiments, execution time of an algorithm is the average of execution time of *5* runs under the same environment.

TABLE 8. Details of parsed response parameters in part of business domains.

Domain	API num	Minimum	Maximum
calendar	2	21	29
QR code	3	3	4
traffic	15	3	25
hospital	7	4	36
book	88	5	22
weather	13	6	92
business	7	5	36
movie	5	10	31
administrative area	5	7	10
poetry	9	6	14
vehicle	7	5	23
flight	5	9	33

C. EFFECTIVENESS AND EFFICIENCY

1) PARSING RESPONSE PARAMETERS AND GENERATING UNITED API-JSON DOCUMENT

In Step 1 of the experiment, Algorithm 1 (*ParseRespara*) is invoked to parse response parameters out of relevant response example during generating every API-JSON document. Finally, *183* API-JSON files are created. Table 8 shows details of parsed response parameters in part of business domains, including business domain (*Domain*), API number (*APInum*), minimum of parsed parameters (*Minimum*), and maximum of parsed parameters (*Maximum*). It can be seen that response parameters are effectively parsed out of response example with JSON format.

In Step 1, we also evaluate execution time of generating API-JSON documents by an extra experiment. Based on crawled API descriptions for all Web APIs in the test case, we automatically create *183* API-JSON files at one time. The creation process includes reading API descriptions from Database, parsing response parameters from response example, and saving to API-JSON files. The result shows that it spent *5350 ms* creating relevant *183* API-JSON files. The most time is only 131ms for single Web API. This means generation of an API-JSON document is very fast.

2) JSON-LD BASED SEMANTIC ANNOTATION AND FUNCTIONAL SEMANTICS EXTRACTION

It is noticed that many Web APIs have dozens of response parameters. This makes annotation work tedious. To facilitate the annotation work, JWASA provides user-friendly operation interfaces to directly edit data in JSON format and to retrieve vocabularies in existing ontologies.

Meanwhile, in Step 2 of the experiment, through adding necessary attributes into API-JSON documents, *183* API-JSONLD templates are created. Based on these templates, we manually annotate *183* APIs in Step 3 of the experiment, and finally generate *183* API-JSONLD documents. Table 9 shows details of parameters in five semantically annotated Web APIs from *flight* domain, including API name, the number of old request parameters (*Reqpara*), the number of annotated request parameters(*Areqpara*), the number of added virtual parameters in request (*Reqvobj*), the number of old response parameters (*Respara*), the number of semantically annotated response parameters (*Arespara*), the number of added virtual parameters in response (*Resvobj*), and function type (*Functiontype*). It is noticed that fewer parameters are semantically annotated in request/response than old parameters. Specially, those annotated parameters are close to relevant business, and are distinguished from other parameters with the annotator's experience. Also, except API "*Flight No. Query*", for other four APIs, some virtual parameters are introduced in their request or response parameters.

In addition, for *183* API-JSONLD documents, we carry out Algorithm 2 to set attribute *inputs* and *outputs* in each document. Finally, the two attributes are set correctly according to annotated request/response parameters and function type of current document. Algorithm 2 is invoked when an annotated API-JSONLD document is saved, and can be completed in several milliseconds.

3) AUTOMATIC ONTOLOGY GENERATION

In Step 3 of the experiment, Algorithm 3 (*OntologyGeneration*) is invoked to generate ontologies used in an API-JSONLD document, when this document is saved. During annotation process of the *183* APIs, *14* business ontologies are automatically generated and saved as OWL files. Details of these ontologies are shown in Table 10, including domain, concept number (*CNum*), object property number (*PNum*),

TABLE 9. Details of five annotated APIs in *flight* domain.

API name	Reqpara	Areqpara	Reqvobj	Respara	Arespara	Respvobj	Functiontype
Realtime Flight Query	6	3	3	28	4	2	information providing
History Flight Query	2	2	0	25	5	2	information providing
Flight No. Query	2	1	0	23	7	0	information providing
Flight Cities	0	0	0	9	3	1	dictionary service
Flight Query	5	3	2	33	4	2	dictionary service

TABLE 10. Details of generated ontologies.

No.	Domain	CNum	PNum	APInum
1	common	69	59	60
2	vehicle	82	80	21
3	book	83	228	87
4	weather	49	46	13
5	company	24	21	3
6	poetry	14	23	9
7	administrative division	8	8	5
8	hospital	13	11	7
9	business	15	13	4
10	literature	7	6	2
11	train	14	19	4
12	movie	11	13	5
13	TV	3	2	2
14	flight	10	12	5

and number of APIs using current ontology (*APInum*). It is noticed that an ontology can be used by multiple APIs, such as *60* APIs for ontology *common*, *87* APIs for ontology *book*.

We also design an extra experiment to evaluate efficiency of ontology generation. In the experiment, we assume that semantic vocabularies in all annotated API-JSONLD documents are new vocabularies, and then, based on semantic vocabularies from *183* annotated API-JSONLD documents, we carry out Algorithm 3 to create all business ontologies at one time. The creating process includes reading API-JSONLD files, creating ontologies, and saving to OWL files. The result shows that it spent *3172 ms* creating all *14* business ontologies. Thus, for every API-JSONLD document, the time spending in generating ontologies only is about tens of milliseconds.

In Step 4 of the experiment, for those automatically generated ontologies, we manually correct and improve fewer part of concepts' comments, and add comments of all object properties. Meanwhile, we add some abstract classes/properties into ontology *common*, and they can be set super class/property of those in concrete business ontologies. This can make more relevant APIs be found. For example, *Location* is a new class in ontology *common*, and its subclasses include *Address*, *GPSLocation*, *Station*, *Flightport* in other ontologies; property *locationInCity* is added into the ontology, and it is the super-properties of some properties in other ontologies. Furthermore, previous three semantic conflicts also exist among these ontologies. It is noticed that there are 26 groups of concepts with the same name. Specially, concept *City* appears in *6* different ontologies. *City* in ontology *weather* and *flight* means a city object, but in ontology *common*, *vehicle*, *administrative division* and *hospital* only means city name.

4) AUTOMATIC BRIDGE RULE GENERATION

In Step 5 of the experiment, we carry out Algorithm 3 (*BridgeRuleGeneration*) to facilitate composition among APIs, which can infer implied relationships and eliminate various semantic conflicts among concepts/properties among ontologies. The concrete results under two situations is shown in Table 11, including number of generated bridge rules (*BRnum*), number of bridge rules for various rule types (*equalc*, *intoc*, etc.), number of vocabularies that have different meanings from vocabularies with the same name (*Diffvocab*), and number of manually added bridge rules (*MBRnum*).

TABLE 11. Details of generated bridge rules under two situations.

BRnum	equalc	intoc	ontoc	equalr	intor	ontor	Diffvocab	MBRnum
272	176	5	5	86	0	0	0	0
274	166	8	8	86	3	3	3	9

In the first situation (row 1 in Table 11), there are no vocabularies in *diffvocab* and no bridge rules in *mBrnum*, and the total number of generated bridge rules is *272*. Here, *equalc* and *equalr* rules account for most of them, because there are many vocabularies with the same name. Specially, before running the algorithm, we manually set subclass relationship between *Location* and *Address* in ontology *common*. Thus, five *intoc* and five *ontoc* bridge rules are inferred. In the second situation (row 2 in Table 11), there are *3* vocabularies in *diffvocab* and *9* bridge rules are in *mBRnum*, and *274* bridge rules are generated. Except for *equalc* and *equalr* rules, other four types of rules also are generated. It is noticed that, compared with the first situation, *equalc* rules are less, and *intoc/r*, *ontoc/r* rules become more. This means the vocabularies in *diffcocab* and manual bridge rules are considered during execution of the algorithm.

In the second situation, we found it spent about *2521ms* to generate *274* bridge rules among *14* ontologies. The execution time in seconds can be accepted in practice.

Furthermore, generated bridge rules include various relationships between two vocabularies in two different ontologies or in the same ontology. Therefore, these rules can play an important role in future API automatic discovery and composition.

VII. CONCLUSION

In this article, a semi-automatic Web API semantic annotation approach is provided, namely JWASA. It adopts lightweight JSON-LD format to implement semantic annotation of a Web API, and meanwhile, proposes a whole solution for Web API semantic annotation. In JWASA, two types of documents in JSON format are designed: API-JSON and API-JSONLD, and respectively describe Web APIs and semantic Web APIs. Meanwhile, a common Web API description ontology *WebAPIOnto* is designed to provide semantic vocabularies for common features of Web APIs from different providers. Furthermore, a semantic annotation specification and a series of algorithms are designed to improve effectiveness and efficiency of annotation. These algorithms includes parsing response parameters, extracting I/O parameters, automatic domain ontology generation, and semi-automatic bridge rule inference. Finally, for crawled Web APIs from Internet, JWASA can generate three types of artifacts: API-JSONLD documents, domain ontologies, and bridge rules. All of them are crucial elements for following semantic-based Web API manipulation, e.g: discovery [12] and composition [26]. Based on a prototype system of JWASA and real Web APIs on Internet, a series of experiments are carried out. Experiment results show JWASA is effective and efficient.

JWASA mainly is used to semantically annotate features related to function of Web APIs. Differences among description format from different providers are efficiently handled. This can facilitate future API automatic composition. However, picking semantic vocabulary is completed manually. This requires annotators are professional for relevant business domains. Therefore, in futher, we will improve degree of automation of our JWASA through semantic recommendation technologies [27]. Also, on artifacts of JWASA, we will research automatic invocation, composition and ecosystem evolution [28] of Web APIs.

ACKNOWLEDGMENT

Xianghui Wang thanks her colleagues from Shandong Jianzhu University for their comments.

REFERENCES

[1] I. Nadareishvili, R. Mitra, M. Mclarty, and M. Amundsen, *Microservice Architecture: Aligning Princ., Practices, Culture*. Newton, MA, USA: O'Reilly Media, 2016.

[2] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Semantic annotation of Web apis with sweet," in *Proc. 6th Workshop Scripting Develop. Semantic Web, Colocated*, Heraklion, Crete, Greece, May 2010, pp. 1–13.

[3] C. Luo, Z. Zheng, X. Wu, F. Yang, and Y. Zhao, "Automated structural semantic annotation for restful services," *Int. J. Web Grid Services*, vol. 12, no. 1, pp. 26–41, 2016.

[4] M. Garriga, C. Mateos, A. Flores, A. Cechich, and A. Zunino, "RESTful service composition at a glance: A survey," *J. Netw. Comput. Appl.*, vol. 60, pp. 32–53, Jan. 2016.

[5] A. Ranabahu, A. Sheth, M. Panahiazar, and S. Wijeratne, *Semantic Annotation and Search for Resources in the Next Generation Web With Sa-Rest*. Pune, Italy: Knoesis, 2011.

[6] C. Bizer, K. Eckert, R. Meusel, H. Mãhleisen, M. Schuhmacher, J. Vãlker, H. Alani, L. Kagal, A. Fokue, and P. Groth, "Deployment of rdfa, microdata, and microformats on the Web-a quantitative analysis," in *Proc. Int. Semantic Web Conf.*, 2013, pp. 17–32.

[7] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, and R. V. de Walle, *Survey of Semantic Description of REST APIs*. New York, NY, USA: Springer, 2014.

[8] M. N. Lucky, M. Cremaschi, B. Lodigiani, A. Menolascina, and F. D. Paoli, "Enriching api descriptions by adding api profiles through semantic annotation," in *Proc. Int. Conf. Service-Oriented Comput.*, 2016, pp. 780–794.

[9] C. Peng and G. Bai, "Using tag based semantic annotation to empower client and REST service interaction," in *Proc. 3rd Int. Conf. Complex., Future Inf. Syst. Risk*, 2018, pp. 64–71.

[10] C. Marco and D. P. Flavio, "Toward automatic semantic api descriptions to support services composition," in *Proc. Service-Oriented Cloud Comput.*, vol. 10465, 2017, pp. 159–167.

[11] J. Li, "A fast semantic Web services matchmaker for owl-s services," *J. Netw.*, vol. 8, no. 5, p. 1104, 2013.

[12] M. D. Wilkinson, B. Vandervalk, and L. McCarthy, "The semantic automated discovery and integration (SADI) Web service design-pattern, API and reference implementation," *J. Biomed. Semantics*, vol. 2, no. 1, p. 8, 2011.

[13] G. Kellogg, "JSON-LD: JSON for linked data," in *Proc. Semantic Technol. Bus. Conf.*, 2012, pp. 1–4.

[14] S. M. Sohan, C. Anslow, and F. Maurer, "SpyREST in action: An automated RESTful API documentation tool," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 813–818.

[15] D. Booth and K. L. Canyang, "Web services description language (wsdl) version 2.0 part 0: Primer," *W3C Recommendation*, vol. 26, pp. 39–41, Dec. 2007.

[16] W. A. Bernstein, "Sawsdl-imatcher: A customizable and effective semantic Web service matchmaker," *J. Web Semantics*, vol. 9, no. 4, pp. 402–417, 2011.

[17] M. D. L. Calache and C. R. G. D. Farias, "Graphical and collaborative annotation support for semantic Web services," in *Proc. IEEE Int. Conf. Softw. Archit. Companion (ICSA-C)*, Mar. 2020, pp. 210–217.

[18] U. Lampe, S. Schulte, M. Siebenhaar, D. Schuller, and R. Steinmetz, "Adaptive matchmaking for RESTful services based on hRESTS and MicroWSMO," in *Proc. 5th Int. Workshop Enhanced Web Service Technol.*, 2010, pp. 10–17.

[19] D. Roman, J. Kopecky, T. Vitvar, J. Domingue, and D. Fensel, "Wsmo-lite and hrests: Lightweight semantic annotations for Web services and restful apis," in *Proc. Web Semantics Sci. Services Agents World Wide Web*, vol. 31, 2015, pp. 39–58.

[20] R. Alarcon, R. Saffie, N. Bravo, and J. Cabello, *REST Web Service Description for Graph-Based Service Discovery*. Cham, Switzerland: Springer, 2015.

[21] C. Marco and D. P. Flavio, "A practical approach to services composition through light semantic descriptions," in *Proc. Service-Oriented Cloud Comput.*, vol. 11116, 2018, pp. 130–145.

[22] X. Wang, Z. Feng, and K. Huang, "D3L-based service runtime self-adaptation using replanning," *IEEE Access*, vol. 6, pp. 14974–14995, 2018.

[23] X. Wang, Z. Feng, K. Huang, and W. Tan, "An automatic self-adaptation framework for service-based process based on exception handling," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 5, p. e3984, Mar. 2017.

[24] A. Cheron, J. Bourcier, O. Barais, and A. Michel, "Comparison matrices of semantic restful apis technologies," in *Proc. Int. Conf. Web Eng.*, 2019, pp. 425–440.

[25] *Owl 2 Web Ontology Language Document Overview*, W. W. W. Consortium, Cambridge, MA, USA, 2012.

[26] X. Wang and Z. Feng, "Semantic Web service composition considering iope matching," *J. Tianjin Univ.*, vol. 50, no. 9, pp. 984–996, 2017.

[27] H. Zhang, D. Ge, and S. Zhang, "Hybrid recommendation system based on semantic interest community and trusted neighbors," *Multimedia Tools Appl.*, vol. 77, no. 4, pp. 4187–4202, Feb. 2018.

[28] X. Wang, Z. Feng, S. Chen, and K. Huang, "DKEM: A distributed knowledge based evolution model for service ecosystem," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2018, pp. 1–8.

XIANGHUI WANG (Member, IEEE) received the B.S. and M.S. degrees from the School of Computer Science and Technology, Shandong University, China, in 2002 and 2005, respectively, and the Ph.D. degree from the School of Computer Science and Technology, Tianjin University, China, in 2018. She is currently an Assistant Professor with the School of Computer Science and Technology, Shandong Jianzhu University, China. Her research interests include knowledge engineering and service computing.

QIAN SUN received the B.S. and M.S. degrees from the School of Computer Science and Technology, Tongji University, China, in 2004 and 2007, respectively. She is currently an Instructor with the School of Computer Science and Technology, Shandong Jianzhu University, China. Her research interests include workflow technology and service computing.

JINLONG LIANG received the B.S. degree from the School of Computer Science and Technology, Shandong University, China, in 2006, and the M.S. degree from the Qilu Software College, Shandong University, in 2017. He is currently a Senior Engineer with the Information Center, Shandong Provincial Qianfoshan Hospital, First Affiliated Hospital of Shandong First Medical University, China. His research interest includes enterprise information integration.