PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

TOWARDS AUTOMATIC SERVICE COMPOSITION IN REST

RODRIGO ARTURO SAFFIE KATTAN

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
ROSA ALARCÓN

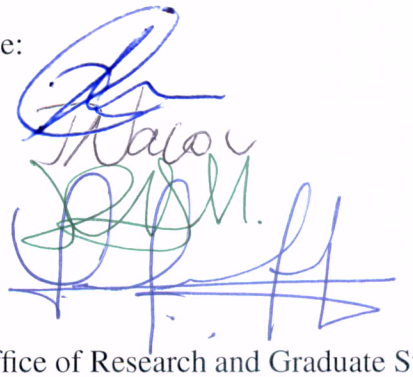
Santiago de Chile, September 2016

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

TOWARDS AUTOMATIC SERVICE COMPOSITION IN REST

RODRIGO ARTURO SAFFIE KATTAN

Members of the Committee:
ROSA ALARCÓN
JAIME NAVÓN
HERNÁN ASTUDILLO
CARLOS BONILLA



Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, September 2016

Gratefully to my parents and siblings

ACKNOWLEDGEMENTS

I would like to thank my Graduate Committee, profs. Jaime Navón, Hernán Astudillo and Carlos Bonilla. I am grateful for their time in reviewing this work. I also thank my colleagues for their companionship and advices, specially Martin Acuña, Patricio Benavente, Nikolas Bravo, Nebil Kawas and Adrián Soto.

I thank my advisor Rosa Alarcón. Rosa's help, patience, kindness and encouragement have been priceless.

I would like to thank my family. Without Eduardo, Fernanda, Felipe, Cristina and Gonzalo, I would not be who I am now.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Representational State Transfer (REST) services and Web Application Programming Interfaces (APIs) have gained considerable attention as service implementation choices, since they favor massive scalability and evolvability. However, most providers of this kind of services describe the rules that must be followed to interact with such services through ad-hoc documentation often written in natural language. Due to the heterogeneity of REST APIs documentation and the lack of a widely accepted standard, automatic REST service composition is difficult to implement. Various approaches for service documentation have been proposed, mainly in the academia; however, due to the industry needs, lightweight Web API documentations have been promoted inside companies. In this thesis we extend and combine various proposals in order to define RAD (REST API description), a lightweight REST API documentation model that allows a graph representation of documented services. We built a case study by documenting real Web APIs with RAD. We populated a graph database from such descriptions and used it to successfully implementing automatic REST service composition.

Keywords: Web API, REST, Service composition, Control flow patterns.

RESUMEN

Los servicios REST y las APIs Web han ganado considerable atención como elección para implementar servicios, dado que favorecen la escalabilidad y evolución. Sin embargo, la mayoría de los proveedores de este tipo de servicios describen las reglas para interactuar con los servicios a través de documentación ad-hoc, generalmente escrita en lenguaje natural. Dada la heterogeneidad de la documentación de las APIs REST, y la falta de un estándar vastamente aceptado por la comunidad, la composición automática de servicios REST es difícil de implementar. Varios intentos de documentación de servicios han sido propuestos, especialmente en la academia; sin embargo, debido a las necesidades de la industria, algunas propuestas de documentación ligera para APIs Web han sido promovidas dentro de las compañías. En esta tesis extendemos y combinamos varias propuestas, para así generar RAD (*REST API Description*), un modelo de documentación ligero para APIs REST, que permite una representación de grafo para los servicios documentados. Construimos un caso de estudio al documentar APIs Web reales con RAD. Poblamos una base de datos de grafo a partir de estas descripciones, y la utilizamos para soportar composición automática de servicios REST.

Palabras Claves: API Web, REST, Composición de servicios, Control de patrones de flujo.

1. INTRODUCTION

Representational State Transfer (REST) (Fielding, 2000) is the architectural style that underlies the Web. It has proved the benefits of its design choices by supporting massive scalability and evolvability, among other advantages. A REST service is a collection of identified resources which are manipulated by a well known set of methods, like the Hypertext Transfer Protocol (HTTP) methods (Fielding et al., 1999). The semantics of these methods are clear to the architectural components (e.g. clients, servers, caches, proxies, etc.).

The previous characteristics facilitate REST services evolvability and scalability in the same way as they do for the Web. Nonetheless, one limitation arises when the consumer of the service is a *machine client* instead of a Web client driven by a human user. The latter merely exposes the resources and leaves the human-user with the responsibility of understanding the underlying semantics at a business level. Since such semantics are presented in natural language, a machine client is not able to understand the business level semantics of a resource. Therefore, it is also impossible to determine the actions a machine client needs to perform in order to accomplish certain business goal. For instance, a *client* must understand that by following certain link or submitting a certain form, a payment is performed. However, following another link could simply add another item to a shopping cart. Both actions differ at the business semantics level, even though at a Web application level they are just HTTP POST methods on different resources.

Interaction between services is desirable since it encourages software reuse and service composition providing aggregated business value at lower cost than building applications from scratch. Automatic service composition will further reduce development cost and time-to-market of composed applications. This purpose has motivated service providers to publish service semantics at business level. Nowadays, a popular approach is the so called Web Application Programming Interface (API) (Richardson et al., 2013), which are a collection of resources and methods typically documented through ad-hoc HTML

pages. These documents describe in more or less detail which methods can be performed on the resources, the required parameters, the restrictions on their values, and the expected results at business level (e.g. examples of the responses). Since the documentation is ad-hoc, machine clients must encapsulate all the Web API logic required to interact with a service. Furthermore, changes on the Web API will break the client or, even worse, lead to underlying inconsistencies. The lack of a machine readable Web API description becomes a severe limitation of automatic REST service composition. Moreover, advanced scenarios (such as dynamic composition at runtime) become very difficult to support.

Various REST service description have been proposed by the academia and recently, some alternatives have been also proposed in the industry (e.g. RAML[1], Swagger[2], recently adopted by the Open API Initiative[3], and Blueprint[4]). Even though they present a step forward towards supporting REST service standardized description, current proposals are operation-centric which hampers hypermedia and hence limiting automatic service discovery, and evolvability. They do not support machine readable, explicit service semantics at business level which makes automatic service composition difficult to implement.

In a previous work (Alarcón et al., 2015), we presented REST API Description (RAD) and demonstrated its benefits to discover the right resource and method required to obtain certain information. This thesis refines and further exploit RAD in order to discover service compositions, that is, workflow fragments that implement more complex business scenarios. We implemented an architecture able to parse RAD service descriptions and generate a graph. The graph is queried to automatically discover service compositions. A REST service composition is a workflow or path of methods performed on resources and chained following certain control-flow patterns. We implemented a test scenario considering 3 popular Web services (i.e. Spotify[5], Songkick[6] and Uber[7]).

[1]RAML: *http://raml.org/*
[2]Swagger: *http://swagger.io/*
[3]Open API Initiative: *https://openapis.org/*
[4]Blueprint: *https://apiblueprint.org/*
[5]Spotify: *https://developer.spotify.com/web-api/*
[6]Songkick: *https://www.songkick.com/developer/*
[7]Uber: *https://developer.uber.com/docs/api-overview/*

1.1. Summary of contributions

Our key contributions in this thesis are fourfold:

(i) We propose a metamodel for describing REST services. This representation allows to capture the semantics of a service, in addition to its logic.

(ii) We propose an algorithm to dynamically and automatically compose REST Web services described with our metamodel. This algorithm is designed to quickly find all possible compositions of services, according to their descriptions.

(iii) We implement a case study using real Web services, adapting their descriptions to our meta-model. This experiment allowed us to validate the capability of RAD for representing the constraints and needs of existing Web services. Also, we validated the capability of the algorithm for finding plausible service composites on runtime.

(iv) We evidenced challenges that must be resolved in order to improve automatic and dynamic composition of REST Web services.

1.2. Organization of this document

The remainder of the thesis is organized as follows: in Chapter 2, we present related work of REST Web services description and composition. In Chapter 3, we introduce the RAD metamodel for describing REST Web services, specifying its main features. In Chapter 4, we present the algorithm to compose RAD-based services. In Chapter 5, we explain the implementation of the dataset, the studied scenarios and the results of our experiments. Finally, we conclude in Chapter 6 with insights of this work and future challenges.

2. RELATED WORK

Traditionally, Web services are described by WSDL (Web Service Description Language), a XML-based document (Chinnici et al., 2007). WSDL describes the service interface (operations, parameters and an endpoint URL) and its conditions to be consumed. Messages interchanged between clients and services must be encoded following the XML-based Simple Object Access Protocol (SOAP) schema (Box et al., 2000).

Traditional Web services are focused on operations, whereas REST services consider resources as its cornerstone. Resources are identified by *resource identifiers* (e.g. Uniform Resource Identifiers); *resource representations* are a set of bytes (e.g. an HTML document) conveying information about the resource state at a particular time.

Another REST foundation is the *Uniform Interface* constraint. It determines that REST components must support the same interface, that resources are manipulated through representations (e.g. retrieve or update a resource state with a new representation), and that REST components (e.g. clients, servers) interact trough self-descriptive messages (e.g. HTTP messages) that include all the information necessary to process such message. Also, the engine of a REST application must follow a *hypermedia constraint*, that is, a resource's representation must embed the necessary controls (e.g. a submit button) and links that inform the client the set of actions available at the current application state.

In traditional Web services, a single endpoint encapsulates an arbitrary number of user-defined operations whose semantics, pre and post-conditions are defined by each service provider. In REST, a service is a collection of identified resources that can be manipulated by a well defined set of methods (e.g. HTTP methods), which facilitates service evolvability by leveraging Web standards (e.g. data formats, network protocols, etc.), and service scalability by exploiting REST architectural constraints (layers, caches, etc.).

2.1. REST service description

In contrast to traditional Web services, REST resources are explicit, methods are finite and representations clearly encapsulate fragments of the business process. The limitation for machine clients is that the semantics of such elements are not defined in a machine processable-way at business level.

Proposals for machine friendly service descriptions that encapsulate service semantics have been made for traditional services. For instance, OWL-S (Martin et al., 2004) is a semantic service description based on ontological models. An ontology is a formal and explicit specification of a shared conceptualization of a certain domain (Gruber, 1993). It typically comprehends a set of concepts, their properties as well as the relationships among them. OWL-S proposes a set of ontologies describing the service domain (e.g. banking), the Web service mechanisms (e.g. operations according to WSDL), the Web service communication protocol (according to SOAP standards) and the control-flow that regulates services compositions (e.g. parallel, alternative, loops, etc.). On one hand, OWL-S is a highly expressive and rich conceptualization of the services domain widely used in semantic service research. On the other hand, OWL-S is highly complex and does not fit the REST service model (i.e. resources, hypermedia or HTTP methods semantics are not supported). A lightweight approach, SAWSDL (Kopecky et al., 2007), is a W3C recommendation for semantic service description that consists of a minimal set of elements that can be used to annotate standard WSDL. The approach is to include references (i.e. URIs) to concepts described on a separate ontology; the standard does not prescribe neither the domain ontology, nor the service semantic model nor the ontology representation language.

For the case of REST services, the Web Application Description Language (WADL) (Hadley, 2006) has been proposed for a description document. This document is equivalent to WSDL for traditional services, and should be annotated with semantic references following the SAWSDL approach. Resulting Web services will be manipulated

through SPARQL queries, and the authors propose a mapping between HTTP methods and SPARQL commands. However the semantics of such actions are not considered at business levels. The semantics of the data were only considered as being manipulated. This proposal also requires the developer to encode on the client all the business logic in order to interact with a service. Similarly to WSDL, WADL has been criticized by its complexity and verbosity (Kopecky et al., 2008; Verborgh et al., 2013; Gregorio, 2007), it resembles WSDL operation-centric approach and ignores REST's hypermedia constraint. WADL representation for hypermedia and links decouples such elements from the representation that contains them making hypermedia a forced and complex paradigm.

Other approaches, such as hRESTs (Kopecky et al., 2008), propose an HTML microformat to annotate the actual HTML pages typically used to describe Web APIs. However, actions's semantics are not considered and the approach can hardly cope with the complexity that current Web APIs present (e.g. optional parameters, optional media type responses, metadata, etc.). Additionally, ReLL (Alarcón & Wilde, 2010) is a REST service description that fully considers REST principles. In an experiment, it was used by a crawler in order to navigate the resources of some REST services, demonstrating the description capability for exploiting the REST hypermedia constraint. ReLL descriptions semantics were also obtained through an additional layer (Alarcón & Wilde, 2010) which was used to obtain the semantic dataset equivalent to the crawled data. ReLL considered one type of action, the GET method, and assumed a single interpretation of such kind of action (i.e. *reading* the resources' state) and the service interface was also simple (i.e. a set of parameters), but it considered the representation's semantics and embedded hypermedia controls. HAL (Kelly, 2015) is a JSON description language that focuses on hypermedia considering only GET methods. Hydra (Lanthaler & Gütl, 2013), goes further by considering resources, operations, and hyperlinks represented as templated links. These templates are a property class that relates certain operation to IRI templates and are mapped (through IriTemplateMapping) to a set of supported variables (URI parameters). IRIs are minted at runtime since parameters values are determined also at runtime. Hydra

is based on JSON-LD (Sporny et al., 2014) that adds lightweight semantics to the description. However, this proposal becomes complex due to the RDF model it is also based on.

A proposal that has gained a lot of traction is Swagger (currently adopted by the Open API Initiative as its core specification). Swagger can be represented as JSON or YAML formats, and it allows to describe resources, operations, and responses. It provides support for specifying operations parameters (optional or required) and responses schemas (including headers) in a simple and intuitive way. The downside is that Swagger does not support semantic associations to its elements nor hypermedia. A similar initiative is RAML, based on YAML, that provides additional support through the provision of a rich data type definitions, as well as URI specification (URI parameters), query parameters specification and various security schemas. RAML is much more expressive but also more complex and less intuitive than Swagger.

2.2. REST service composition

A service composition is typically considered as a combination of service's operations following an specific execution order. If a service does not depend on other services to complete its execution, then it is considered an *atomic* service, otherwise it is *composed* (Dustdar & Schreiner, 2005). Service composition can be *static* if the composition model is defined during the service's design time, or *dynamic* if it is defined at runtime. The composition model is a representation of the set of services to be internally invoked, as well as the data and control-flow that determine its interaction (Dustdar & Schreiner, 2005). The *dynamic* approach facilitates service composition when there are many candidate services available to be part of a composition, reducing development costs and time. It can also facilitate the rapid reaction to failure, business goal changes and personalization. Static or dynamic composition could be defined automatically or manually depending on whether an algorithm chooses service components (automatic service selection) and defines the

control/data flow graph. Finally, if a composed service coordinates the components invocation in a centralized way, it is called an *orchestration*, and *choreography* if each component determines the next participant in the coordination (decentralized).

In REST, *resources* and *resource collections* are the *components* (Pautasso, 2009). Servers embed in representations of resources the set of possible links and controls required to execute a method a resource (i.e. state transition). Clients are responsible for choosing the actual link or control to be executed. This hypermedia approach resembles a choreography where clients and servers cooperate to actually execute certain business process. Naturally, it is possible to hardcode such coordination into a single service implementing this way an orchestration and enforcing a business process completion. However, this approach seriously compromises service scalability (Bellido et al., 2013).

Various strategies have been proposed to implement REST service composition. For instance, Bite (Rosenberg et al., 2008) is a composition language and lightweight framework to create Web-scale workflows based on RESTful services. JOpera (Pautasso, 2009) follows a similar approach but using a visual modeler to specify control and data flow. In (Alarcón et al., 2010), control and data flow is modeled and implemented using a Petri Net, whereas interaction and communication with the resources themselves is mediated by a ReLL service description. JOpera supports dynamic binding of services (Pautasso & Alonso, 2005) whereas the other approaches require static and manual identification of service components. All these approaches follow a centralized orchestration strategy, and assume users consider implicit service semantics to manually design the service composition.

Semantic approaches for machine driven REST service composition rely on semantic Web technologies (e.g. RDF, OWL, N3, etc.) for specifying service semantics and reasoners to implement the composition. For instance, SRSM (Xie et al., 2013) identify a service information (entities) and a transactional layer for resource-oriented service composition. The layers contain specialized resources (i.e. Entity Oriented Resources and Transition Oriented Resources) and services semantics is represented through an OWL

ontology. Automatic service composition depends on OWL-SWRL rules defining the preconditions and effects to be achieved by a search algorithm. RESTdec (Verborgh et al., 2015) proposes the use of N3 language and N3 Logic framework (rules) to create REST service compositions on RDF-based resources. The idea is to define N3 formulas consisting of: a *precondition*, an *HTTP request class*, and a *postcondition*, that are evaluated by an N3-based reasoner called EYE (Verborgh & De Roo, 2015), to create proof-based compositions.

A lightweight approach is proposed in (Bennara et al., 2014), where REST service semantics are modeled as an associated resource. This resource contains both service information and transitions (supported HTTP methods) using JSON-LD. Service composition is achieved by a conversational approach where the client progressively inspects possible actions supported by the retrieved resources. This approach uses lightweight semantic representations, favors dynamic late binding and service evolvability. On the downside, a client cannot know in advance the path of resources and transitions required to achieve a state, and if methods such as POST, PUT or DELETE are required along the way, they cannot be undone. Therefore, the client must know out-of-band (i.e. hard-coded in the client) the required path, at least at a semantic level.

RAD (REST API description) (Alarcón et al., 2015) is another lightweight approach proposing a metamodel to describe Web APIs. It models resources, methods, parameters, responses, links and controls embedded in responses (hypermedia). It is based on a popular Web API description called Swagger, but complements its lack of support for complex parameter management with the corresponding RAML approach. It also allows API developers to annotate resources, parameters and methods with semantic references following the SAWSDL and JSON-LD approach, but keeping the description as minimal as possible. RAD descriptions are translated into a graph that can be queried to discover specific services.

3. RAD: REST API DESCRIPTION

RAD descriptions separate REST service elements into two layers: *semantic* and *activity* (figure 3.1). The semantic layer captures the meaning and purpose of resources, parameters and actions. The activity layer contains elements that are realizations of the semantic layer. A RAD description can be implemented in different formats, from annotated HTML with microdata, to JSON, YAML or XML documents (among others). Compared with our previous work (Alarcón et al., 2015), in this thesis we refine the metamodel by eliminating redundant concepts. For instance, actions and method elements were decoupled in the previous activity layer, whereas in this version they correspond to the **Method** element. The **Representation** element was also refined to explicitly support hypermedia controls that refer to a resource and a method. Cardinality among elements were also revised since the model's elements are much more reusable.

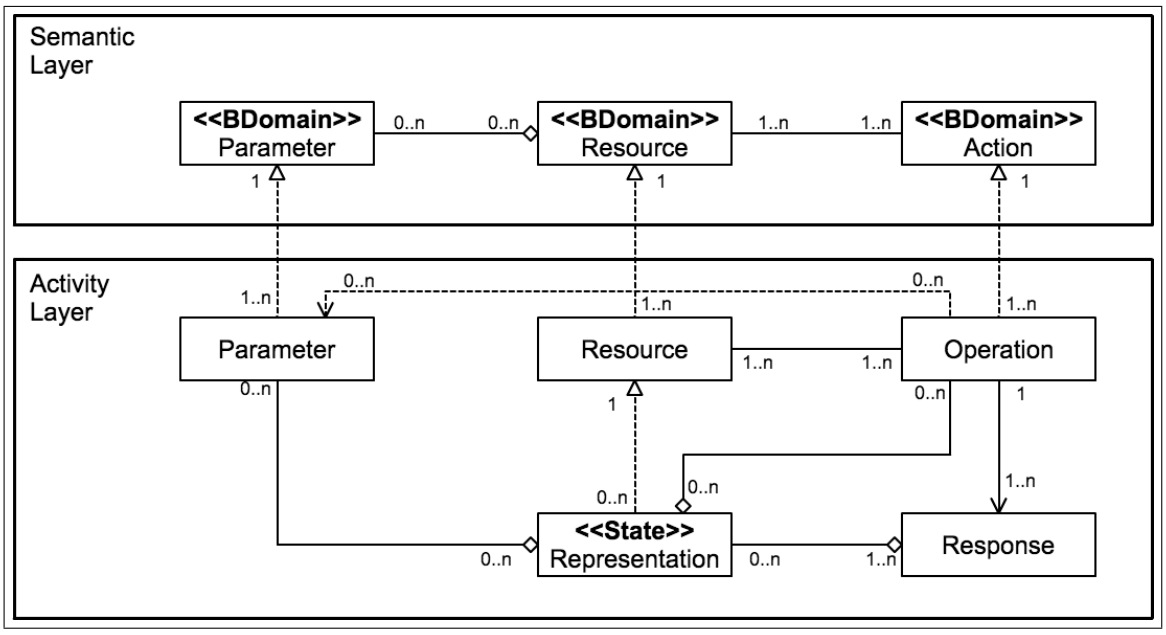


Figure 3.1. RAD metamodel

For the semantic layer, **Resources**, **Parameters** and **Actions** are concepts in the *business domain* that can be semantically related. These concepts represent the semantic aspect of the elements in the activity layer, but they are not bound to any particular knowledge representation formalism. The semantic layer is bound to the activity layer through the description implementation following the SAWSDL approach (i.e. a lightweight reference), without overloading the description. The semantic layer is used to relate different services based on their meaning and purpose. Notice that **Actions** are related to at least one **Resource** and vice versa.

The activity layer is bound to REST Web services. **Resource** elements are identified by their URI and are related to at least one **Method** element (e.g. GET, PUT, POST, DELETE, etc. for the case of the HTTP protocol). In order to execute these operations, the **Operation** element may require **Parameter** elements. Input **Parameters** can be provided in different ways inside a request: as part of the URI itself (URI variables), headers, query or body. **Parameters** can be reused by many operations. After executing an operation, a Web service may return a **Response** element that represents the response message (including the HTTP code). These responses also include the resources' state in the form of a **Representation** element. A **Representation** is comprised of the expected information to be returned by a service and may include output **Parameter** and other **Methods** (hypermedia).

3.1. RAD Concept Vocabulary

As described before, resources, parameters and actions have associated concepts in the business domain that are described in a separate document, a vocabulary. The vocabulary relates the *reference* values to URIs describing concepts unambiguously. In our approach we considered and extended the concepts defined by the Schema.org[1] specification. Schema.org is promoted by Microsoft, Google and Yahoo and can be used as HTML markup that enriches the search results snippets. It comprehends a set of entities

[1]Schema.org: `http://schema.org/`

and extension mechanisms. In this thesis, we used the old mechanism (changed on May 2015), that allows to extend a concept by adding properties following the pattern ”BaseConcept/newProperty”. That is, append the property name, starting with a lowercase, after the concept name followed by a ”/” character. Similarly, a class can be refined by an specialized concept name, starting with uppercase: ”BaseConcept/SpecializedConcept”. Camel case is requested.

```
{
  "name": "RAD-Schema.org",
  "version": "1.0",
  "description": "Extension and adaptation of Schema.org's dictionary for RAD.",
  "created_at": "3/9/2015",
  "updated_at": "03/02/2016",
  "baseUri": "http://schema.org",
  "prefixes": {
    "resources":{
      "@Place": {
        "reference": "/Place",
        "parameters": {
          "@placeLatitude": "/latitude",
          "@placeLongitude": "/longitude",
          "@placeGeo": "/geo",
          "@placeIpv4": "/ipv4",
          "@placeIdentifier": "/identifier",
          "@placeName": "/name",
          "@placeAdress": "/adress",
          "@placeStreet": "/street",
          "@placeZip": "/zip"
        }
      }, …
    },
    "actions": {
      "@AchieveAction": "/AchieveAction",
      "@AddAction": "/AddAction",
      "@AssessAction": "/AssessAction",
      "@CheckAction": "/CheckAction",
```

Figure 3.2. Associated *Schema.org* based vocabulary snippet

The vocabulary was designed as a JSON document (Figure 3.2). The required keys are *name*, *version*, *baseUri* and *prefixes*. Prefixes are abbreviations of conceptual entities such as resources, parameters and actions. They must start with a ’@’ symbol and are related to explicit URIs through the *reference* key. Each entity property can be defined trough a *parameter* key, which value is the URI fragment that must be appended to the base concept, as defined by Schema.org extension mechanism.

3.2. RAD as a JSON document

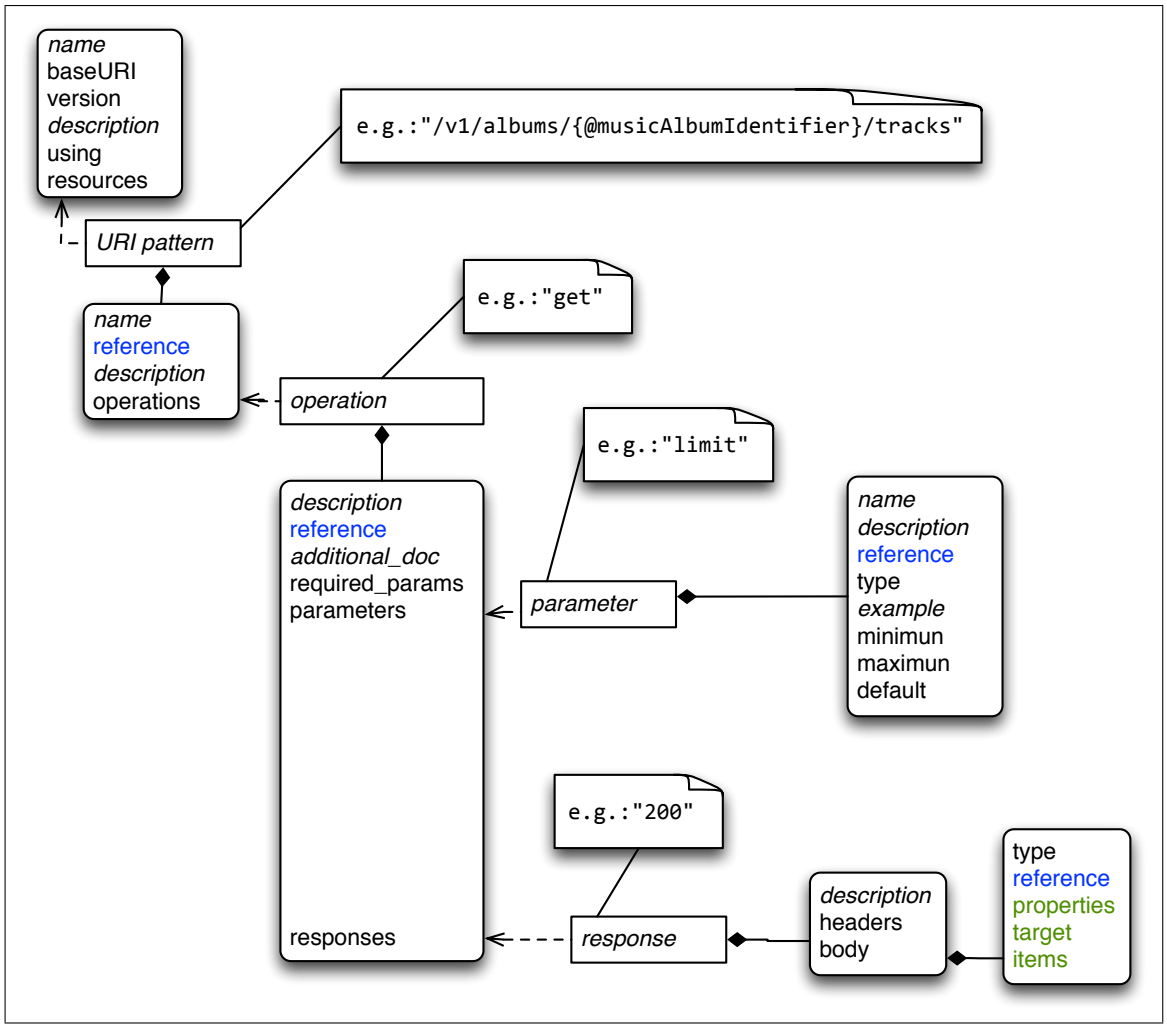


Figure 3.3. JSON implementation schema of RAD

In this section we present an implementation of the RAD metamodel as a JSON document (see figure 3.3 for an overview of the corresponding JSON schema). A RAD JSON description serves two purposes. On one hand it serves as the basis for a documentation. On the other hand it serves as a machine-readable description for machine-clients. In figure 3.3, the keys that are destined to document the service are presented in italics in the figure (*name*, *description*, *additional_doc*, and *example*) and they are considered as optional keys. Also, the semantic relation between *Parameters*, *Resources* and *Methods*

are modeled as *reference* keys (in blue in the figure) and specialized modifiers for output parameters are presented in green.

A RAD JSON description must have *baseURI*, *version*, *using* and *resources* keys. The *baseURI* key refers to the root of the service's entry-points (Webber et al., 2010), the *version* key represents an identifier for modifications in the description, the *using* key specifies the referenced semantic vocabulary and the *resources* key describes each resource of the Web service. A resource object is identified by its URI template, used as the key of the object. It is possible to include semantic references to URI variables using the "@" modifier as seen in figure 3.3 (`/v1/albums/{@musicAlbumIdentifier}`), where *@musicAlbumIdentifier* indicates that the variable part of the URI is associated to a concept, defined in the semantic vocabulary referenced in the description.

```
{
  "name": "Spotify",
  "baseURI": "https://api.spotify.com",
  "version": "v1",
  "description": "Our Web API endpoints give external applications access to Spotify catalog and user data.",
  "using": "rad-schema-1.0.json",
  "resources": {
    "/v1/albums": {
      "name": "Collection of Albums",
      "reference": "@MusicAlbumCollection",
      "description": "Spotify's albums.",
      "operations": {
        …
      }
    },
    "/v1/albums/{@musicAlbumIdentifier}": {
      …
    }
```

Figure 3.4. Spotify Web API described as RAD JSON: *resources* key snippet

A *resource* object (see figure 3.4 for an example) must have a semantic *reference* and related *method* keys. A *reference* key associates the resource with a concept in the vocabulary. The *method* key refers to the network protocol method available for the resource (e.g. GET, POST, etc. for the case of HTTP).

Method keys are *reference*, *required_params*, *parameters* and *responses* (figure 3.5). The *reference* key associates the method to the corresponding semantic element as described in the vocabulary. The *required_params* key comprehends boolean expressions to be evaluated for the parameters (URI variables are always required). The *parameters*

```
"operations": {
  "get": {
    …
  },
  "post": {
    "description": "Add one or more tracks to a user's playlist.",
    "reference": "@AddAction",
    "additional_doc": "https://developer.spotify.com/web-api/add-tracks-to-playlist/",
    "required_params": "!Authorization AND ((#uris AND !Content-Type) OR NOT #uris)",
    "parameters": {
      "!Authorization": {
        "name": "Authorization Access Token",
        "description": "A valid access token from the Spotify Accounts service.",
        "reference": "@webApplicationApiKey",
        "type": "string"
      },
      "!Content-Type": {
        "name": "Content Type",
        "description": "Required if the IDs are passed in the request body.",
        "reference": "@webApplicationContentType",
        "example": "application/json",
        "type": "string"
      },
      "uris": {
        "name": "Uris",
        "description": "A comma-separated list of Spotify track URIs to add.",
        "reference": "@musicRecordingCollectionUri",
        "type": "string",
        "example": "spotify:track:4iV5W9uYEdYUVa79Axb7Rh"
      },
      "position": {
        "name": "Playlist ID",
        "description": "The position to insert the tracks, a zero-based index.",
        "reference": "@musicPlaylistPosition",
        "type": "integer",
        "example": "2"
      },
      "#uris": {
        "name": "Uris",
        "description": "A JSON array of the Spotify track URIs to add.",
        "reference": "@musicRecordingCollectionUri",
        "type": "array",
        "example": [
          "spotify:track:4iV5W9uYEdYUVa79Axb7Rh",
          "spotify:track:1301WleyT98MSxVHPZCA6M"
        ]
      }
    },
    "responses": {
```


Figure 3.5. Spotify Web API described as RAD JSON: *operation* and *parameter* keys snippet

object is a set of descriptions of the required and optional input parameters required to perform the method. Parameters can be part of the URI, the body (requires a '#' prefix) or the header (requires a '!' prefix). The *responses* object is the set of possible response message whose semantics are relevant for the API. Optional attributes of a *method* object

are *description* and *additional_doc* for documentation purposes. The *additional_doc* value represents a link where to find more information about the method.

The required keys for a *parameter* object are *reference* and *type*. Again, the *reference* key associates the parameter with a semantic concept in the vocabulary. The *type* represents the programmatic type of the variable, possible values are *string*, *integer*, *boolean* and *array*. Optional attributes are *name*, *description* and *example* for documentation purposes. Parameter values can be also restricted by the following keys: *enum* (indicating a restricted set of possible values), *default* (indicating a default value), *minimum* indicating a minimum value and *maximum* value for integers.

```
"responses": {
  "200": {
    "description": "On success, the HTTP status code in the response header is 200 OK.",
    "headers": [],
    "body": {
      "type": "object",
      "reference": "@MusicAlbum",
      "properties": {
        "album_type": {
          "type": "string",
          "reference": "@musicAlbumType"
        },
        "artists": {
          "type": "array",
          "reference": "@MusicGroupCollection",
          "items": {
            "type": "object",
            "reference": "@MusicGroup",
            "properties": {
              "href": {
                "type": "hyperlink",
                "target": "/v1/artists/{@musicGroupIdentifier}"
              },
              "id": {
                "type": "string",
                "reference": "@musicGroupIdentifier"
              },
              "name": {
                "type": "string",
                "reference": "@musicGroupName"
              },
```

Figure 3.6. Spotify Web API described as RAD JSON: *response* key snippet

A *response* object is identified, for the case of HTTP, by the response code. The required keys in this case are *headers* and *body* (figure 3.6). The *headers* key is an array containing the relevant message headers expected in the response (e.g. HTTP headers).

The *body* value describes our expectation regarding the response. The *body* has three mandatory keys associated: *reference*, that relates a resource representation to a concept in the vocabulary; *media* that specifies the response media type; and *type* that determines the data type of the information contained in the body. In the current version we only support the *application/json* media type. Accepted values for *type* are those defined by JSON Schema[2] (i.e. *string*, *integer*, *number*, *object*, *array*, *boolean*, *null*) and *hyperlink*. The *hyperlink* value requires an additional *target* key to indicate the URI of a referenced resource in the response (hypermedia controls).

3.3. RAD as a graph

We chose to use a graph database for storing RAD descriptions because the relationships between the RAD elements form a graph of arbitrary topology. Figure 3.7 presents the graph model that represents a RAD description and mimics the RAD metamodel shown in figure 3.1.

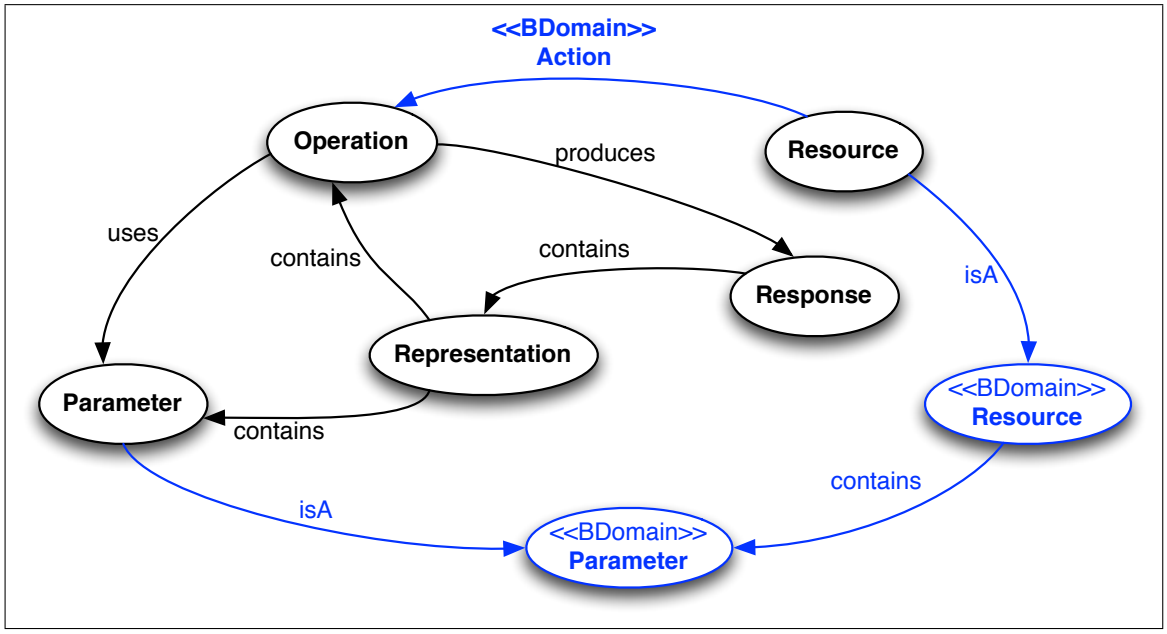


Figure 3.7. RAD graph model

[2]JSON Schema: `http://json-schema.org/`

4. RAD-BASED SERVICES COMPOSITION

We consider a REST composition as a workflow (or a path in the RAD graph) comprehending a set of methods that allow a client to reach a *goal* (or final state). Such goal comprises the set of output parameters produced when executing the path's methods. The methods are executed following certain control-flow patterns. In this thesis, we identify only three control-flow patterns, namely *sequence*, *alternative* and *parallel split - synchronization* (parallel for short) (Bellido et al., 2013).

A sequence pattern defines the consecutive invocation of methods, without any guard condition associated. A sequence can be inferred from the methods' dependency of input and output parameters as exploited in (Vairetti et al., 2016).

An alternative pattern allows the execution of only one of two possible services depending, in our case, on the user decision. Our approach consists on creating a workflow plan that include all possible choices (methods) but let the user to choose. Again, an alternative pattern can be inferred from the method signature (i.e. if two methods produce the same output but differ in their input, they may indicate an alternative pattern).

A parallel split allows a single thread of execution to be split in two or more branches. The synchronization pattern requires that the execution thread is halted until it receives all the results from previous methods executed in parallel. Two or more methods that do not depend among them and have access to all of their input parameters are considered as following the parallel-split pattern.

Automatic RAD services composition is implemented by a backtracking algorithm (see figures 4.1 and 4.2). In order to manage data heterogeneity, we consider the semantic equivalent of input and output parameters (i.e. parameter concept), instead of each particular parameter. The algorithm *Input* is a client's *composition request* specifying the

required goal: the set of concepts in the vocabulary associated to output parameters (*output_concepts*). The client may optionally indicate a set of concepts associated to input parameters (*input_concepts*), that represent information that the client has.

Parallel and alternative paths can be reduced to a sequence model in order to calculate a critical execution path (Bellido et al., 2013). We consider such critical sequence as *steps* (a path is a sequence of steps). A step is a set of *operations* that do not depend among them, and their output parameters are required by the following step in a path (or this set of operations satisfies the client's *goal*). Operations in a *final step* must produce all the output concepts required. Operations in a *first step* must be executable with the *input concepts* supplied by the client (if needed). Along a path, each operation must be executed at most once.

Input: $input\_concepts$, $output\_concepts$

Output: $solution\_paths$

1: $goal\_operations$ $\leftarrow$ All operations that return at least one concept of $output\_concepts$
2: $final\_steps$ $\leftarrow$ Steps containing combinations of $goal\_operations$, that executed in parallel return all $output\_concepts$ (excluding redundant operations for each concept)
3: $banned\_steps$ $\leftarrow$ $final\_steps$ (or super-sets of each one) banned from being used inside a path
4: **for** $step$ in $final\_steps$ **do**
5: $step.required\_param\_concepts$ $\leftarrow$ All the combinations of parameters concepts required to execute all the operations of $step$
6: $path$ $\leftarrow$ Create path from $step$
7: **for** $combination$ in $step.required\_param\_concepts$ **do**
8: **if** $combination$ is subset of $input\_concepts$ **then**
9: Add $path$ to $solution\_paths$
10: break
11: **end if**
12: **end for**
13: **if** $path$ not in $solution\_paths$ **then**
14: Add $path$ to $candidates$
15: **end if**
16: **end for**

Figure 4.1. Pseudo-code of composition algorithm: part 1

```
17: while candidates do
18:   for path in candidates do
19:     Remove path from candidates
20:     for combination in path.first_step.required_param_concepts do
21:       previous_operations ← All operations that return a parameter concept from
      combination, not available in input_concepts
22:     end for
23:     previous_steps ← Steps containing combinations of previous_operations, that
      executed in parallel return all parameter concepts in combination (excluding redun-
      dant operations for each concept and banned operations of the path), and different to
      banned_steps
24:     for step in previous_steps do
25:       step.required_param_concepts ← All the combinations of parameters con-
      cepts required to execute all the operations of step
26:       new_path ← Create new path from path and prepend step to it
27:       for combination in step.required_param_concepts do
28:         if combination is subset of input_concepts then
29:           Add new_path to solution_paths
30:           break
31:         end if
32:       end for
33:       if new_path not in solution_paths then
34:         Add new_path to candidates
35:       end if
36:     end for
37:   end for
38: end while
return solution_paths
```

Figure 4.2. Pseudo-code of composition algorithm: part 2

The algorithm starts by identifying the operations (*goal_operations*) that produce at least one *output_concept* (line 1). Then, the set of possible *final_steps* is created from *goal_operations* (line 2). A *final_step* is the combination of operations (or a single operation) that produces all the output concepts. An operation is included in a step if it returns a parameter concept expected from the step, given that no other operation of that step returns the same concept (no redundancy). Then, for each operation in each *final_step*, the algorithm finds all the combinations of required input concepts needed to execute the step (line 5).

It may be the case that a particular combination includes additional parameters than those provided by the client. In that case, the algorithm determines that such step requires further analysis and marks the step as a *candidate* (line 14). *Candidates* are paths containing operations that can produce all the output concepts, but require more input concepts than those defined by the client. *Candidates* are explored in a backward chaining way. Since the backward chaining process can consume a high number of resources and time, process limits may be required (e.g. timeout, max. number of steps, max. number of operations, max. number of solutions, etc.). If all the input concepts of a step are known, each operation of the step can be fully executed, so the path that the step belongs to is considered a valid solution path (line 9).

Then, all *candidates* are evaluated (line 17). In order to evaluate a *candidate*, the algorithm identifies the *first_step* in the *candidate*'s path (the step further from the final step, i.e. the newer step), and the combination of all input parameters for the step's operations (line 20 to 22). Then, the algorithm finds possible *previous_steps* (line 23) that return the input concepts required by the *first_step*. For instance, lets suppose that an operation in the *first_step* requires input parameter concepts A, B and C. The algorithm finds as previous operation *Op1*(returns $\{A\}$), *Op2*(returns $\{B,C\}$) and *Op3*(returns $\{A,C\}$). Hence, the *previous_steps* would be *Step1*(*Op1* and *Op2*) and *Step2*(*Op2* and *Op3*). Note that *Op1* and *Op3* would not be part of a same step due to the redundancy of the concept A. However, *Op2* and *Op3* are part of *Step2* even if there is redundancy of concept C because

each operation provides at least a third required concept. As said before, operations inside a step (e.g. *Op1* and *Op2*) can be executed in parallel (they do not depend among them). Each step of a path is executed before the following steps (sequence pattern).

In order to determine if a *candidate* is a solution, the algorithm evaluates whether the operations in a *previous_step* have available all the required input concepts (line 28). In such case, the *previous_step* is added as *first_step* to the *candidate*, which is now a solution (line 29). If the *previous_step* can not be executed, the new path is considered as a *candidate*, and the *candidates* evaluation continues. The evaluation of *candidates* stops when there are no more *candidates*, or the whole graph has been traversed so there are no more operations that can be added to a path.

The complexity of the algorithm depends mainly on the number of operations that return at least one *output_concept*. Also, it is influenced by the number of *required parameters concepts* in each *final_step*. More combinations of *required parameters concepts* could produce more *candidates* to evaluate, because there could be more alternatives to reach each *final_step*. Additionally, the number of alternatives between an executable step only with *input_concepts* and a *final_step* increases the number of *candidates*, and hence the complexity.

5. IMPLEMENTATION AND EVALUATION

In order to create a proof-of-concept prototype, we searched for real Web APIs. These services have to adhere to REST constraints, provide comprehensive documentation and fall into a related domain. Even though some REST constraints are violated, we selected the following three Web services: Spotify, Songkick and Uber. The Spotify API provides access to its music streaming service's catalogue. The Songkick API grant access to a live music database with information about upcoming and past concerts, as well as setlists. The Uber API allows a client to ask for types of transportation services, estimate price and arrival time of a ride, as well as a user's profile and activity information.

We manually created RAD JSON descriptions for each Web API, as well as a vocabulary based on Schema.org. We had to extend the vocabulary to model the concepts considered in the Web APIs. We implemented our approach by refining the RAD parser presented in (Alarcón et al., 2015). The parser, written in Python, transforms RAD JSON descriptions and vocabulary files into a RAD graph. JSON files are validated by a JSON Schema template before being parsed. We chose the popular database Neo4j[1] to store the dataset because it provides a native graph model. Also, with the help of Py2Neo[2] library, both loading and interacting with data is effortless. The composition algorithm was implemented also in Python and use the Py2Neo library to access the database.

5.1. Characteristics of the dataset

Table 5.1 presents a summary of the generated nodes and edges in the graph. Tables 5.2 and 5.3 present further detail of the generated nodes. Tables 5.4 and 5.5 presents the number of shared concepts among Web APIs in terms of resources (Table 5.4) and parameters (Table 5.5). Songkick and Spotify share the major number of concepts, as expected since they address the same business domain.

[1]Neo4j: *http://neo4j.com/*
[2]Py2Neo: *http://py2neo.org/2.0/*

Table 5.1. Nodes and edges in the graph database

	Nodes	**Edges**
Vocabulary	173	130
Spotify	611	1189
Uber	175	304
Songkick	318	579
Total	1277	2202

Table 5.2. Activity layer nodes

API	**Resources**	**Operations**	**Parameters**	**Responses**	**Representations**
Spotify	20	27	513	28	23
Uber	10	11	129	15	10
Songkick	15	15	260	15	13

Table 5.3. Semantic layer nodes

API	**Resource Concepts**	**Actions**	**Parameter Concepts**
Spotify	10	9	63
Uber	7	4	46
Songkick	6	3	39

Table 5.4. Shared resource concepts

	Spotify	**Uber**	**Songkick**
Spotify		1	3
Uber	1		0
Songkick	3	0	

Table 5.5. Shared parameter concepts

	Spotify	**Uber**	**Songkick**
Spotify		8	10
Uber	8		8
Songkick	10	8	

5.2. Scenarios of evaluation

In order to test our approach, we defined three evaluation scenarios that differ on the composition *goal* and the expected number of involved APIs (*scope*). The scenarios are:

- Scenario 1:
  - *Scope: 2 Web APIs, Goal: Obtaining the* name *of a music group.*

  In this case, it is expected that only the two APIs related to music (Spotify and Songkick) are involved in the solution. The expected output parameter must correspond to the concept: `http://schema.org/MusicGroup/name/`.

- Scenario 2:
  - *Scope: 2 Web APIs, Goal: Obtaining the* name *and* popularity *of a music group.*

  We modify the previous scenario and ask for an additional *output concept* related to the popularity of a music group (`http://schema.org/MusicGroup/popularity/`). Again, we expect that only Songkick and Spotify APIs are part of the solution.

- Scenario 3:
  - *Scope: 3 Web API, Goal: Obtaining an* estimated fare *for a taxi ride to a music group's concert.*

  In this case, we expect that the three APIs, Spotify, Songkick and Uber are involved in the solution. The corresponding concept for the goal is `http://schema.org/Estimate/value/`. This concept corresponds to an estimation of a product's price. In the dataset, the only service that returns this concept is Uber.

5.3. Input configuration

For each scenario, it is mandatory an access token provided by the Web service to identify the client. We considered 3 cases for *input concepts*:

- 1 parameter: No additional parameter is provided other than the API key (i.e. `http://schema.org/WebApplication/apiKey/`).
- 2 parameters: A concept closely related to the scenarios goal is provided. We considered the concept of a music group identifier since it should be used in some solutions (`http://schema.org/MusicGroup/identifier/`).
- 3 parameters: The client does not know the *input concepts*, but knows how to find them through the APIs search capability. Parameters related to the search concept (`http://schema.org/Search/type/` and `http://schema.org/Search/query/`) are considered as *input concepts*, in this case. We choose this configuration since it is a very common scenario.

5.4. Results

We ran our experiments on an Intel Xeon processor with Turbo up to 3.3GHz, 1 vCPU and 1 GB of RAM, running on Ubuntu 14.04. We performed the tests 10 times and we averaged the execution time in order to obtain a reliable measure. We present the results of the algorithm to find solutions with increasing steps, in three scenarios. We also present the average search time for each scenario, as well as the number of valid paths (i.e. those that produce the requested goal). For long running executions we defined a limit of 7 steps to stop the algorithm. Remaining candidates represent paths that could be solutions for a composition request, but have not yet reached an executable first step when the algorithm was halted.

5.4.1. Scenario 1

Table 5.6 presents the results for the evaluation of scenario 1. As the search algorithm incorporate more steps, the number of solutions increases quickly. The solutions' growth is due to the presence of alternative steps to obtain the required *output concept*. The number of solutions found involve 4 steps at most (there are no more candidates pending to examine). As shown in Table 5.7, the algorithm is able to find all the answers in less than 2 seconds, independently of the number of *input concepts*. The solutions for 1 and 2 steps for 1 *input concept* are presented in Figure 5.1. As in Figures 5.2 and 5.3, black nodes correspond to the Spotify API, red nodes to the Songkick API and green nodes to the Uber API. Concepts are presented in blue. Arrows represent GET operations, and they follow a sequence pattern. There are 10 alternative solutions considering 2 steps. Unexpectedly, Uber API resources are also part of a viable solution for the 2 steps set.

Table 5.6. Solutions by number of steps for scenario 1

Input Parameter Concepts	**Steps**			
	1	**2**	**3**	**4**
http://schema.org/WebApplication/apiKey/	1	10	18	18
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	6	15	18	18
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	3	12	19	18

Table 5.7. Summary of solutions for scenario 1

Input Parameter Concepts	**Total Solutions**	**Candidates**	**Execution Time (seconds)**
http://schema.org/WebApplication/apiKey/	47	0	1.3590 ± 0.1423
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	57	0	1.3154 ± 0.0232
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	52	0	1.3712 ± 0.1206

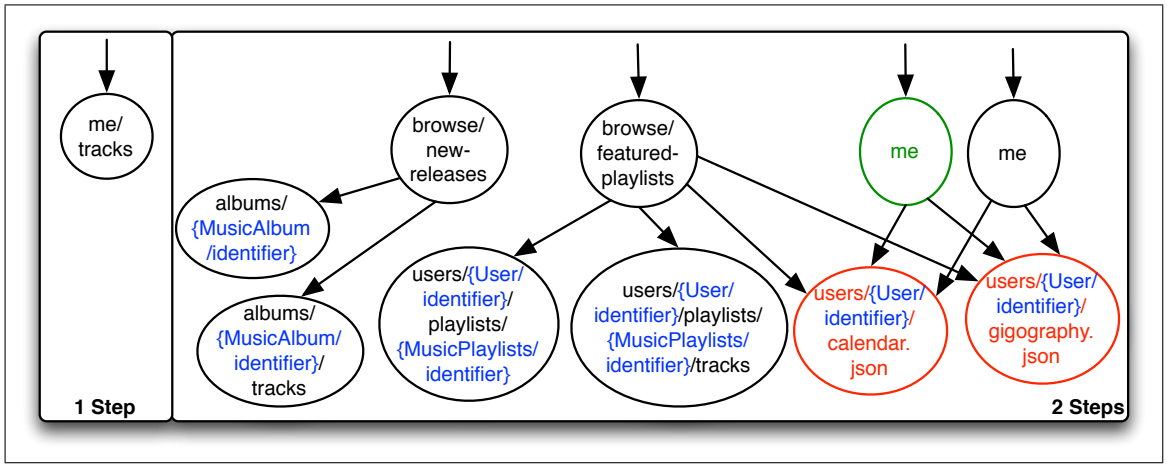


Figure 5.1. Compositions found for scenario 1, with 1 and 2 steps for 1 *input concept*

5.4.2. Scenario 2

In Table 5.8, the number of solutions increases for 1 parameter compared to scenario 1 since the *input concepts* for the target resources were not provided. Through backtracking, the algorithm identifies the operations that produce the required input, but they require additional *input concepts*, creating more candidates with more steps. In this scenario, the limit of 7 steps is reached, remaining various candidate compositions to solve.

Independently of the *input concepts*, the algorithm correctly finds the only two operations that return the popularity of a music group and the music group's name as can be seen in Figure 5.2. When we compare Figures 5.2.a and 5.2.b, we can identify a significant difference. In the later case, the provided concept already refers to a music group identifier, so that, finding the resource and operation is straightforward.

For scenarios 1 and 2, the paths created by the algorithm allow to answer these business cases (among others):

- Name (and popularity) of a user's favorite tracks music groups.
- Name (and popularity) of music groups in new releases or featured playlists.
- Name (and popularity) of a user's event calendar music groups.

Table 5.8. Solutions by number of steps for scenario 2

Input Parameter Concepts	Steps						
	1	**2**	**3**	**4**	**5**	**6**	**7**
http://schema.org/WebApplication/apiKey/	0	2	36	326	2.834	22.398	148.190
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	2	0	0	0	0	0	0
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	1	4	54	516	4.682	37.192	241.774

Table 5.9. Summary of solutions for scenario 2

Input Parameter Concepts	Total Solutions	Remaining Candidates	Execution Time (seconds)
http://schema.org/WebApplication/apiKey/	173.786	954.996	103.2259 ± 1.0574
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	2	0	0.0733 ± 0.0012
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	284.223	1.024.054	89.6786 ± 1.5204

- Name (and popularity) of a user's playlist music groups.
- Name (and popularity) of an album's music groups.
- Name (and popularity) of music groups similar to one another.

5.4.3. Scenario 3

Table 5.10 presents the results for the evaluation of scenario 3. As in previous cases, we vary the number of *input concepts*. Table 5.11 presents a summary of the execution. We can observe a large number of solutions in all cases. This behavior is due to the existence of only one operation returning an estimate of a ride, but none of its required parameters are supplied as part of the initial input. Hence, the algorithm must find many alternatives to supply such *input concepts*. Hence, there are no solutions of one step (see Table 5.10): the shortest path involves at least two steps.

Figure 5.3 presents the compositions found with 2 and 3 steps. For example, a solution for 2 *input concepts* is highlighted with dotted lines. It begins requesting artists similar to a

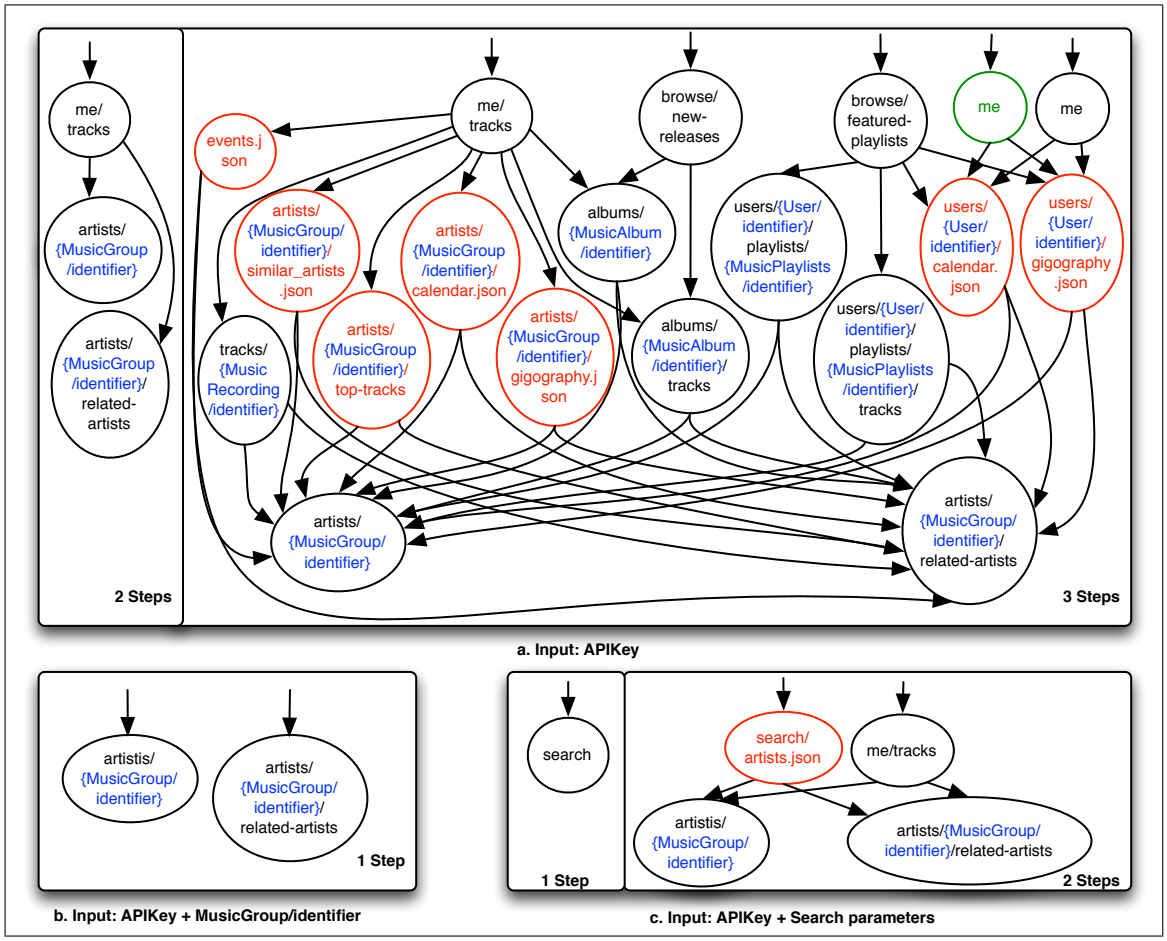


Figure 5.2. Compositions found for scenario 2, with 1, 2 and 3 steps

Table 5.10. Solutions by number of steps for scenario 3

Input Parameter Concepts	**Steps**						
	1	**2**	**3**	**4**	**5**	**6**	**7**
http://schema.org/WebApplication/apiKey/	0	1	11	132	1.726	25.539	336.803
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	0	3	20	178	2.016	18.083	103.672
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	0	3	22	350	5.525	78.364	1.010.114

particular one (GET on the resource `api.spotify.com/v1/artists/{http://schema.org/MusicGroup/identifier/}/related-artists`), followed by requesting the planned concerts for such artists (GET on resource `api.songkick.com/`

Table 5.11. Summary of solutions for scenario 3

Input Parameter Concepts	Total Solutions	Remaining Candidates	Execution Time (seconds)
http://schema.org/WebApplication/apiKey/	364.212	4.663.031	394.1312 ± 11.9393
http://schema.org/WebApplication/apiKey/ *http://schema.org/MusicGroup/identifier/*	123.972	343.197	78.4665 ± 0.9224
http://schema.org/WebApplication/apiKey/ *http://schema.org/Search/query/* *http://schema.org/Search/type/*	1.094.378	5.322.899	288.9120 ± 2.6779

`api/3.0/events.json`), and then requesting an estimated taxi fare for a chosen event (GET on resource `api.uber.com/v1/estimates/price`).

Independently of the input parameters, the algorithm finds the resource that provides the expected output (GET on the Uber resource: *estimates/price*). Only for the input including the music group name concept, a POST operation (blue arrow) is considered in the 3 steps solution set. Again the only patterns identified are sequence and alternative. Parallel patterns were found in solution sets with more steps (not shown). The compositions found in this scenario represent the following situations:

- Cost of going to a venue
- Cost of going to an artist's future events
- Cost of going to a user's upcoming events
- Cost of going to concerts of similar artists to another that a user likes

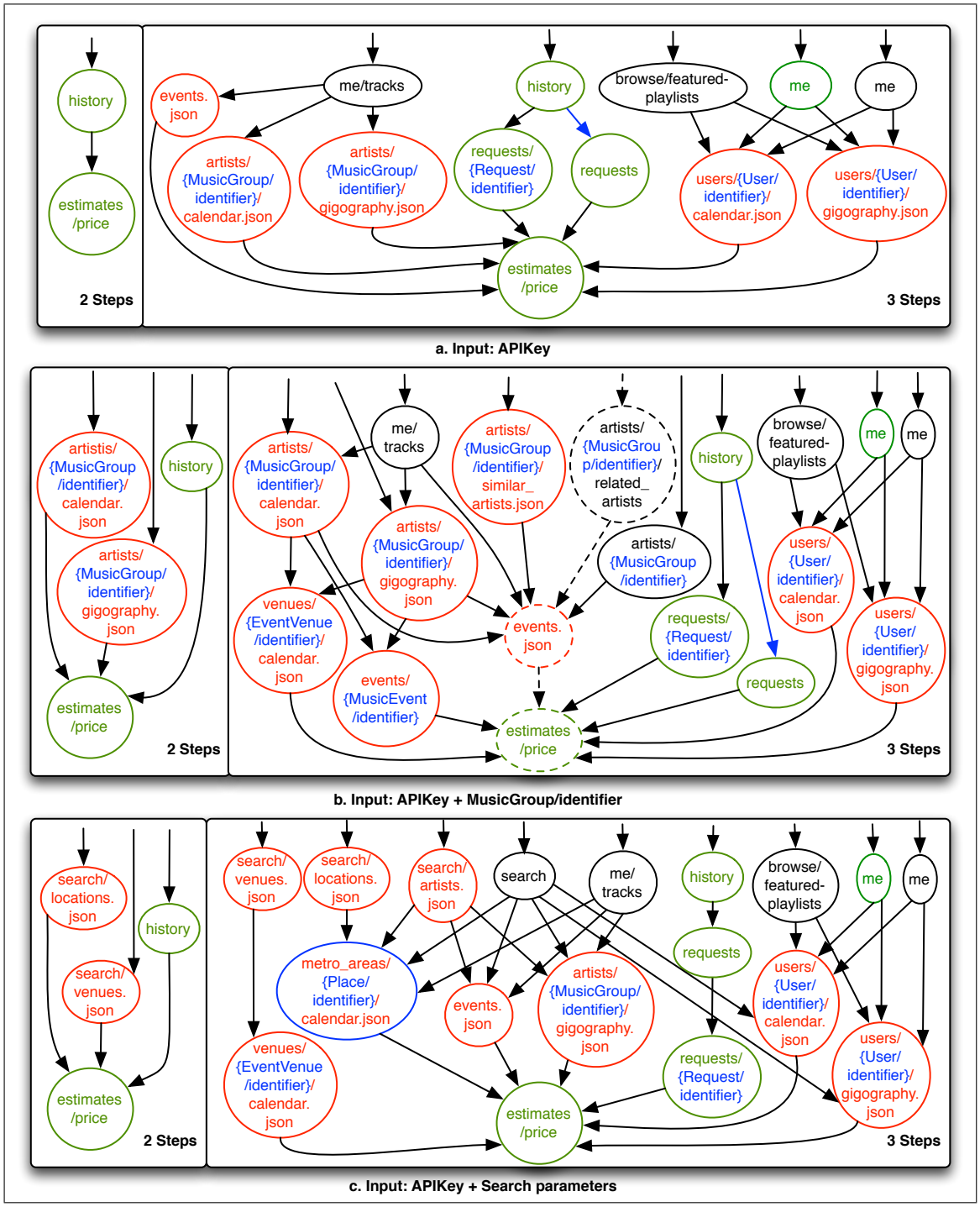


Figure 5.3. Compositions found for scenario 3, with 2 and 3 steps

6. CONCLUSIONS

One of the contributions of this thesis is the RAD description metamodel for REST services and its implementation using JSON. This proposal results in a lightweight approach that is capable of modeling well-know, industry level Web APIs. We reviewed various Web APIs in order to find those services closest to the REST style and test our approach. Our implementation was able to support most of the common practices followed by Web APIs when dealing with their input and output parameters in a lightweight style. That is, some parameters are optional, some are mandatory, some are present in the header, others in the body, others in the URI scheme, and some require certain data types as seen in figures 3.5 and 3.6. One property we did not support, however, was the dependency between the response and particular values of the input parameters. In some services, the response's structure may vary depending on the values of the input parameters.

A second contribution is the lightweight metagraph based on the proposed metamodel. The metagraph made possible not only to discover specific services (Alarcón et al., 2015) but also to support a complex task such as automatic and dynamic service composition with reasonable performance. The metamodel can be implemented in various ways, for instance as an RDF ontology, or as any other graph-based approach.

One advantage of our lightweight approach for the semantic layer is that it can be integrated with existing Web services descriptions, regardless of their format (e.g. HTML, JSON, YAML, XML, etc.), without interfering with the exposed information. It can also refer to specific semantic models such as existing ontologies and Linked Data[1]. This layer makes possible to bind different services, based on the meaning and purpose of their exchanged information and hence make possible a rich service composition as can be seen in the results in figures 5.1, 5.2 and 5.3.

However, one of the disadvantages of our proposal is that the description is separated from the service itself, as an additional layer. This factor increases coupling between

[1]Linked Data: *http://linkeddata.org/*

services and their descriptions and limits service evolvability. A way to lessen this disadvantage is to implement clients that consider service descriptions as information models of what they can expect, instead of guaranteed contracts during execution time.

Also, when a large number of compositions are generated, a ranking strategy is required for a client to effectively use the proposed compositions. For instance, solutions with fewer steps may be preferred since they may be executed faster. However, a proper quality model representing client's interest (such as the cost of a service) is required.

In addition, more complex control flow patterns could be supported by the algorithm, which would yield in even more solutions. Also, the removal of some restrictions in the composition algorithm could lead to more interesting solutions.

Our proposal is based on services signature such as input and output parameter concepts, however, other elements to consider could be the semantics of the actions, the status codes of responses and the response metadata. This extension may result in solutions closest to the client's goal. For instance, a client's goal to change a resource's state may be satisfied by a solution that do not return the changed state itself, but a HTTP code.

Composition results would improve with a better definition and use of concepts. Some concepts could be too general for some cases, so it would require the creation of more specific concepts. These new concepts should be related with the generic one, and their bound should be taken into account while composing services. Also, some concepts were assumed to be equivalent (e.g. identifiers and tokens). Generally, however, this is far from trivial in real applications. A further refinement of the concept hierarchy may be required leading to less solutions for a composition request, but with more business value.

As for future work, we will focus on supporting the implementation and execution of the compositions themselves. Again, this is far from trivial since it will require to face differences in variable's types; even though they are semantically equivalent, data types may be drastically different.

REFERENCES

Alarcón, R., Saffie, R., Bravo, N., & Cabello, J. (2015). REST Web Service Description for Graph-Based Service Discovery. In *Engineering the Web in the Big Data Era* (pp. 461–478). Springer.

Alarcón, R., & Wilde, E. (2010). Linking data from restful services. In *Third Workshop on Linked Data on the Web, Raleigh, North Carolina (April 2010).*

Alarcón, R., & Wilde, E. (2010). RESTler: crawling RESTful services. In *Proceedings of the 19th international conference on World wide web* (pp. 1051–1052). ACM.

Alarcón, R., Wilde, E., & Bellido, J. (2010). Hypermedia-driven RESTful service composition. In *Service-Oriented Computing* (pp. 111–120). Berlin, Heidelberg: Springer.

Bellido, J., Alarcón, R., & Pautasso, C. (2013). Control-flow patterns for decentralized restful service composition. *ACM Transactions on the Web (TWEB)*, *8*(1), 5.

Bennara, M., Mrissa, M., & Amghar, Y. (2014). An approach for composing RESTful linked services on the web. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion* (pp. 977–982). International World Wide Web Conferences Steering Committee.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., ... Winer, D. (2000). *Simple object access protocol (SOAP) 1.1*.

Chinnici, R., Moreau, J.-J., Ryman, A., & Weerawarana, S. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, *26*, 19.

Dustdar, S., & Schreiner, W. (2005). A survey on web services composition. *International journal of web and grid services*, *1*(1), 1–30.

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures* (Unpublished doctoral dissertation). University of California, Irvine.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol–http/1.1.* RFC 2616, June.

Gregorio, J. (2007). *Do we need WADL.* `http://bitworking.org/news/193/Do-we-need-WADL`.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, *5*(2), 199–220.

Hadley, M. J. (2006). *Web application description language (WADL).* Sun Microsystems, Inc.

Kelly, M. (2015). JSON hypertext application language. `https://tools.ietf.org/html/draft-kelly-json-hal-02`.

Kopecky, J., Gomadam, K., & Vitvar, T. (2008). hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on* (Vol. 1, pp. 619–625).

Kopecky, J., Vitvar, T., Bournez, C., & Farrell, J. (2007). Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, *11*(6), 60–67.

Lanthaler, M., & Gütl, C. (2013). Hydra: A Vocabulary for Hypermedia-Driven Web APIs. *LDOW*, *996*.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., ... others (2004). OWL-S: Semantic markup for web services. *W3C member submission*, *22*, 2007–04.

Pautasso, C. (2009). RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, *68*(9), 851–866.

Pautasso, C., & Alonso, G. (2005). Flexible binding for reusable composition of web services. In *International conference on software composition* (pp. 151–166).

Richardson, L., Amundsen, M., & Ruby, S. (2013). *Restful web apis*. " O'Reilly Media, Inc.".

Rosenberg, F., Curbera, F., Duftler, M. J., & Khalaf, R. (2008). Composing restful services and collaborative workflows: A lightweight approach. *Internet Computing, IEEE*, *12*(5), 24–31.

Sporny, M., Kellogg, G., Lanthaler, M., & W3C RDF Working Group. (2014). JSON-LD

1.0: a JSON-based serialization for linked data. *W3C Recommendation*, *16*.

Vairetti, C., Alarcón, R., & Bellido, J. (2016). A Semantic Approach for Dynamically Determining Complex Composed Service Behaviour. *Journal of Web Engineering*, *15*(3-4), 310–338.

Verborgh, R., Arndt, D., Van Hoecke, S., De Roo, J., Mels, G., Steiner, T., & Gabarró, J. (2015). The Pragmatic Proof: Hypermedia API Composition and Execution. *CoRR*, *abs/1512.07780*.

Verborgh, R., & De Roo, J. (2015). Drawing Conclusions from Linked Data on the Web: The EYE Reasoner. *IEEE Software*, *32*(3).

Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., & Vallés, J. G. (2013). Capturing the functionality of Web services with functional descriptions. *Multimedia tools and applications*, *64*(2), 365–387.

Webber, J., Parastatidis, S., & Robinson, I. (2010). *Rest in practice: Hypermedia and systems architecture*. " O'Reilly Media, Inc.".

Xie, C., Cai, H., & Jiang, L. (2013). Ontology Combined Structural and Operational Semantics for Resource-Oriented Service Composition. *Journal of Universal Computer Science*, *19*(13), 1963–1985.