REST Web Service Description for Graph-Based Service Discovery

Rosa Alarcon, Rodrigo Saffie, Nikolas Bravo(✉), and Javiera Cabello

Computer Science Department, Pontificia Universidad Catolica de Chile, Santiago, Chile
ralarcon@ing.puc.cl, {rasaffie,ngbravo,jacabell}@uc.cl

Abstract. Unlike WSDL/SOAP based services, REST services lack a widely accepted service description since it increases the coupling between clients and servers, hampering service evolution. In practice, REST services are described through informal, ad-hoc and semi-structured documents, often written in natural language, which worsens the level of coupling. Most of the few REST service descriptions currently proposed follow an operation-centric approach with unclear additional benefits for developers and consumers. We propose a service description model focused on hypermedia allowing the generation of a graph that captures state transitions in an *activity layer*; we also capture resource, transition, and response semantics in a *semantic layer*. Using graph queries we traverse the graph and facilitate service discovery and composition. The service model was implemented as Microdata-based annotations, and a JSON description. A prototype was developed using Neo4J, and a set of real Web APIs was chosen to illustrate our approach.

1 Introduction

The Web is an Internet-scale distributed *hypermedia* that provides a uniform way of accessing information through embedding *action* controls within the information retrieved from remote sites (i.e. *representations*). These features have made possible for the Web to evolve from a content-distribution platform to an application platform, and nowadays, to a distributed services platform, where functionality can be integrated into new services for massive consumption.

There are two main approaches to provide Web services. One is based on WSDL/SOAP standards and is pervasive in B2B scenarios; the other called REST services or Web APIs (Application Programmable Interfaces) is pervasive in the Web. Web APIs are a popular way of providing *service connectors* while avoiding the complexity of the stack of technologies and standards that SOAP-based Web services require. In practice, Web APIs present some drawbacks such as RPC calls tunneled over HTTP, and limited service evolvability since they introduce coupling between clients and servers by means of *service descriptions* written in natural language, typically provided as HTML pages. REST services, on the other hand, require additional constraints such as content-negotiation, the appropriate use of network protocol and hypermedia (to include links and

P. Cimiano et al. (Eds.): ICWE 2015, LNCS 9114, pp. 461–478, 2015.
DOI: 10.1007/978-3-319-19890-3_30

controls in the service response) so that clients know the actions available at any point in the interaction. In practice, most of the self-called REST services lack some constraints (mainly hypermedia).

REST Web services discovery and composition consider the provision of services destined to be consumed by other services (e.g. a machine-client). Like Web APIs, REST services are generally accompanied with informal documents (e.g. HTML pages), written in natural language, describing the resource types at the application domain level and the set of *entry points* (static URIs), URI patterns, authentication mechanisms, supported protocols, operations, media types and samples requests and responses. However, service discovery and composition still lack an extensible and universal strategy that facilitates machines to interact with services [1]. HTML documentation written in natural language not only introduces high coupling between clients and servers as well as unexpected complexity (e.g. the description fails to present clearly the assumptions, configurations, requirements, preconditions or secondary effects) but also makes impossible for machines to make sense on the usage of the service interface. Hence, a common practice is the provision of libraries that embed the service description rules and hide its complexity. Such approach typically becomes outdated, inaccurate or unclear forcing machine-client developers to engage in trial-error phases to allow client recovery. Under these conditions, automatic service discovery and composition are hard to support or plainly impossible to automate. The main difficulty is that current descriptions require human intelligence to understand the service's expectations about the client. Few attempts have been made to propose machine-readable REST service descriptions. Moreover, such descriptions fail to accommodate the REST architectural constraints, and focus extensively on the operations provided by the services instead of the flexibility that the REST uniform interface offers, such as the hypermedia constraint.

In this paper we present RAD (REST API Description), which is a service description that considers the REST uniform interface constraint, that is, identification of resources; manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state. We present two implementations of RAD: in JSON, allowing the generation of human-readable documentation; and in Microdata that can be embedded in the HTML service description. Both implementations are based on the RAD metamodel which considers an *activity* layer, capturing resource state transition mechanism, and a *semantic* layer, which captures the resource's and transition's semantics.

The result is a graph that captures both the underlying graph of state transitions modeled by a REST service and the semantic graph associated. Service discovery consists of discovering the nodes corresponding to entrypoints, and service composition corresponds to the discovery of paths in the graph that allows satisfying certain goal. We described three popular Web APIs using RAD, and we implemented a couple of use cases to demonstrate the feasibility of our approach.

2 Background

REST, the *Representational State Transfer* [2], is an architectural style that determines a set of constraints such as functional extensibility (i.e. code on demand) and stateless client-server interaction (i.e. the server shall not store context information about the client, which remains responsible for storing such information and providing it as-needed in the request messages). It also requires interaction visibility, metadata providing cache control information, and a uniform interface between *architectural* components. The uniform interface requires the identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.

Data is the central element in REST, being the *resource* the basic information abstraction, which has associated a *resource identifier*, minted by the resource author. A resource is a conceptual mapping to a set of entities or *representations* whose format is negotiable. Resources are abstract and are realized by representations making unnecessary to classify resources according to a type or implementation (since the representation is served to the client, whereas the resource itself is hidden by the server). Representations comprise data and *metadata* (about the data, the *representation*, the *resource*, etc.) indicating the current or intended resource state, or the value of other resource.

Client's requests include *control data*, a resource identifier, and an optional representation. Servers perform actions on the resources according to the request and provide a response that may include control and resource metadata, and an optional representation. Control data determines the purpose of the message (requested action, response meaning) and the actions of intermediary components such as caches. Representation's data format is known as *media type*; representations convey the set of state transitions available for a resource at a determined state in the form of links and controls (from now on just links). User agents (e.g. Web browsers) behave as engines that move a Web application from one state to the next according to the end-user actions (i.e. by executing a link or control).

3 Related Work

3.1 REST Service Description

Some languages have been proposed to create REST services descriptions. For instance, WSDL [3] is a standardized language used to describe WSDL/SOAP services and REST services, it follows an RPC operation-oriented style and do not support hypermedia, content negotiation or metadata. The Web Application Description Language (WADL) [4] describes REST services in terms of *resources*, URI patterns, media types and schemas of the expected *request* and *response*. Representations support parameters with links to other resources. However WADL do not provide support for link discovery or URI generation for new resources, ignoring the dynamic nature of REST. The resulting model is operation-centric and introduces additional complexity with unclear benefits

for both human and machine-clients. These descriptions are kept independently from the service so that maintainability issues may arise [5].

Semantic descriptions have been also proposed for REST services. For instance, RESTdesc [6] expresses REST service functionality in RDF, including a request's preconditions, postconditions, and quantifiers. Resources are typed and links semantics are also considered. The description itself is highly flexible and complex; it requires knowing the resources' URIs in advance in order to perform advanced queries. Less complex approaches are SA-REST [7] and hREST [8]; both propose a new resource (e.g. an HTML document) containing the input and output parameters, methods, and URIs, written as RDFa property value pairs (SA-REST) [9] or Microformat annotations [10] (hREST). Links and forms are supported but dynamic discovery of resources following such links is not considered. Microformats are not extensible and they require specific processing rules for each domain [1], [11]. RDFa solves Microformats limitations since they are generic and can be combined and extended, however, it is more complex [12].

RESTdoc [13] is a framework based on Microformats including adapters to transform the data to RDFS automatically, reducing the development complexity. RESTdoc also includes machine-learning techniques to infer the graph of interlinked resources from the user behavior; however, it ignores hypermedia. [14] and [15] go a step forward and infer the semantic model from request-response samples or from the HTML documentation pages. Like RESTdoc, they infer relationships between resources but ignore the hypermedia links. Hydra [16] is a vocabulary that allows providers to describe a Web API in terms of a main entry point, and a set of resources (classes), properties, and links, which are CRUD operations over HTTP. Resources and link semantics are handled as references to external entities following a JSON-based format (JSON-LD). Hydra moves a step forward in recognizing the relevance of hypermedia, but introduces non-REST concepts such as CRUD operations.

Industry approaches such as RAML, the RESTful API Modeling Language (http://raml.org/), proposes a YAML and JSON open specification for describing practical REST APIs that are human and machine readable, but leaving out the issues related to the REST hypermedia constraint. Similar approaches are proposed by Swagger (http://swagger.io/), Blueprint (https://apiblueprint.org/), and Mashape (https://www.mashape.com/), which provide either a resource-oriented description with various degrees of completeness or even a framework that generate mockup code. Again, neither of them includes hypermedia support nor a strategy for service discovery or composition.

3.2 REST Service Discovery

REST Service Discovery techniques follow either a directory-based (services are registered in either a centralized or a decentralized directory), directory less (a P2P overlay structure plays the directory role) or a hybrid infrastructure. The discovery, selection or matchmaking itself is performed by similarity algorithms

that follow a logic-based (reasoning) or non-logic based (data mining, graph matching, schema matching, text similarity, among others) approach [17].

Nowadays the de facto service open central repository is *ProgrammableWeb* (www.programmableweb.com) with 12.889 Web APIs (retrieved by February 2015). It hosts self-declared REST APIs, but in practice many of them partially satisfy the REST constraints. The lack of a standardized service description for REST has a negative effect on global service discovery. For instance, in Programmable Web, services search is based on keyword match of service's description and metadata. Service discovery is a difficult task due to its large number, diversity of names and technical aspects.

In [18] a model for naming service interfaces from their objects, processes and business logic is proposed. An iterative search engine that filters the results based on the resources relationships enables service search. In [19], the goal is to discover similar service's operations. WADL described services are stored in a centralized repository; service's operation match is based on common parameters, using WordNet [20] to exploit similarity of synonyms. Another centralized repository, iServe, stores hREST described services as graphs interconnecting resources and parameters at the semantic level [21]. Service discovery consists of identifying resources and parameters using SPARQL queries on the graph.

A REST service is a collection of addressable and dereferenceable resources interconnected by links that determine the possible state transition for each resource. Hence, we focus on entrypoint discovery, that is, the links that make possible for a user to perform an action and achieve a goal.

3.3 REST Service Composition

Dynamic service composition is an important challenge, it requires generating a composition plan (control flow), selecting the required Web services, coordinating conversations between services (including data flow) and executing the composition at runtime. It is typically based on business constraints, on planning with user interaction, on context, on the data model, or on the service's signatures (inputs and outputs). REST service composition research focuses on orchestration, with JOpera [22] being the most complete framework. JOpera models control and data flow visually, and an engine executes the resulting composed service. In [23], control flow is specified in SPARQL and services could be WSDL/SOAP-based endpoints or REST resources. In [24] control and data flow is modeled and implemented using a Petri Net whereas interaction and communication with the resources themselves is mediated by a service description called ReLL [25]. ReLL is based on a hypermedia-oriented metamodel where links are considered first-class citizens. In [26] the BPEL composition language is extended through specialized activities that make possible resources invocation. Hypermedia constraint is implemented via inspection of responses. Data-Fu [27] is an RDF-based declarative language that allows developers to express service interaction as HTTP operations to change resource's state and retrieve response's data (which is an RDF graph). The hypermedia constraint is not considered and only links between resources (at the semantic level) can be discovered.

4 An Illustrative Example

In order to illustrate our approach we present the following scenario consisting of three Web APIs:

Spotify https://developer.spotify.com/web-api/,
Songkick https://www.songkick.com/developer/,
Uber https://developer.uber.com/.

The Spotify API provides access to its music streaming service's catalogue. The Songkick API provides access to a live music database with information about upcoming and past concerts as well as setlists. The Uber API allows the user to ask for transportation service types, price and an estimated time of arrival (ETA), as well as user's profile and activity. Let's consider the following use cases for service discovery and composition:

Discovery:
A user wants to find a given artist's playlists.
Such goal can be satisfied by GET operations (i.e. reflection) on resources provided by both Songkick, which provides the setlist for a given musical event, and Spotify, which provides an artists top tracks. The interfaces of both services are quite different from one another, so it is necessary to rely on its semantics rather than the syntactic description.

Composition:
A user wants to attend a concert but her favorite band is not in town. She decides to go to a similar artist's concert and needs to know the venues where they will play as well as information of taxis to reach the event.
Such goal can be satisfied by GET operations (i.e. retrieve artists and concerts data, consult on prices, etc.) on resources provided by Songkick or Spotify (similar artists, venues). The interaction with Uber requires to consult for fares and ETA (through GET) in order to request a taxi later. Again, interfaces are heterogeneous and service's semantics will make possible to discover the appropriate resource. In addition, it is necessary to follow various steps and support a failure path (e.g. there may be no similar band, the venue could be too far away in distance or time, or the cab price may be too high or too late). Also it is important to consider the semantics of the operations since a resource may include several links in their representation corresponding to various GETs, POSTs, etc., each with different semantics.

In the following sections we present RAD, a metamodel for REST service description, as well as the implementation of the described scenarios.

5 REST API Description (RAD)

The proposed metamodel is depicted in Figure 1. We separate modeling elements into two layers: a semantic layer that captures semantics of resources, parameters

and actions; and an activity layer that is responsible for modeling REST service's interaction components. In our model, links are first-class citizens and represent an `Action` at a business level of abstraction.

`Actions` are key in our model, they are associated to operations, that is, a network protocol method (e.g. HTTP GET, HTTP POST, etc.) applied to a given `Resource`. Actions semantics or `Action Concept` denotes the intention of the operation at a business level (e.g. to buy, to rent, to search, etc.). Actions encapsulate state transition operational details and semantics. For instance, depending on the parameters, the resources https://api.spotify.com/v1/search will respond a GET operation with representations of *albums*, *artists*, *playlist* and *tracks*. At an operational level (i.e. HTTP), these invocations are the same, but at a business level they convey different semantics. Hence, we model each request as a separate link that is composed of one operation (e.g. an HTTP method) performed on a resource, requiring (or not) a set of parameters. An operation execution produces a response or representation that corresponds to a concept.

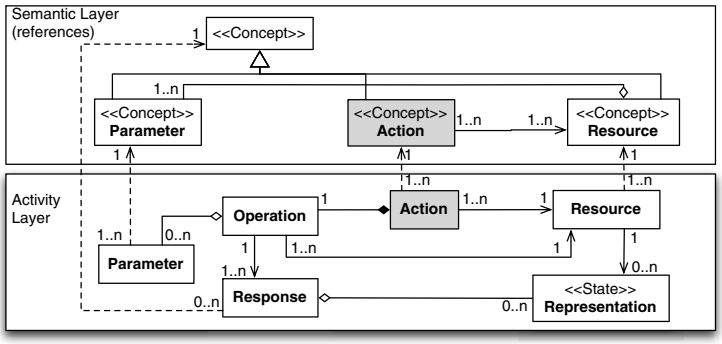

Fig. 1. The RAD metamodel includes a semantic and an activity layer

5.1 Semantic Annotations in RAD

Semantic descriptions have clear advantages; they can be integrated and used in combination with HTML hierarchy enriching its elements and attributes. Lightweight data formats such as Microformats and Microdata are easy to use by developers, they do not interfere with the information to be presented to the end user, and they can be understood by machines, while keeping all the information in one file. Semantic descriptions seem to be a good alternative to facilitate the automation of service discovery and invocation. In our model, Parameters, Resources and Links are associated with a corresponding Concept that models the semantics of each element in a Semantic Layer.

In our case, we extended Schema.org (http://schema.org/) data model with a set of concepts following the RAD metamodel as seen in Figure 2(a). We also implemented the RAD metamodel using JSON for creating a separate and

complete description of a RESTful web service as seen in Figure 2(b). In this case, we do not determine the representation of such semantics but follow the SAWSDL/SAREST [28] approach in that the binding between elements in each layer is achieved through a model reference attribute that links and maps service elements to the semantic model.

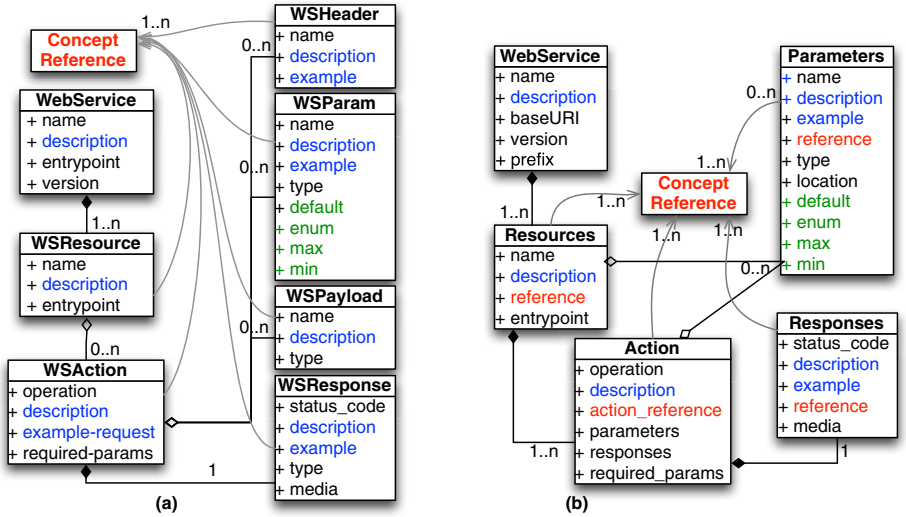

Fig. 2. The RAD metamodel implemented following Microdata and Schema.org principles (a) as well as a JSON format (b)

5.2 RAD Implementation in Microdata

We implemented the RAD metamodel using Microdata for the semantic annotation of RESTful web services (i.e. HTML description pages). Microdata is simple, flexible and more adaptable than other techniques for semantic description such as Microformats or RDFa [1]. Microdata syntax is simple and includes the *itemscope*, *itemtype* and *itemprop* HTML attributes. Each HTML content element is considered an item, each item consists of key-value pairs where *itemprop* is the key and the HTML content element is the value. Itemscope defines a new item that corresponds to the *subject* being described, while itemtype specifies its *type* (i.e. the associated concept). Both itemtype and itemprop can take any value, but in order to facilitate that concepts are understood properly, such values must correspond to a shared vocabulary between service providers and consumers.

The Schema.org data models and Microdata results in a simple structure that facilitates adding explicit semantics to HTML content without additional complexity. Schema.org vocabulary (derived from RDF Schema, as a semantic model) is a hierarchical vocabulary of markup schemas for primitive and specialized data types or `Things`, such as `Action`, `Events`, etc. It is promoted by Bing,

```
<div itemscope itemtype="http://schema.org/WebService">
  <h1 itemprop="name">Uber API<h1>
  <meta itemprop="entrypoint" content="http://developer.uber.com/">
  <div itemprop="resource" itemscope itemtype="http://schema.org/WSResource">
    <meta itemprop="entrypoint" content="/v1/estimates/time">
    <h2 itemprop="name">TIME ESTIMATES</h2>
    <p itemprop="description">The Time Estimates endpoint returns ETA's... </p>
    <h3>Resource</h3>
    <div itemprop="action" itemscope itemtype="http://schema.org/WSAction">
      <meta itemprop="reference" content="http://schema.org/Action/GetAction/GetTaxiTravelTimeCollectionAction">
      <meta itemprop="required_params" content="start_latitude AND start_longitude">
      <p><span itemprop="operation">GET</span> /v1/estimates/time<p>
      <h3>Query Parameters</h3>
      <ul>
        <li itemprop="param" itemscope itemtype="http://schema.org/WSParam">
          <p>Name: <span itemprop="name">start_latitude</span></p>
          <p>Type: <span itemprop="type">float</span></p>
          <p>Description:  <span itemprop="description">Latitude component of location.</span></p>
          <meta itemprop="reference"
           content="http://schema.org/Thing/Intangible/StructuredValue/GeoCoordinates/latitude">
        </li>
        <li itemprop="param" itemscope itemtype="http://schema.org/WSParam">
          <p>Name: <span itemprop="name">start_longitude</span></p>
          <p>Type: <span itemprop="type">float</span></p>
          <p>Description:  <span itemprop="description">Longitude component of location.</span></p>
          <meta itemprop="reference"
           content="http://schema.org/Thing/Intangible/StructuredValue/GeoCoordinates/longitude">
        </li>
      </ul>
      <h3>Response</h3>
      <div itemprop="response" itemscope itemtype="http://schema.org/WSResponse">
        <meta itemprop="status_code" content="200">
        <meta itemprop="media" content="application/json">
...
```

Fig. 3. A snippet depicting the usage of Microdata and Schema.org principles for describing the *Time estimates* resource, for the arrival of a taxi in Uber

Google and Yahoo! as HTML markup that Web search engines use to improve the display of search results. The schemas extensibility mechanism is based on the "/" separator to specify "paths" that specializes a data type or a property. `Data Types` and `Enumerations` are written in camelcase starting with a capital letter, whereas `Properties`, also written in camelcase, start with a lowercase.

Schema.org does not consider descriptions for web services (neither WSDL/-SOAP nor REST). A `WebService` is the entity that encompasses the service elements; a service can contain one or more REST resources (`WSResource`). Certain actions may be performed on resources (`WSAction`) specifying the HTTP required operation (e.g. GET, POST, etc) and parameters (`WSParam`) to be sent in the request as well as their `type`, `default` values and `enumerated` values. Parameters may be included as headers (`WSHeader`) or in the payload (`WSPayload`). Attributes such as `name` and `example` and `description` are included only for documentary purposes. The expected `response` of the operation is also included. All attributes are optional; they may or may not be present in the description.

The model was applied to a Web API available on the Web: the Uber Products API https://developer.uber.com/v1/endpoints/. Figure 3 presents a simplified snippet of our Microdata approach for annotating the `Time Estimates` resource of the Uber Products API. For simplicity, we have removed style classes.

5.3 RAD Implementation in JSON

The JSON implementation satisfies two requirements, to generate human-readable documentation, and to allow machine clients to understand the rules under which they must interact with the service. An implementation overview can be seen in Figure 2(b). A REST Web Service is described by a human-readable `name` and `description` elements, a `base URI` (it may refer to the root service entrypoint), and a `version` identifier (a string). As observed in Figure 4, a *prefix* is a key-value namespace (starting with the '@' symbol) that shortens references to semantic elements (*Concept* in Figure 2(b)).

```
{
 "name": "Songkick",
 "baseURI": "http://api.songkick.com/api/3.0",
 "version": "3.0",
 "description": "The Songkick API gives you easy access to the biggest live music ....",
 "prefixes": {
  "@schema": "http://schema.org",
  "@Thing": "@schema/Thing",
  "@Action": "@schema/Action",
  "@apikey": "@Thing/CreativeWork/SoftwareApplication/WebApplication/apikey/songkickApiKey",
  "@identifier": "@Thing/identifier",
  "@artist_id": "@identifier$Organization/PerformingGroup/MusicGroup/songkickId",
  "@Collection": "@Thing/Collection",
  ...
 },
 "resources": {
    "/artists/{@artist_id}/similar_artists.json": {
       "name": "Collection of similar artists",
       "reference": "@Collection/MusicGroupCollection",
       "description": "A list of artists similar to a given artist, based on our tracking and ....",
       "actions": {
          "get": {
              "description": "Get a list of artists similar to a given artist.",
              "reference": "@Action/SearchAction/SearchMusicGroupAction",
              "additional_doc": "http://www.songkick.com/developer/similar-artists",
              "required_params": "apikey",
              "parameters": {
               "apikey": {
                 "name": "API Key",
                 "description": "Your API Key",
                 "reference": "@apikey",
                 "type": "string",
                 "example": "ABC123DEFG"
               }
              },
              "responses": {
                  "200": {
                     "description": "On success, the HTTP ...",
                     "media": [
                        "application/json"
                     ],
                     "reference": [
                         "@Collection/MusicGroupCollection"
                     ]
..
```

Fig. 4. A snippet depicting the usage of JSON for describing the *Similar Artists* resource

Unlike the Microdata model, the JSON model describes a set of resources; hence, Web services are composed of `Resources` described by a human-readable `name` and `description` elements, and a conceptual entity representing its semantics through a `reference` (a URI). `Resources` are the keystones of our description. Each `Resource` has an entrypoint, some entrypoints are absolute URLs whereas others refer to resource instances and hence the URL includes parameters whose values must be solved at runtime (`type="path"`); such parameters are also associated (reference) to a concept. `Resources` as a whole have also a reference to its corresponding resource `Concept`, as well as a human-friendly `name` and `description`.

`Actions` are associated to operations, that is, a network protocol method (e.g. HTTP GET, HTTP POST, etc.) applied to a given `Resource`. Each operation has a reference to an `Action Concept` as well as a human-friendly `description`. Operations have also `Parameters` and a logical expression that allow developers to specify the rules that determine which parameters are required. `Parameters` have a `name`, `type` (boolean, string, etc.), a location (path, url, header, or body), a `reference` to a concept, and optional specifications depending on each type such as `maximum`, `minimum`, `enumeration`, and `default` values. An `example` of possible values is also considered. Finally, a `Response` fully connects the graph by containing a reference to a `Concept`, which is expected to be returned together with an HTTP `status code`, `description` and `media type`. Naturally we are describing the characteristics of an expected response, however due to the dynamic nature of REST the actual response may vary.

5.4 Prototype Implementation

We implemented a prototype of our proposal storing Web services information (nodes and arcs) in the graph database Neo4j. This database was chosen because of its high flexibility, performance and scalability. There are two possible inputs for the current implementation: a Microdata annotated HTML description, and a JSON description. We implemented two Python parsers, the first one transforms Microdata annotated HTML descriptions into JSON; the second one processes JSON descriptions and generates the nodes and arcs to be stored in the database using the Py2neo library. JSON descriptions are validated by JSON Schema before being parsed.

6 Validation

Service descriptions, either in Microdata or JSON implementation, are parsed to generate a graph (Figure 5). Nodes and arcs are labeled with attributes: rounded rectangles for nodes, and square rectangles for arcs. Nodes and arcs have an internal identifier in the graph, GRI (Graph Resource Identifier).

`Resources`, `Operations`, `Parameters`, and `Responses` are nodes in the graph, whereas `Actions` become a semantic arc connecting resources and operations. `Resources`, `Actions`, `Responses`, and `Parameters` are associated to `Concepts`

through a `reference` arc. `Concepts` themselves are nodes described by two attributes, a `URI` and a `label` indicating its type. `Concepts` can be part of a complex semantic model such as an ontology (e.g. see relationship "isA" from node 6 to 3) or not, depending on the providers choice. For the case of parameters, they are classified into 4 types (path, url, header, and body) and they refer to a `Concept Parameter`. Relationships between RAD activity layers are also included and annotated with the semantics of the relationship (e.g. uses and results) and a GRI identifier. Figure 6 presents the result obtained when parsing the JSON and Microdata descriptions corresponding to the three APIs described in the illustrative example.

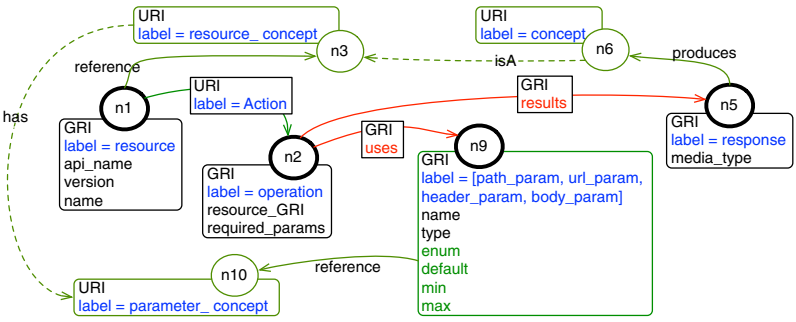

Fig. 5. The structure of the graph produced from JSON and Microdata implementation

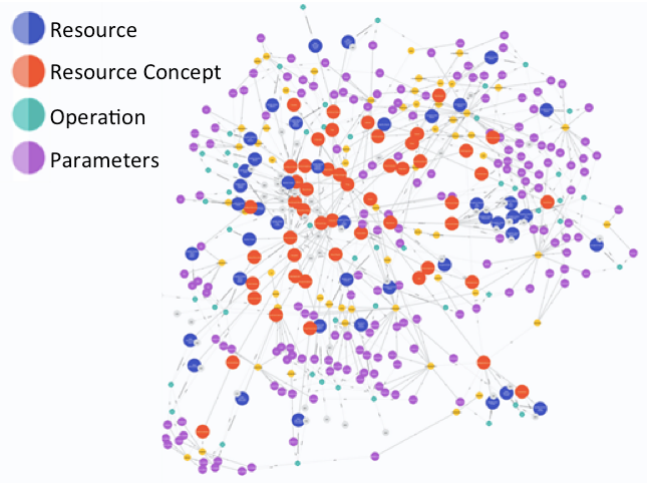

Fig. 6. The resulting graph, after parsing Spotify, Songkick and Uber APIs

```
(a)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters)
WHERE concept.GRI=~ 'http://schema.org/Thing/CreativeWork/MusicPlaylist.*'
AND action.GRI = 'http://schema.org/Action/GetAction/GetMusicRecordingCollectionAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter
RETURN concept, resources, operations, parameters

(b)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters), (operations)-[:results]-(responses)
WHERE concept.GRI = 'http://schema.org/Thing/Collection/MusicGroupCollection'
AND action.GRI = 'http://schema.org/Action/GetAction/GetMusicGroupCollectionAction/GetSimilarArtistsAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter  AND responses: Response
RETURN concept, resources, operations, responses, parameters

(c)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters), (operations)-[:results]-(responses)
WHERE concept.GRI = 'http://schema.org/Thing/Collection/MusicEventCollection'
AND action.GRI = 'http://schema.org/Action/SearchAction/MusicEventSearchAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter  AND responses: Response
RETURN concept, resources, operations, responses, parameters

(d)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters), (operations)-[:results]-(responses)
WHERE concept.GRI = 'http://schema.org/Thing/Collection/TaxiCollection'
AND action.GRI = 'http://schema.org/Action/GetAction/GetTaxiCollectionAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter  AND responses: Response
RETURN concept, resources, operations, responses, parameters

(e)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters), (operations)-[:results]-(responses)
WHERE concept.GRI = 'http://schema.org/Thing/Collection/TaxiFareCollection'
AND action.GRI = 'http://schema.org/Action/GetAction/GetTaxiFareCollectionAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter  AND responses:Response
RETURN concept, resources, operations, responses, parameters

(f)
MATCH (concept)<-[:`is a`]-(resources)-[action]->(operations)-[:uses]->(parameters), (operations)-[:results]-(responses)
WHERE concept.GRI = 'http://schema.org/Thing/Collection/TaxiTravelTimeCollection'
AND action.GRI = 'http://schema.org/Action/GetAction/GetTaxiTravelTimeCollectionAction'
AND concept: `Resource Concept`  AND resources: Resource  AND operations: Operation
AND parameters: Parameter  AND responses:Response
RETURN concept, resources, operations, responses, parameters
```

Fig. 7. Cypher queries to implement service discovery (a) and composition (b, to f) according to the example in section 4

6.1 Discovery

Let's consider again the Discovery example: *A user wants to find a given artist playlists*. Assuming that the user knows the Schema.org vocabulary, she must search for all `Resource Concepts` that are an instance of a `Musical Playlist` data type, as well as the parameters and operation required performing the HTTP request. Figure 7(a) shows such query, written in Cypher [29] which follows a syntax similar to SQL. As can be seen in Figure 7, Cypher queries are composed of three sections: the `MATCH` statement traverses the graph, through nodes (declared with parenthesis), relationships (if specified, declared with square brackets) and a direction (if specified, declared with arrows). When executed, the `MATCH` statement stores the nodes and relationships into variables. The `WHERE`

statement imposes restrictions on such variables; and the `RETURN` statement determines the variables to be returned.

The query described in Figure 7(a) searches for graph fragments that hold the following form: *resource* nodes are linked through *action* arcs to *operation* nodes, which in turn are linked to *parameter* nodes through a `uses` arc (`MATCH` statement). In this case, the intended Action is to obtain a collection of music recordings (`action.GRI="/GetMusicRecordingCollectionAction"`); the Resource Concept is the set of music playlists (`concept.GRI="/MusicPlaylist"`). The result shown in Figure 8 reveals that there are two available operations for the given `Action` and `Resource Concept`, one provided by Songkick (a setlist for a given concert), and one by Spotify (the artists top tracks list).

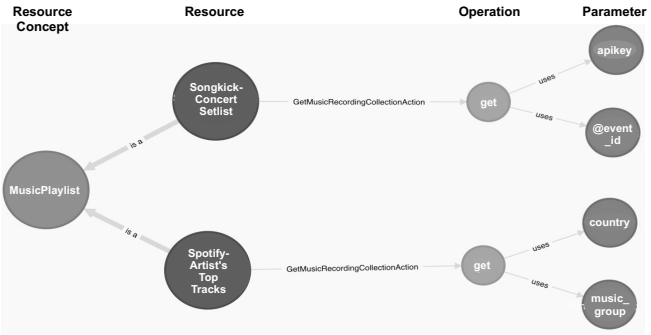

Fig. 8. Music playlists provided by Spotify and Songkick according to query (a) in Figure 7

6.2 Composition

Let's consider the Composition example: *A user wants to attend a concert but her favorite band is not in town. She decides to go to a similar artist concert and needs to know the venues where they will play as well as information of taxis to reach the event.* Again, assuming the user knows the Schema.org vocabulary, she needs to perform the following set of queries:

First, it is necessary to discover services allowing to search for similar artists (`action.GRI="/GetSimilarArtistsAction"`). Such services must provide resources of the `MusicGroupCollection` kind as a result. Figure 7(b) shows a Cypher query that follows the rules described in section 6.1. The query results are shown in Figure 9(a), we can observe that both Spotify and Songkick provide such feature although through different resources (*Related Artists* and *Similar Artist Collection*, respectively) and parameters. Semantically equivalent resources refer to the same concept in the extended Schema vocabulary. Equivalence between these resources is explicitly defined at a conceptual level in order to implement a prototype as a proof of concept.

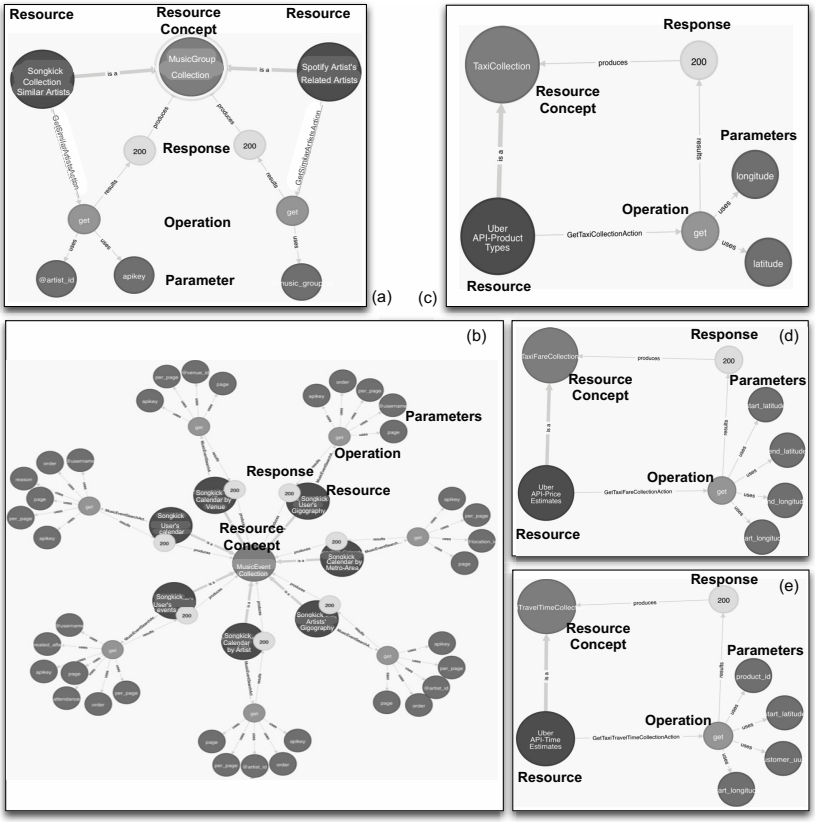

Fig. 9. Results for the various composition queries b to f in Figure 7

Afterwards, the user must search for information about the shows of the artists obtained in the previous query (Figure 7(c)). In this case, only Songkick can provide such functionality although through 7 resources (e.g. User's events, User's calendar, Calendar by Venue, etc.) as shown in Figure 9(b).

The next step is to find information about taxi availability to the found venues (Figure 7(d)). Now, only one service (Uber) provides an entrypoint to such request: `"/GetTaxiCollectionAction"` in Figure 9(c). Notice that in this case, the resource is more general than a *Taxi* concept, so it provides information of various product types, been one of them, the *TaxiCollection*. Queries (e) and (f) in Figure 7 presents the next steps: to find the estimated price and the estimated travel time and provide responses similar to the request for Taxi availability as shown in Figure 9(d) y (e).

Notice that, at this point, service composition cannot be performed automatically since it is necessary to support data flow. For instance, in order to find similar artists the user must provide information such as the *artists_id* and

apikey for the case of Songkick and the *music_group* name for the case of Spotify. Once the service and the action to find music event venues are identified, the user must indicate the artist's name to locate the right venue. Only two resources out of the 7 found supports the *artist_id* and *apikey* parameters, the *Artist's Gigography* resource and the *Calendar by Artist* resource, both require additional configuration parameters such as *page*, *order*, and *per_page*. However the semantics of the former resource refers to past events, whereas the latter resource refers to upcoming events. In all these cases, it is necessary the human intervention to resolve ambiguities and clarify her preferences.

7 Conclusions

Our main goal is to facilitate the discovery of REST web services as well as their possible compositions, demonstrating the advantages of including hypermedia in REST service descriptions. In previous work we have demonstrated the advantages of following a fully decentralized, stateless (i.e. REST compliant) approach in terms of scalability and availability [30]. Two fundamental assumptions have been made in this paper: service providers will follow a shared vocabulary in order to semantically annotate their services (e.g. Schema.org); and that services responses follow a hypermedia approach interconnecting services through links. Various techniques have been proposed to automatically relate resources and concepts, including logic-based and non-logic based approaches [31]. We follow a non-logic, heuristic, and convention based approach since we are implementing a proof of concept tool, however our intention is to continue exploring a natural language processing, and non-logic approach.

Our proposal adds an extra layer to the REST architecture (i.e. business level supported by REST services) making maintenance a concern. A pending and no less important issue is data flow between operations, that is, how to transform response data into parameters to execute a new operation. The challenge not only relies on casting different data types (e.g. string, boolean, integer), but also on respecting and producing data that is semantically equivalent and which format could vary significantly (e.g. hash codes for different APIs that are produced following different protocols).

Natural language descriptions pose an additional challenge for machine-clients making it clear the need for introducing service semantics. Annotations seem to be a good alternative to achieve this goal, and Microdata is a good fit due to its simplicity, although other approaches such as Linked Data could be used. This proposal aims to provide a strategy for the discovery and composition of REST Web services, which foster service descriptions understandable by both developers and machines. As future work, we will focus on the discovery of relationships between resources at an activity and semantic level, as well as a more friendly way of expressing users needs.

References

1. Mayer, S., Guinard, D.: An extensible discovery service for smart things. In: Proceedings of the Second International Workshop on Web of Things, p. 7. ACM (2011)
2. Fielding, R.T.: Architectural styles and the design of network-based software architectures, Ph.D. dissertation, University of California, Irvine, Irvine, California (2000)
3. Chinnici, R., Gudgin, M., Moreau, J.-J., Schlimmer, J., Weerawarana, S.: Web services description language (wsdl) version 2.0 part 1: Core language, W3C working draft, vol. 26 (2004)
4. Hadley, M.: Web application description language, World Wide Web Consortium, Member Submission SUBM-wadl-20090831 (August 2009)
5. John, D., Rajasree, M.: Restdoc: Describe, discover and compose restful semantic web services using annotated documentations. International Journal of Web & Semantic Technology (IJWesT) 4(1) (2013)
6. Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., Gabarró Vallés, J.: Description and interaction of restful services for automatic discovery and execution. In: Proceedings of the FTRA 2011 International Workshop on Advanced Future Multimedia Services (2011)
7. Lathem, J., Gomadam, K., Sheth, A.P.: Sa-rest and (s)mashups: Adding semantics to restful services. In: First IEEE International Conference on Semantic Computing (ICSC 2007), Irvine, California, pp. 469–476, September 2007
8. Kopecký, J., Gomadam, K., Vitvar, T.: Hrests: An html microformat for describing restful web services. In: 2008 IEEE/WIC/ACM International Conference on Web Intelligence, Sydney, Australia, pp. 619–625, December 2008
9. Adida, B., Birbeck, M., McCarron, S., Pemberton, S.: Rdfa in xhtml: Syntax and processing – a collection of attributes and processing rules for extending xhtml to support rdf, World Wide Web Consortium, Recommendation REC-rdfa-syntax-20081014, October 2008
10. Khare, R., Çelik, T.: Microformats: a pragmatic path to the semantic web. In: Proceedings of the 15th International Conference on World Wide Web, pp. 865–866. ACM (2006)
11. Adida, B.: hgrddl: Bridging microformats and rdfa. Web Semantics: Science, Services and Agents on the World Wide Web **6**(1), 54–60 (2008)
12. Bizer, C., Eckert, K., Meusel, R., Mühleisen, H., Schuhmacher, M., Völker, J.: Deployment of RDFa, Microdata, and Microformats on the Web – A Quantitative Analysis. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 17–32. Springer, Heidelberg (2013)
13. John, D., Rajasree, M.: Restdoc: Describe, discover and composerestful semantic web services using annotated documentations. International Journal of Web & Semantic Technology (IJWesT) 4(1) (2013)
14. Taheriyan, M., Knoblock, C.A., Szekely, P., Ambite, J.L.: Semi-automatically modeling web apis to create linked apis. In: Proceedings of the First Linked APIs workshop at the Ninth Extended Semantic Web Conference, May 2012. http://lapis2012.linkedservices.org/papers/2.pdf
15. Ly, P.A., Pedrinaci, C., Domingue, J.: Automated Information Extraction from Web APIs Documentation. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 497–511. Springer, Heidelberg (2012)

16. Lanthaler, M., Gütl, C.: Hydra: A vocabulary for hypermedia-driven web apis. In: LDOW, Citeseer (2013)
17. Klusch, M.: Service discovery. In: Alhajj, R., Rokne, J. (eds.) Encyclopedia of Social Networks and Mining (ESNAM). Springer (2014)
18. Roy, M., Suleiman, B., Weber, I.: Facilitating Enterprise Service Discovery for Non-technical Business Users. In: Maximilien, E.M., Rossi, G., Yuan, S.-T., Ludwig, H., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6568, pp. 100–110. Springer, Heidelberg (2011)
19. Khorasgani, R.R., Stroulia, E., Zaïane, O.R.: Web service matching for restful web services. In: 13th IEEE International Symposium on Web Systems Evolution (WSE), 115–124. IEEE (2011)
20. Fellbaum, C.: WordNet. Wiley Online Library (1998)
21. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: Iserve: a linked services publishing platform. In: CEUR Workshop Proceedings, vol. 596 (2010)
22. Pautasso, C.: Composing RESTful Services with JOpera. In: Bergel, A., Fabry, J. (eds.) SC 2009. LNCS, vol. 5634, pp. 142–159. Springer, Heidelberg (2009)
23. Krummenacher, R., Norton, B., Marte, A.: Towards Linked Open Services and Processes. In: Berre, A.J., Gómez-Pérez, A., Tutschku, K., Fensel, D. (eds.) FIS 2010. LNCS, vol. 6369, pp. 68–77. Springer, Heidelberg (2010)
24. Alarcon, R., Wilde, E., Bellido, J.: Hypermedia-Driven RESTful Service Composition. In: Maximilien, E.M., Rossi, G., Yuan, S.-T., Ludwig, H., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6568, pp. 111–120. Springer, Heidelberg (2011)
25. Alarcón, R., Wilde, E.: Restler: Crawling restful services. In: Rappa, M., Jones, P., Freire, J., Chakrabarti, S. (eds.) 19th International World Wide Web Conference, pp. 1051–1052. ACM Press, Raleigh (2010)
26. Haupt, F., Fischer, M., Karastoyanova, D., Leymann, F., Vukojevic-Haupt, K.: Service composition for rest. In: IEEE 18th International Enterprise Distributed Object Computing Conference, EDOC 2014, pp. 110–119. IEEE (2014)
27. Stadtmüller, S., Speiser, S., Harth, A., Studer, R.: Data-fu: A language and an interpreter for interaction with read/write linked data. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1225–1236. International World Wide Web Conferences Steering Committee (2013)
28. Sheth, A.P., Gomadam, K., Lathem, J.: Sa-rest: Semantically interoperable and easier-to-use services and mashups. IEEE Internet Computing **11**(6), 91 (2007)
29. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, pp. 195–204. ACM (2013)
30. Bellido, J., Alarcón, R., Pautasso, C.: Control-flow patterns for decentralized restful service composition. ACM Transactions on the Web (TWEB) **8**(1), 5 (2013)
31. Lee, Y.-J.: Semantic-based data mashups using hierarchical clustering and pattern analysis methods. Journal of Information Science and Engineering **30**(5), 1601–1618 (2014)