Rapidly Integrating Services into the Linked Data Cloud

Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and José Luis Ambite

University of Southern California
Information Sciences Institute and Department of Computer Science
{mohsen,knoblock,pszekely,ambite}@isi.edu

Abstract. The amount of data available in the Linked Data cloud continues to grow. Yet, few services consume and produce linked data. There is recent work that allows a user to define a linked service from an online service, which includes the specifications for consuming and producing linked data, but building such models is time consuming and requires specialized knowledge of RDF and SPARQL. This paper presents a new approach that allows domain experts to rapidly create semantic models of services by demonstration in an interactive web-based interface. First, the user provides examples of the service request URLs. Then, the system automatically proposes a service model the user can refine interactively. Finally, the system saves a service specification using a new expressive vocabulary that includes lowering and lifting rules. This approach empowers end users to rapidly model existing services and immediately use them to consume and produce linked data.

Keywords: linked data, linked API, service modeling

1 Introduction

Today's Linked Open Data (LOD) cloud consists primarily of databases that have been translated into RDF and linked to other datasets (e.g., DBpedia, Freebase, Linked GeoData, PubMed). Often, information is not current (e.g., Steve Jobs was listed as CEO of Apple Computer in DBpedia for months after his passing), and timely information is not available at all (e.g., the LOD cloud has no information about the current weather or events for any city).

Web APIs provide the opportunity to remedy this problem as there are thousands of Web APIs that provide access to a wealth of up-to-date data. For example, the `programmableweb`[1] lists over 6,000 APIs that provide data on an immense variety of topics. The problem is that most of these APIs provide data in JSON and XML and are not in any way connected to the LOD cloud. For example, in the `programmableweb` only 65 APIs (about 1%) provide information in RDF, and the rest provide information in XML and JSON. Our goal is to make it easy to connect the remaining thousands of XML and JSON-based

[1] `http://www.programmableweb.com/apis`

Web APIs to the LOD cloud. To do so we need to make it easy to represent the semantics of Web APIs in terms of well known vocabularies, and we need to wrap these APIs so that they can consume RDF from the LOD cloud and produce RDF that links back to the LOD cloud.

Several approaches have been developed to integrate Web APIs with the Linked Data cloud. One approach is to annotate the service attributes using concepts of known ontologies and publish the service descriptions into the cloud [10, 11]. This allows users to easily discover relevant services. A second approach is to wrap the APIs to enable them to communicate at the semantic level so that they can consume and produce linked data [9, 5]. A third approach is to create a uniquely identifiable resource for each instance of an API invocation and then link that resource to other data sets [13, 12]. Using this approach it is possible to invoke the API by dereferencing the corresponding resource URI.

The main obstacle preventing these approaches from gaining wide acceptance is that building the required models is difficult and time-consuming. In these approaches, a developer needs to create a model that defines the mapping from the information consumed and produced by Web APIs to Semantic Web vocabularies. In addition, the developer must also write the lowering and lifting specifications that lower data from RDF into the format expected by the Web APIs and then lift the results of the Web API invocations to RDF. Writing these models and the required lowering and lifting specifications often requires in-depth knowledge of RDF, SPARQL, and languages such as XPath or XSLT[2].

The key contribution of our work is a method to semi-automatically build semantic models of Web APIs, including the lowering and lifting specifications. In our approach, users provide sample URLs to invoke a service and the vocabularies they want to use to model the service. The system automatically invokes the service and builds a model to capture the semantics of the inputs, outputs and relationships between inputs and outputs. The system also provides an easy-to-use web-based graphical interface through which users can visualize and adjust the automatically constructed model. The resulting models, represented in RDF using standard vocabularies, enable service discovery using SPARQL queries. In addition, our system can automatically generate the lowering and lifting specifications from these models so that the Web APIs become immediately executable and able to consume and produce linked data.

2 Overview

The objective of our approach is to create linked APIs by combining traditional Web APIs and the Linked Data cloud in two aspects. We want to publish semantic service descriptions into the cloud that can be used by the linked data community in service discovery and composition. We also want to deploy APIs that interact at the semantic level, directly consuming Linked Data and generating RDF data that is linked to the input data. Since most Web APIs use the

[2] Extensible Stylesheet Language Transformations.

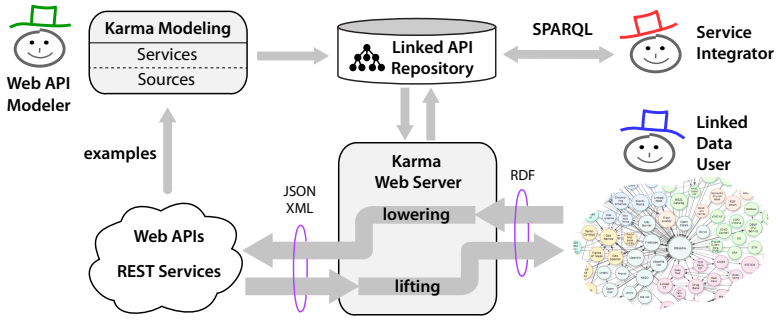


Fig. 1. The overview of our approach to create linked APIs.

HTTP GET method, in this paper, we focus on these APIs and assume that all inputs for service invocation are embedded in the invocation URL. The approach presented in this paper builds upon and extends Karma [3, 14, 15], our modeling and integration framework.[3]

Figure 1 shows the three main steps of our approach. The first step [14], the foundation of the rest of the process, is to semi-automatically build a service model that represents the semantics of the API functionality (Section 3). Users first provide examples of service request URLs. Karma then invokes the services and constructs a worksheet that contains both the inputs and the outputs produced. In this step we leverage our prior work on modeling sources, using it to construct a model of the input/output worksheet. The process is fast because users only need to provide a few examples of service request URLs and then adjust the automatically generated model. The process is also easy because users interact with the system through a graphical user interface and are not required to know Semantic Web technologies such as RDF, SPARQL, and XSLT.

The second step is to formally represent the semantic models built in the previous step (Section 4). Once the user models the API, Karma automatically generates the service descriptions and stores them in a repository. The linked API repository provides a SPARQL interface that service integrators can use for service discovery. For example, a service integrator can issue a query to retrieve all the APIs that return neighborhood information given latitude and longitude. A service integrator can also employ reasoning algorithms to generate a plan to achieve a specific goal.

The final part of our method is to deploy linked APIs on a Web server where they can be directly invoked, consuming and producing RDF (Section 5). The Web server provides a REST interface that Linked Data users can use to retrieve RDF data. It uses the service descriptions in the repository to automatically lower the input RDF, to invoke the actual Web API, and to lift the output to return linked (RDF) data.

[3] This paper is a significantly extended version of a workshop paper [14].

3 Semi-Automatically Modeling Web APIs

Our approach to model Web APIs using Karma consists of two parts. In the first part, users provide Karma a collection of sample invocation URLs. Karma uses these URLs to invoke the APIs and construct a worksheet that contains the inputs and corresponding outputs for each service invocation. In the second part, we use our prior Karma work to construct a model of the resulting worksheet. In this section we describe our prior Karma work, describe the procedure to construct the inputs/outputs worksheet from the invocation URLs, and illustrate the whole process using an example.

3.1 Previous Work on Source Modeling

To provide a better understanding of our new work on service modeling, in this section we briefly review our previous work on source modeling [3]. The process of modeling sources in Karma is a semi-automatic process that is initiated when users load a data source. Karma supports importing data from various structured sources including relational databases, spreadsheets, JSON, and XML. Then, users specify the vocabularies to which they want to map the source, and Karma automatically constructs a model that users can adjust. The output is a formal model that specifies the mapping between the source and the target ontology. Specifically, Karma generates a GLAV source description [3, 7].

The modeling process consists of two steps. The first step is to characterize the type of data by assigning a *semantic type* to each column. In our approach, a semantic type can be either an OWL class or the range of a data property (which we represent by the pair consisting of a data property and its domain). We use a conditional random field (CRF) [6] model to learn the assignment of semantic types to columns of data [2]. Karma uses this model to automatically suggest semantic types for data columns. If the correct semantic type is not among the suggested types, users can browse the ontology through a user friendly interface to find the appropriate type. Karma automatically re-trains the CRF model after these manual assignments.

The second part of the modeling process is to identify the relationships between the inferred semantic types in the ontology. Given the domain ontology and the assigned semantic types, Karma creates a graph that defines the space of all possible mappings between the source and the ontology [3]. The nodes in this graph represent classes in the ontology, and the links represent properties. The mapping is not one to one, because there might be several instances of the same class present in the source.

Once Karma constructs the graph, it computes the source model as the minimal tree that connects all the semantic types. The minimal tree corresponds to the most concise model that relates all the columns in a source, and this is a good starting point for refining the model. We use a Steiner tree algorithm to compute the minimal tree. Given an edge-weighted graph and a subset of the vertices, called Steiner nodes, the goal is to find the minimum-weight tree in the graph that spans all the Steiner nodes. The Steiner tree problem is NP-complete,

but we use a heuristic algorithm [4] with an approximation ratio bounded by $2(1 - 1/l)$, where l is the number of leaves in the optimal Steiner tree.

It is possible that multiple minimal trees exist, or that the correct interpretation of the data is captured by a non-minimal tree. In these cases, Karma allows the user to interactively impose constraints on the algorithm to build the correct model. Karma provides an easy-to-use GUI in which the user can adjust the relationships between the source columns [3].

3.2 Service Invocation

To enable Karma to model services in the same way that it models structured sources, we need to generate a table of example inputs and outputs. To this end, Karma asks the user to provide samples of the Web API requests. Karma parses the URLs and extracts the individual input parameters along with their values. For each request example, Karma invokes the service and extracts the output attributes and their values from the XML or JSON response. At the end, Karma joins the inputs and the outputs and shows them in a table.

Once this table is constructed, we apply our prior Karma work on source modeling to construct a model of the table. As described above, our source modeling technique captures the relationships among all columns of a source. Consequently, when we apply it to the table constructed from the API invocation URLs, the resulting model will capture the relationships between the inputs and outputs of the API.

An alternative method to collect examples of the API inputs and outputs is to extract such information from the documentation pages of the APIs. According to a comprehensive study on Web APIs by Maleshkova et al. [8], 83.8% of the APIs indexed in `programmableweb` provide a sample request and 75.2% of them also provide a sample response. In future work we plan to mine these documentation pages to extract examples of inputs and outputs.

3.3 Example

We illustrate our service modeling approach with an example from the GeoNames APIs[4]. We model the `neighbourhood` API,[5] which takes the latitude and longitude of a geographic feature as input and returns information about the neighborhood of that feature. To keep the example concise, we only consider the region name, the nearby city, the country code, and the country name. An example of the API invocation URL and the API response are shown in Figure 2(a) and (b). Figure 2(c) shows the table of inputs and outputs that Karma constructs using the invocation URLs listed in the first column of the table.

In the next step, Karma treats the service table as a data source and maps it to the ontologies given by the user (in our example GeoNames[6] and WGS84[7]

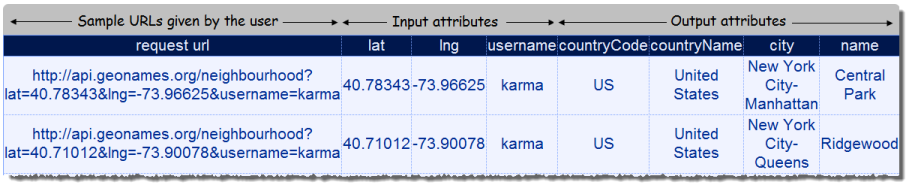
[4] `http://www.geonames.org/export/ws-overview.html`
[5] `http://www.geonames.org/export/web-services.html#neighbourhood`
[6] `http://www.geonames.org/ontology/ontology_v3.01.rdf`
[7] `www.w3.org/2003/01/geo/wgs84_pos`

```
http://api.geonames.org/neighbourhood?lat=40.78343&lng=-73.96625&username=karma
```

(a) Service Invocation URL.

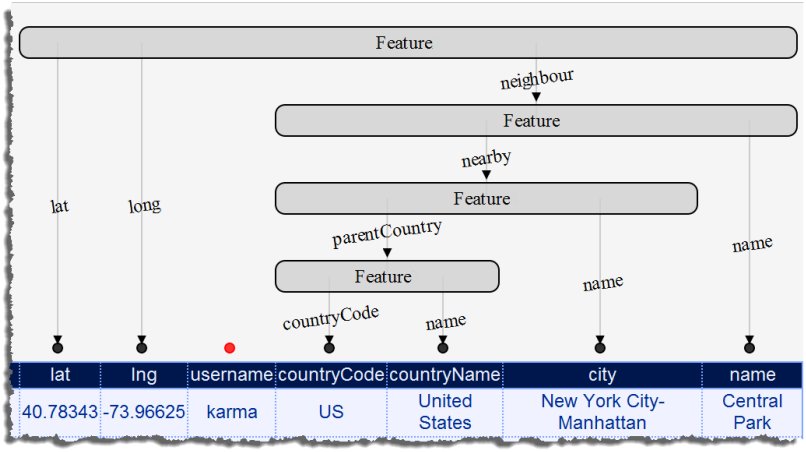
```
<geonames><neighbourhood>
      <countryCode>US</countryCode>
      <countryName>United States</countryName>
      <city>New York City-Manhattan</city>
      <name>Central Park</name>
       ...
</neighbourhood></geonames>
```

(b) Service response (XML).



Sample URLs given by the user	Input attributes		Output attributes				
request url	lat	lng	username	countryCode	countryName	city	name
http://api.geonames.org/neighbourhood?lat=40.78343&lng=-73.96625&username=karma	40.78343	-73.96625	karma	US	United States	New York City-Manhattan	Central Park
http://api.geonames.org/neighbourhood?lat=40.71012&lng=-73.90078&username=karma	40.71012	-73.90078	karma	US	United States	New York City-Queens	Ridgewood

(c) The user provides examples of the service requests. Karma extracts the input parameters, invokes the API, extracts the output attributes from the invocation response, and joins the outputs with the input data in one table.



(d) Screenshot showing the service model in Karma.

neighbourhood($lat, $long, @countryCode, @countryName, @city, @name) →
gn:Feature(v1) ∧ wgs84:lat(v1, $lat) ∧ wgs84:long(v1, $long) ∧ gn:*neighbourhood*(v1, v2) ∧
gn:Feature(v2) ∧ gn:name(v2, @name) ∧ gn:*nearby*(v2, v3) ∧
gn:Feature(v3) ∧ gn:name(v3, @city) ∧ gn:*parentCountry*(v3, v4) ∧
gn:Feature(v4) ∧ gn:countryCode(v4, @countryCode) ∧ gn:name(v4, @countryName)

(e) Logical LAV rule representing the semantics of the neighbourhood API. Input and output attributes are marked with $ and @ respectively. The API is described with terms from two ontologies: GeoNames (gn:) and WGS84.

Fig. 2. Karma service modeling process: (a) Web API invocation URL, (b) XML response, (c) Web API inputs and outputs in Karma's interface, (d) Semantic model in Karma's interface, and (e) formal model as a LAV rule.

ontologies). Karma automatically recommends the top four most likely semantic types (a class or a data-property/domain pair) for each column and the user assigns the semantic type by either selecting one of the suggested types or choosing another type from the ontology. After each assignment, Karma uses its Steiner Tree algorithm to recompute the tree that relates the semantic types.

The final model is shown in Figure 2(d). The service inputs (`lat` and `lng`) are mapped to the *lat* and *long* properties of a *Feature*. This feature is related via the *neighbor* object property to another *Feature*, which in turn is related to other geographical features that describe the remaining outputs (the nearby city and the parent country). Users can change the semantic types by clicking on the black circles above the column names. They can also adjust the other elements of the model by clicking on the property names (labels on the arrows) to select alternative properties or to choose alternative domains for those properties.

Figure 2(e) shows the LAV rule that captures the formal semantics of the service model that was shown in graphical form in Figure 2(d). In this rule, *Feature* is a class and *neighbourhood*, *nearby*, *name*, and *parentCountry* are properties in the GeoNames ontology, and *lat* and *long* are properties in WGS84 ontology.

4 Building a Linked API Repository

To integrate Web APIs to the Linked Data Cloud we publish an RDF representation of the API models in the Linked API Repository. In this section, we describe how we represent the linked APIs in the repository and how these declarative representations provide support for service discovery and composition.

4.1 Representing Linked APIs

Our models of Web APIs represent both the syntax and the semantics of the API. The syntactic part provides the information needed to invoke the service, including address URL, HTTP method, access credentials, and input parameters. The semantic part represents the types of the inputs and the outputs and the relationship among them in terms of a target vocabulary (ontology).

There are several vocabularies to represent linked services. WSMO-Lite[8] and Minimal Service Model[9] (MSM) are RDF vocabularies that can represent the syntax of Web APIs, but can only partially represent the semantics. They can represent the types of the inputs and outputs using terms in ontologies, but they cannot represent the relationships among them. Other approaches [5, 12] use SPARQL graph patterns to define the inputs and outputs. They can model relationships among inputs and outputs, but discovery is difficult as there are no standard facilities to query graph patterns.

We introduce an expressive ontology that extends and combines the strengths of the existing vocabularies in one model. It can represent the semantics of

[8] `http://www.w3.org/Submission/WSMO-Lite/`
[9] `http://cms-wg.sti2.org/minimal-service-model/`

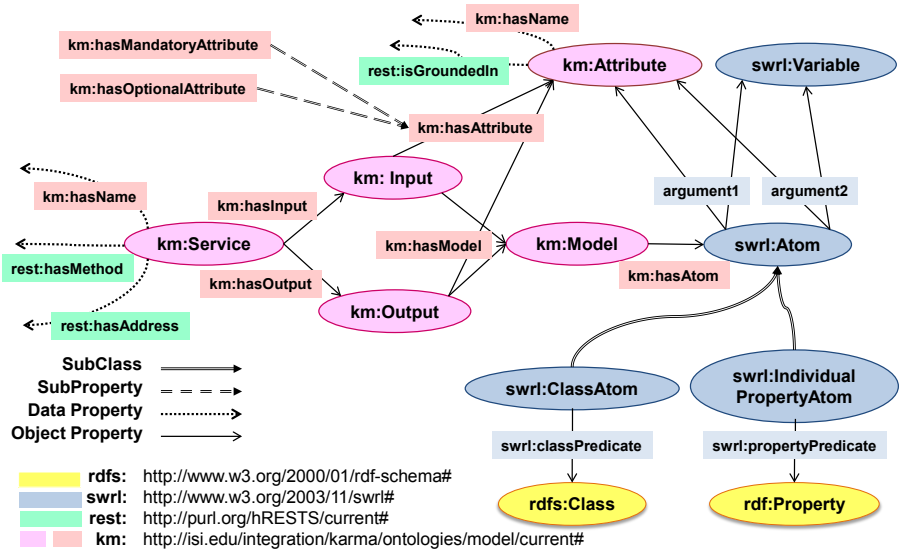


Fig. 3. The ontology that we use to formally describe Web APIs.

services including relationships among inputs and outputs, and it uses RDF(S) so that models can be queried using SPARQL.

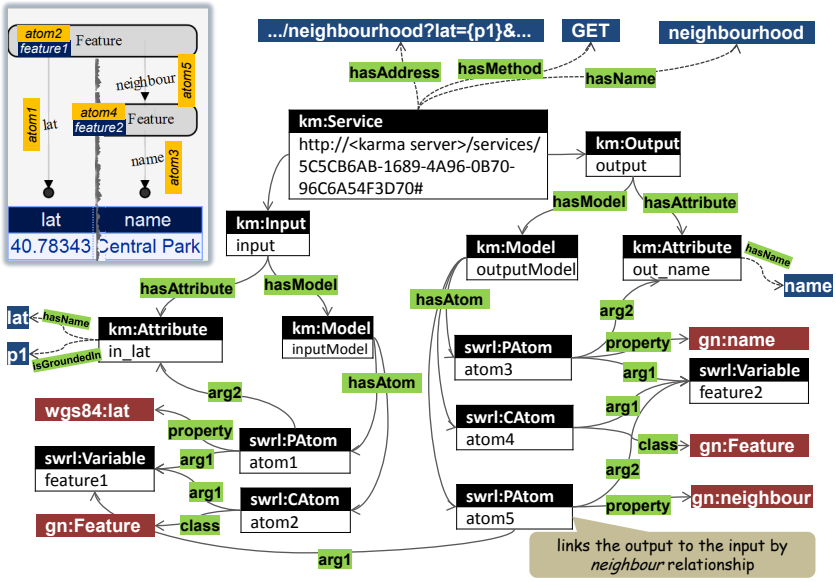
Figure 3 shows our ontology for modeling Web APIs. We re-use the SWRL[10] vocabulary to define the input and output model. In this ontology, both input and output have a *Model* which includes one or more *Atom* instances. A *ClassAtom* shows the membership of an instance to a class and an *IndividualPropertyAtom* describes an instance of a property.

We map the service semantic model (Figure 2(d)) to the introduced vocabulary by adding a *ClassAtom* for each class instance (rounded rectangles) and an *IndividualPropertyAtom* for each property (arrows). For example, to express the part of the semantic model where the top `Feature` box is connected to the the `lat` column, we first create a *ClassAtom* whose *classPredicate* has the value *gn:Feature* and its *argument1* is a new *Variable*. Then, we add an *IndividualPropertyAtom* in which the *propertyPredicate* is *wgs84:lat*, *argument1* is the same variable in the *ClassAtom*, and *argument2* is the URI of the *lat* input attribute. Figure 4 includes both graphical and N3 notation of a snippet of the `neighbourhood` API model. The input and output model is interpreted as a conjunctive formula, which is the RDF rendering of the LAV rule generated by Karma that captures the semantics of the Web API (cf. Figure 2(e))

4.2 Querying the Repository

Karma stores the API descriptions in a triple store. This linked API repository offers a SPARQL interface that can be used to discover desired APIs. Our rich

[10] Semantic Web Rule Language: `http://www.w3.org/Submission/SWRL/`




```
@prefix : <http://<karma server>/services/5C5CB6AB-1689-4A96-0B70-96C6A54F3D70#> .
@prefix gn: <http://www.geonames.org/ontology#> .
@prefix wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
...
: a km:Service;
    km:hasName "neighbourhood" ;
    hrests:hasAddress "http://api.geonames.org/neighbourhood?
                       lat={p1}&lng={p2}&username={p3}" ^^
                       hrests:URITemplate ;
    hrests:hasMethod "GET"; km:hasInput :input; km:hasOutput :output.

:input a km:Input;
    km:hasAttribute :in_lat, ... ;
    km:hasModel :inputModel .
:in_lat a km:Attribute;
    km:hasName "lat" ;
    hrests:isGroundedIn
       "p1"^^rdf:PlainLiteral .
...
:feature1 a swrl:Variable .
:inputModel a km:Model;
    km:hasAtom
       [ a swrl:ClassAtom ;
       swrl:classPredicate gn:Feature;
       swrl:argument1 :feature1 ];
    km:hasAtom
       [ a swrl:IndividualPropertyAtom;
       swrl:propertyPredicate wgs84:lat;
       swrl:argument1 :feature1;
       swrl:argument2 :in_lat];
    ...
```

```
:output a km:Output;
    km:hasAttribute :out_name, ... ;
    km:hasModel :outputModel .
:out_name a km:Attribute;
    km:hasName "name" .
...

:feature2 a swrl:Variable .
:outputModel a km:Model;
    km:hasAtom
       [ a swrl:ClassAtom ;
       swrl:classPredicate gn:Feature;
       swrl:argument1 :feature2] ;
    km:hasAtom
       [ a swrl:IndividualPropertyAtom ;
       swrl:propertyPredicate gn:neighbour;
       swrl:argument1 :feature1 ;
       swrl:argument2 :feature2];
    km:hasAtom
       [ a swrl:IndividualPropertyAtom ;
       swrl:propertyPredicate gn:name ;
       swrl:argument1 :feature2 ;
       swrl:argument2 :out_name];
    ...
```

Fig. 4. A snippet of the neighbourhood API model represented both graphically and formally (N3 notation).

models support a variety of interesting queries. The following SPARQL query finds all services that take latitude and longitude as inputs. It is difficult to support this type of query in models that use SPARQL graph patterns because the patterns are represented as strings and it is difficult to reason over them.

```
SELECT ?service WHERE {
?service km:hasInput [km:hasAttribute ?i1, ?i2].
?service km:hasInput [km:hasModel [km:hasAtom
[swrl:propertyPredicate wgs84:lat;   swrl:argument2 ?i1],
[swrl:propertyPredicate wgs84:long;  swrl:argument2 ?i2]]]}
```

In the WSMO-Lite and MSM specifications, which are in RDF, inputs and outputs of a service are linked to concepts and properties in ontologies, so they support the previous example query. However, they do not support answering more complex questions that take into account the relationships between the attributes. For example, suppose a user wants to find services that return the neighborhood regions given a latitude and longitude. These models cannot be used to answer this question because they cannot represent the relationship (`neighbour`) between the inputs (`lat, lng`) and the output regions. In our model, we can write the following SPARQL query:

```
SELECT ?service WHERE {
?service km:hasInput [km:hasAttribute ?i1, ?i2].
?service km:hasOutput [km:hasAttribute ?o1].
?service km:hasInput [km:hasModel [km:hasAtom
[swrl:classPredicate    gn:Feature; swrl:argument1 ?f1],
[swrl:propertyPredicate wgs84:lat;  swrl:argument1 ?f1; swrl:argument2 ?i1],
[swrl:propertyPredicate wgs84:long; swrl:argument1 ?f1; swrl:argument2 ?i2]]].
?service km:hasOutput [km:hasModel [km:hasAtom
[swrl:classPredicate    gn:Feature;   swrl:argument1 ?f2],
[swrl:propertyPredicate gn:name;      swrl:argument1 ?f2; swrl:argument2 ?o1],
[swrl:propertyPredicate gn:neighbour; swrl:argument1 ?f1; swrl:argument2 ?f2]]]}
```

5 Deploying Linked APIs

Publishing service descriptions into the Linked Data cloud is the first step in creating linked APIs. The next step is to deploy APIs that are able to consume data directly from the cloud and also to produce linked data. One of the benefits of our service models is that they include lowering and lifting instructions that our system can directly execute. This allows the user to easily wrap existing Web APIs without writing separate lowering and lifting specifications. In this section, we describe how we set up linked APIs and how we enable them to communicate at the semantic level (RDF).

5.1 Invoking a Linked API

As shown in Figure 1, the Karma Web server is the component that enables users to invoke the linked APIs. Users communicate with this server through a REST interface to send the RDF as input and get linked data as output. If the user sends a GET request to this endpoint,[11] he will receive the service specification in RDF. Calling the linked API with a POST request is the method to feed linked data to a service. Karma performs the following steps to execute these POST requests, which include the input RDF in its body:

- The Karma Web server extracts the service identifier from the request and uses it to retrieve the model from the repository.
- The server verifies that the input RDF satisfies the input semantic model and rejects requests that are incompatible with the input model. To do this it creates a SPARQL query according to the API input model and executes it on the input data.
- The server uses the model to do lowering, creating the appropriate URL for the original Web API (section 5.2).
- The server invokes the Web API and gets the results.
- The server uses the output model to convert the output from XML/JSON to RDF and link it to the input (section 5.2).
- The server returns the linked data to the caller in RDF.

To enable callers to determine the appropriate RDF graph that can be used as the input, the Karma Web server offers an interface[12] to get the SPARQL pattern that corresponds to the service input. The caller can execute this SPARQL query on its RDF store to obtain the appropriate input graph to call the linked API.

5.2 Lowering and Lifting

One of the advantages of our approach is that our models contain all the information needed to automatically execute the required lowering and lifting, obviating the need to manually write their specification. We describe the lowering and lifting processes using an example. Suppose that the server receives a POST request for the `neighborhood` API. The body of the request has RDF triples from the GeoNames data source with the coordinates of geographical features.

Figure 5 illustrates the lowering process. When the Karma Web server receives the HTTP POST request, it extracts the RDF triples from the body and retrieves the model from the repository. The *swrl:classPredicate* of the *ClassAtom* and the *swrl:propertyPredicate* of the *IndividualPropertyAtom* are URIs of the corresponding classes and properties in the input RDF (e.g., *wgs84:lat*), enabling the server to retrieve the corresponding values (e.g., `40.74538`). Every input attribute (e.g., *:in-lat*) has a *isGroundedIn* property which indicates its position in

[11] The address of the REST API is `http://<karma server>/services/{id}` in which `id` is the service identifier created automatically by Karma when it generates the API description.

[12] `http://<karma server>/services/{id}/input?format=SPARQL`

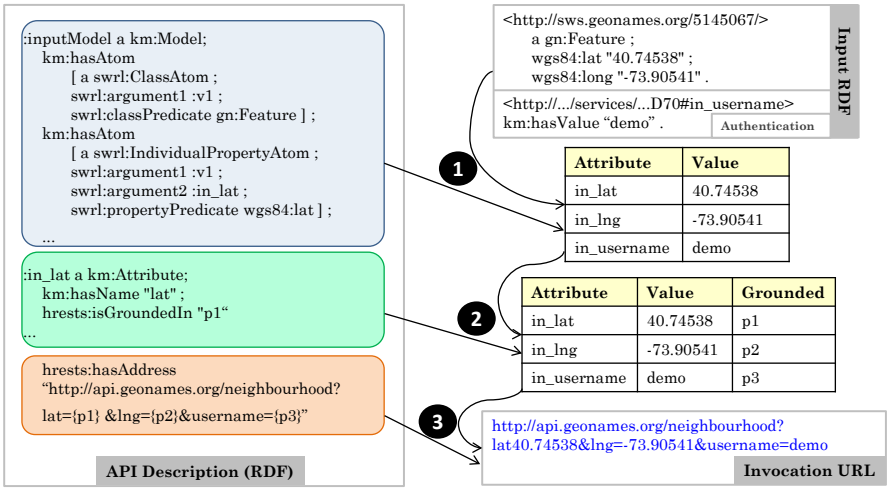


Fig. 5. Lowering the RDF data to create the invocation URL of the Web API.

the service address. This enables the server to build the invocation URL directly from the service specification without writing an explicit lowering schema. For the input parameters such as authentication key that are not part of the semantic model, the user would provide separate statements in the input graph. For instance, in the example in Figure 5, the user would add a triple, such as `<serviceURI:username km:hasValue "demo">`, to the input data.

In the next step, the Karma Web server links the outputs to the input RDF in order to return additional information about the inputs to the user. The service returns the outputs in XML or JSON, and these need to be converted to RDF. Figure 6 illustrates the lifting process. First, for each output attribute (e.g., *:out_countryCode*), Karma uses *km:hasName* property to get its name (e.g., `countryCode`). This name is compared to the XML tags to identify the corresponding value (e.g., `US`). Then, Karma exploits the output model of the API to create a RDF graph of the output values. If there is a variable in the output model that is also part of the input model (e.g., *:v1*), that means Karma already knows its value from the input RDF. For the other variables in the output model (e.g., *:v2*), Karma creates blank nodes according to their type (e.g., *gn:Featue*), denoted by *swrl:classPredicate* property. A video demonstrating how to model the neighbourhood API in Karma and wrap it as a linked API is available on the Karma web site[13].

6 Related Work

Three recent approaches address the integration of services and the Linked Data cloud. The first approach, called Linked Services [10, 11], focuses on annotating

[13] `http:/isi.edu/integration/karma`

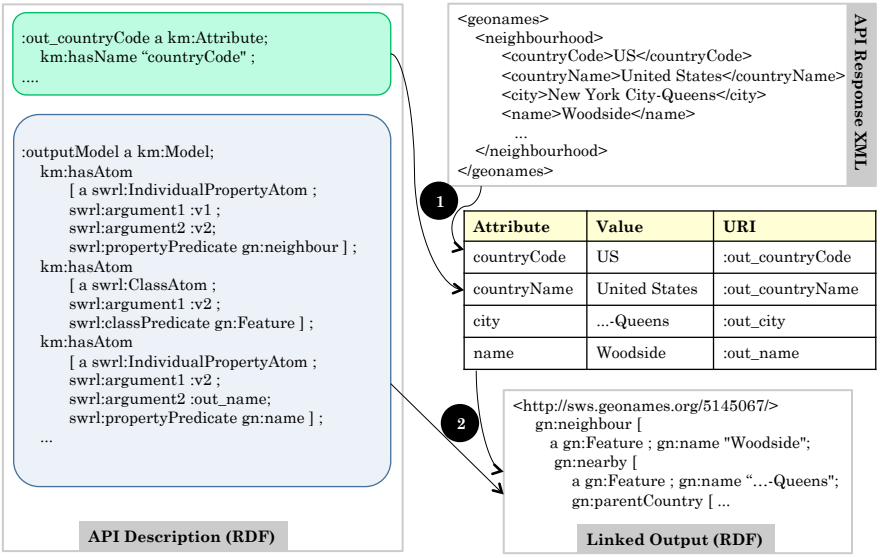


Fig. 6. Lifting the XML response to create linked data.

services and publishing those annotations as Linked Data. The second approach creates services, called Linked Open Services (LOS) [9, 5], that consume and produce Linked Data. Finally, Linked Data Services (LIDS) [12, 13] integrates data services with Linked Data by assigning a URI to each service invocation. The service URI is linked to resources in the Linked Data cloud and dereferencing the URI provides RDF information about the linked resources.

Linked Services uses a simple RDF ontology, called Minimal Service Model (MSM), to annotate the service and publish it as Linked Data. MSM uses the `modelReference` property of the SAWSDL vocabulary [1] to map service inputs and outputs to the concepts in ontologies. In Karma, rather than just annotating the service attributes, we also model the relationships between them in order to support more sophisticated service discovery and composition. Moreover, when it comes to consuming and producing RDF for existing Web APIs, MSM required modelers to provide explicit lowering and lifting schema using the `sawsdl:loweringSchema` and `sawsdl:liftingSchema` relations. In contrast, our modeling approach includes enough information to automatically derive the lowering and lifting instructions, avoiding the need to use additional languages, such as XSLT, to express the lowering and lifting scripts.

LOS and LIDS use SPARQL graph patterns to model inputs and outputs, thus providing a conjunctive logical description similar to our models. Therefore, all three approaches can model the relationships between inputs and outputs. However in LOS and LIDS, the graph patterns are represented as strings in the service description, limiting the power of automatic discovery and composition.

In contrast, Karma uses RDF graphs to model the inputs and outputs, making it possible to discover services using SPARQL queries.

One clever feature of the LIDS approach is that each service invocation has a URI that embeds the input values and this URI is linked to the input resource. Sending a GET request to the invocation URI returns the linked output. This enables the user to get the RDF output on the fly while traversing the input resource on the Linked Data cloud without the requirement to send the input data in a separate phase. Although we have not implemented this capability, generating these kinds of URIs can be done in our approach without difficulty. We can use our service descriptions to find the appropriate input values in a triple store, create the invocation URIs, and link them to the input resources.

Verborgh et al. [16] introduce a new approach, called RESTdesc, to capture the functionality of hypermedia links in order to integrate Web APIs, REST infrastructure, and the Linked Data. The idea is to enable intelligent agents to get additional resources at runtime from the functional description of the invoked API. RESTdesc uses N3 notation to express the service description. Therefore, like LIDS, LOS, and our RDF vocabulary, it can model the relationships between the input and output attributes. However, the main point that differentiates our approach from RESTdesc is that in Karma, the user interactively builds API models. After modeling, Karma automatically generates API specifications and the API modeler does not need to write the descriptions manually.

7 Evaluation

To evaluate our approach, we modeled 11 Web APIs from the GeoNames Web services as linked APIs. The purpose of the evaluation was to measure the effort required in our approach to create linked APIs. We measured this effort in terms of the average time to build the linked API model and the number of action that the user had to perform to build the correct model. We used an extended version of the GeoNames ontology, with additional classes and properties to enable us to build richer semantic models.[14]

Table 1 shows the results of our experiment. The #URLs column indicates the number of sample invocation requests given to Karma by the user as the input of the modeling process. The #Cols column counts the number of columns in the Karma worksheet that were assigned a semantic type. The Choose Type column shows the number of times that the correct semantic type was not in Karma's top four suggestions and we had to browse the ontology to select the correct type. We started this evaluation with no training data. The Change Link column shows the number of times we had to select alternative relationships using a menu. The Time column records the time that it took us to build the model, from the moment the user enters the examples until the time that Karma publishes the service description into the repository.

As the results show, using Karma it took us only 42 minutes to build the models and deploy the linked APIs for the 11 Web APIs. In addition, we did not

[14] The datasets are available at: `http://isi.edu/integration/karma/data/iswc2012`

Table 1. Evaluation results for building linked APIs from the GeoNames APIs

GeoNames API	#URLs	#Cols	#User Actions			Time(min)
			Choose Type	Change Link	Total	
neighbourhood	3	10	10	6	16	6
neighbours	2	9	5	5	10	5
children	2	10	0	5	5	3
sibling	1	9	0	5	5	3
ocean	2	3	0	2	2	1
findNearby	3	11	0	5	5	3
findNearbyPostalCodes	3	11	1	5	6	7
findNearbyPOIsOSM	3	7	5	1	6	3
findNearestAddress	3	14	4	6	10	6
findNearestIntersectionOSM	3	8	5	3	8	3
postalCodeCountryInfo	1	5	3	0	3	2
Total	26	97	76			42

have to write any RDF, SPARQL, XPath or any other code. The whole process is done in a visual user interface, requiring 76 user actions, about 7 per Web API. Equivalent LIDS or LOS models are about one page of RDF/SPARQL/XSL code written by hand. We designed Karma to enable users to build these models quickly and easily enabling them to use the ontologies that make sense for their scenarios. Karma is available as open source[15] and we plan to collect usage statistics and feedback to improve the system and measure its benefits.

8 Discussion

This paper presented our approach to rapidly integrate the traditional Web APIs into the Linked Data cloud. An API modeler uses Karma to interactively build semantic models for the APIs. The system semi-automatically generates these models from example API invocation URLs and provides an easy-to-use interface to adjust the generated models. Our models are expressed in an RDF vocabulary that captures both the syntax and the semantics of the API. They can be stored in a model repository and accessed through a SPARQL interface. We deploy the linked APIs on a Web server that enables clients to invoke the APIs with RDF input and to get back the linked RDF data.

We are working to apply our modeling approach to a large number of available Web APIs. We plan to reduce the role of the user in modeling by mining the Web for examples of service invocations in documentation pages, blogs and forums to automatically construct datasets of sample data to invoke services. We are also working on extending our approach to model RESTful APIs. Extracting the input parameters from a RESTful API is not as straightforward as a Web API because there is not a standard pattern to embed the input values in the URL. However, collecting more samples of the API requests and analyzing the variable parts of the URLs will enable us to automatically extract the inputs of RESTful APIs.

[15] `https://github.com/InformationIntegrationGroup/Web-Karma-Public`

Acknowledgements. This research is based upon work supported in part by the National Science Foundation under award number IIS-1117913. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or any person connected with them.

References

1. Farrell, J., Lausen, H.: Semantic Annotations for WSDL and XML Schema (2007), `http://www.w3.org/TR/sawsdl/`, w3C Recommendation
2. Goel, A., Knoblock, C.A., Lerman, K.: Exploiting Structure within Data for Accurate Labeling Using Conditional Random Fields. In: Proceedings of the 14th International Conference on Artificial Intelligence (ICAI) (2012)
3. Knoblock, C., Szekely, P., Ambite, J.L., Goel, A., Gupta, S., Lerman, K., Muslea, M., Taheriyan, M., Mallick, P.: Semi-Automatically Mapping Structured Sources into the Semantic Web. In: Proceedings of the 9th Extended Semantic Web Conference (ESWC) (2012)
4. Kou, L., Markowsky, G., Berman, L.: A Fast Algorithm for Steiner Trees. Acta Informatica 15, 141–145 (1981)
5. Krummenacher, R., Norton, B., Marte, A.: Towards Linked Open Services and Processes. In: Proceedings of the 3rd Future Internet Symposium (FIS) (2010)
6. Lafferty, J., McCallum, A., Pereira, F.: Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In: Proceedings of the 18th International Conference on Machine Learning (2001)
7. Lenzerini, M.: Data Integration: A Theoretical Perspective. In: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS) (2002)
8. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating Web APIs on the World Wide Web. In: Proceedings of the 8th IEEE European Conference on Web Services (ECOWS) (2010)
9. Norton, B., Krummenacher, R.: Consuming Dynamic Linked Data. In: First International Workshop on Consuming Linked Data (2010)
10. Pedrinaci, C., Domingue, J.: Toward the Next Wave of Services: Linked Services for the Web of Data. Journal of Universal Computer Science 16(13) (2010)
11. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: iServe: A Linked Services Publishing Platform. In: Proceedings of the Ontology Repositories and Editors for the Semantic Web Workshop (ORES) (2010)
12. Speiser, S., Harth, A.: Towards Linked Data Services. In: Proceedings of the 9th International Semantic Web Conference (ISWC) (2010)
13. Speiser, S., Harth, A.: Integrating Linked Data and Services with Linked Data Services. In: Proceedings of the 8th Extended Semantic Web Conference (2011)
14. Taheriyan, M., Knoblock, C.A., Szekely, P., Ambite, J.L.: Semi-Automatically Modeling Web APIs to Create Linked APIs. In: Proceedings of the Linked APIs for the Semantic Web Workshop (LAPIS) (2012)
15. Tuchinda, R., Knoblock, C.A., Szekely, P.: Building Mashups by Demonstration. ACM Transactions on the Web (TWEB) 5(3) (2011)
16. Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró Vallés, J., Van de Walle, R.: Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web. In: Proceedings of the 3rd International Workshop on RESTful Design (2012)