Comput Sci Res Dev (2018) 33:291–303
https://doi.org/10.1007/s00450-017-0384-1

SPECIAL ISSUE PAPER

API governance support through the structural analysis of REST APIs

Florian Haupt[1] · **Frank Leymann**[1] · **Karolina Vukojevic-Haupt**[1]

Published online: 22 September 2017

Abstract Today, REST APIs have established as a means for realizing distributed systems and are supposed to gain even more importance in the context of Cloud Computing, Internet of Things, and Microservices. Nevertheless, many existing REST APIs are known to be not well designed, resulting in the absence of desirable non-functional properties that truly RESTful systems entail. Although existing analysis show, that many REST APIs are not fully REST compliant, it is still an open issue how to improve this deficit and where to start. In this work, we apply structural analysis of REST APIs in order to support API governance, resulting in a set of basic and aggregated metrics that characterize an API set and also guide further governance tasks. We apply the structural analysis on a set of 286 real world APIs and then demonstrate how to derive suitable metrics that represent the perceived complexity of an API, complemented and validated by a survey of developers following the AHP process. As a result, we provide effective support for API governance, helping to identify and remedy problems in APIs.

Keywords REST · Interface description language · Analysis · API governance

1 Introduction

The architectural style *Representational State Transfer (REST)* has become a popular choice for the realization of service-oriented architectures. Based on the core technologies of the World Wide Web (WWW), mainly the Hypertext Transfer Protocol (HTTP) together with URIs and MIME types, it promises simplicity, standards-based interoperability and ubiquitous availability on all kind of platforms [1]. Even more important are the implications of the REST style on the non-functional properties of a REST-compliant software system. Distributed software systems that follow the REST style are assumed to support inter alia software longevity, independent evolution of its components, scalability, and extensibility [2]. The main challenge in achieving these desirable non-functional properties is the REST-compliant design and realization of services.

It has been shown that many APIs that claim to follow the REST style are not REST compliant at all [3–5]. A first step towards a REST compliant API is the correct usage of the HTTP protocol, respecting its syntactical as well as semantic specification [6]. However, being REST compliant typically requires more effort than this [7]. One of RESTs core constraints is called Hypertext as the Engine of Application State (HATEOAS). It demands that clients of a REST API are guided by the responses they receive from the API. Each response contains metadata like hyperlinks or forms that tell the client where it can go next and what actions are possible in the current state of its conversation with the API. Fulfilling this constraint has a major impact on the structure of a REST API, as it typically results in a graph-like structure of resources connected by hyperlinks.

In order to improve the state of the art in the design and realization of REST APIs, it is crucial to be aware of this state of the art. In this context, the goal of this work is to utilize the structural analysis of REST APIs in order to provide API governance support. For that, a framework for the structural analysis of REST APIs [8] is used to derive a set of metrics as well as graphical representations for a given set of APIs. These data then provides an overview and characterization of the set of APIs under investigation. We envision that knowing

✉ Florian Haupt
florian.haupt@iaas.uni-stuttgart.de

1 Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany

and analyzing these data can effectively be applied for supporting API governance tasks. Advantages of our approach are that it can already be applied at design-time (as it requires only API models but no implementations) and that it is easily applicable to huge sets of APIs (as it can be executed automatically).

The rest of the paper is structured as follows. In Sect. 2 we give an overview about existing works on the analysis of REST APIs, including an identification and categorization of different analysis approaches as well as a classification of our work. As we intend to analyze real-world APIs, Sect. 3 gives an overview about common REST API description languages and their spread. Our analysis is based on a metamodel for REST APIs that we have developed in previous work [22,23]. This metamodel is introduced in Sect. 4, followed by Sect. 5 describing the transformation of the Swagger and RAML description languages into this metamodel. The core contribution of this paper, the application of structural analysis of REST APIs for supporting API governance, is presented in Sect. 6, comprising a metrics-based API analysis as well as metrics for calculating user-perceived API complexity. In Sect. 7 we discuss threats to the validity of our work and describe how we minimized them. Section 8 concludes the paper with a discussion of the main results and a short outlook to future work.

2 Related work

Several works already target the analysis of REST APIs. In this section, we first give an overview about relevant related work. Then, we categorize the existing approaches and position our work with respect to them.

2.1 Literature overview

A first analysis of REST APIs has been conducted in [4]. The authors investigated a set of 222 Web APIs taken from ProgrammableWeb.com, a popular Web API directory. Web APIs are further distinguished in RPC-Style, RESTful, and Hybrid. The analysis has been conducted manually and focuses on technical aspects of the selected APIs. The authors present amongst other things statistics about supported representation types, authentication mechanisms, and the availability of API documentation.

The work presented in [9] analyzes a set of 12 REST APIs with respect to a set of five patterns and eight antipatterns. For each of these (anti-) patterns the authors define a corresponding heuristics and detection algorithm. These heuristics and detection algorithms are based on the observation and investigation of request and response messages exchanged with an API. For the analysis, a REST API is first called several times and all request- and response-messages are gathered and stored. Then, the gathered messages are processed by the (anti-) pattern detection algorithms.

The work of [9] is continued and extended in [10]. Here, the authors focus on the analysis of the URI structure of REST APIs using a set of five linguistic patterns and antipatterns that are applied to a set of 15 REST APIs. Although the naming of URIs should not matter to the client at all (as it is supposed to follow RESTs HATEOAS constraint, i.e. to navigate through an API by following hyperlinks), it is still important for the realization, operation and maintenance of REST APIs. The general analysis approach is the same as in [9]. Each (anti-) pattern has a corresponding heuristics and detection algorithm, which are then applied to a set of previously gathered request messages.

In [11] a set of three REST APIs from three well-known cloud providers is analyzed with respect to a set of 73 best practices compiled from literature. The analysis is based on available API documentation and has been conducted manually, followed by a detailed analysis of the results.

In [12] a dataset of 78GB of HTTP traffic from an Italian mobile internet provider is analyzed with respect to REST principles and guidelines. First, all requests targeting APIs for machine consumption (in contrast to web pages consumed by humans) are identified and extracted. Then, the authors define a set of five best practices for REST APIs and a corresponding set of 18 heuristics for the compliance with these best practices. These heuristics are then implemented and applied to a representative sample of the whole dataset. In addition, the same heuristics are used to calculate the maturity level of the investigated REST APIs with respect to the maturity model by Richardson [13].

2.2 Summary and categorization

The existing works on the analysis of REST APIs follow different approaches that can be summarized and categorized as shown in Fig. 1. The analyzes presented in [4,11] are based on human-readable API documentation and have been conducted manually (type 1). In contrast, the analyzes presented in [9,10,13] are based on the automated analysis of request and response messages (type 2). Both analysis types have their advantages and disadvantages.

The first analysis type, the manual analysis of human-readable documentation, can in general cover more and detailed aspects than any automated analysis. In addition, only the documentation of an API is required, i.e. the analysis may even be conducted during design time when an API is not yet implemented. On the other hand, being conducted manually, this type of analysis cannot be applied meaningfully to arbitrary large sets of REST APIs. In addition, the quality of the results strongly depends on the qualification of the humans conducting the analysis as well as on the quality, completeness, and correctness of the API documentation.

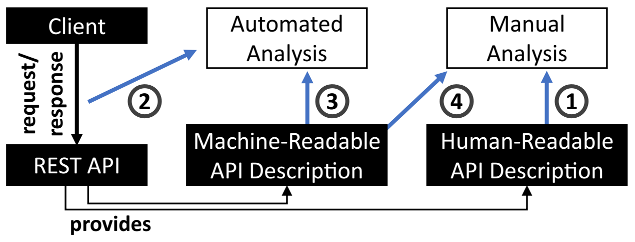


Fig. 1 Analysis approaches

The second analysis type, the automated analysis of request and response messages, can be easily applied to huge sets of APIs in a repeatable and traceable manner. However, it requires that the API under analysis is implemented and accessible. Another challenge is that the analysis only covers those parts of an API that are covered by the set of messages under investigation.

What has not been covered so far, to the best of our knowledge, is the analysis of REST APIs based on machine-readable API descriptions and its application for API governance. REST API description languages like Swagger [14] and RAML [15] gain more and more importance, which amongst other things recently resulted in the Open API Initiative [16] as a standardization approach for API descriptions. We use this potential to allow for new approaches in API governance supported by structural API analysis approaches.

In this paper, we apply an automated analysis (type 3) focusing on the structure of REST APIs (Sect. 6.1) using an analysis framework developed in previous work [8]. This approach can already be applied at design time, as it only requires a description (a model) of an API, but no implementation. Being an automated analysis, it can also be easily applied to huge sets of APIs, making our API governance support also applicable to scenarios comprising many APIs.

The last analysis type, the manual analysis of machine-readable API descriptions (type 4), builds on a graphical representation of the structure of REST APIs and targets the in-depth analysis of selected APIs. This analysis type is also supported by the analysis framework we developed in previous work [8] and can also already applied at design time.

3 Common REST API description languages

There exist many languages for the description of REST APIs from both, academia as well as industry. For our work, we are concentrating on description languages that are commonly used in real world, assuming that API descriptions based on these languages will then be available for a wide range of real-world APIs. In the following, we will first discuss our selection process and then introduce the description languages that have been selected for further investigation, Swagger and RAML.

3.1 Selection process

As a first step towards selecting a suitable set of REST API description languages to be used in our analysis approach, we conducted a Google search for the term “rest api description language”. In the result set of this search, the following names appeared repeatedly: API Blueprint [17], I/O Docs [18], Swagger [14], RAML [15], WADL [19], WSDL 2.0 [20], Open API [16].

To narrow this set further down and to get a better understanding of the dissemination and usage of these languages, we applied additional quantitative criteria as shown in Table 1. For each description language, we conducted a Google search using the search term “{description language name}” + “REST” and noted the total number of search results. We also used StackOverflow.com (SO), a popular developer community, to search for questions having the name of a description language in their title as well as for questions that have been tagged with the corresponding description language. On GitHub.com, a popular service offering Git repositories for software development, we searched for repositories having the name of each description languages in their title. As most of the description languages have their specification documents hosted on GitHub, we also considered how often these specifications have been starred (marked as favorite) and forked.

The results of our searches shown in Table 1 indicate that Swagger seems to be the most widespread description language used in real world projects. What has to be considered here is that Swagger has a special relationship to the Open API specification. The Open API Initiative is highly based on Swagger; the current Open API specification v2.0 is identical to the latest Swagger specification v2.0. Therefore, whenever we consider Swagger in the following, all statements also apply to Open API. Following the numbers in Table 1, we also consider RAML in our work, as it also seems to have some practical relevance. Although WADL scores high for some of the criteria, we do not consider it in this work. The WADL specification is rather old and, according to our experience, continuously losing practical relevance.

3.2 Swagger and RAML

The core of the Swagger framework is the Swagger specification that defines how to describe a REST API as JSON (or YAML) file. In the following, we will call such files that follow the Swagger specification *Swagger files*. In the context of this work, we focus on the structure of a REST API, i.e. the resources and their relationships. When reducing Swagger to this aspects, we can create a simplified metamodel

Table 1 Quantitative selection criteria

	Google search	SO by title	SO by tags	GitHub projects	Spec. star/fork
API blueprint	20,400	79	188	440	4714/1344
I/O docs	4120	0	15	26	1830/430
Swagger	2,790,000	2465	2551	4638	5321/1583
RAML	120,000	148	181	926	2501/367
WADL	93,100	178	192	192	–
WSDL 2.0	20,800	23	7	0	–
Open API	143,000	48	30	662	5321/1583

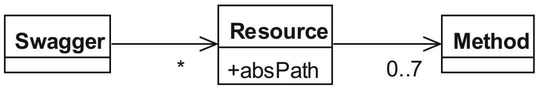


Fig. 2 Simplified Swagger metamodel

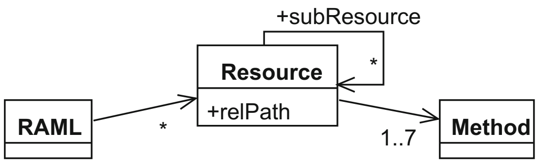


Fig. 3 Simplified RAML metamodel

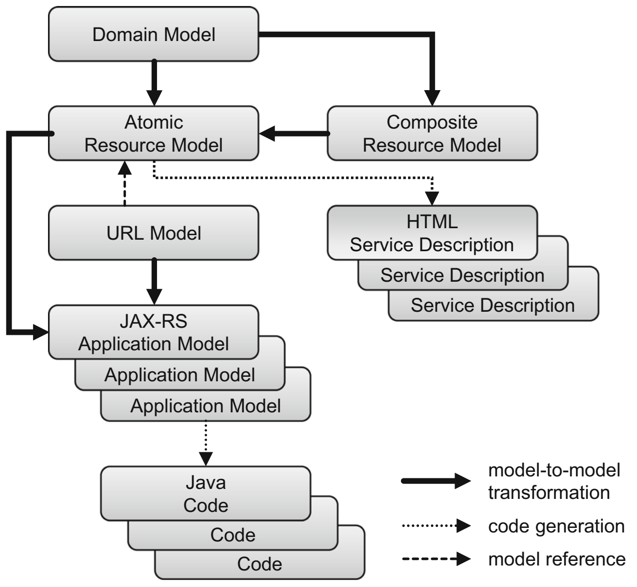


Fig. 4 Metamodels for REST APIs [22]

for it as shown in Fig. 2. A Swagger file describes an REST API as a set of resources. Each resource is identified by an absolute path and supports a set of HTTP methods. Although the HTTP specification [6] defines eight methods, Swagger only supports a subset of six HTTP methods as well as the additional PATCH method [21].

Regarding the general structure, Swagger and RAML files are very similar. A noteworthy difference is however, that RAML allows the explicit modeling of resource hierarchies. A simplified metamodel for RAML focusing on API structure is shown in Fig. 3.

Analyzing the structure of a REST API based on the (simplified) metamodels of Swagger and RAML has some serious drawbacks. First, the different metamodels would require (at least in parts) different realizations for the same analyzes, resulting in increased effort as well as an increased probability for inconsistencies and faults. Second, Swagger models do not explicitly model the structure of a REST API at all. At first glance, Swagger only lists a set of resources. Any relationship between these resources, like the formation of a resource hierarchy, has to be extracted from the structure of the URIs of the resources.

Consequently, we decided that our analysis would be based neither on the Swagger metamodel nor on the RAML metamodel but on a metamodel for REST APIs that we have developed in previous works [22]. In the following section, we will introduce the relevant parts of this metamodel and show how the Swagger and RAML metamodel can be mapped on it.

4 A canonical metamodel for REST APIs

In order to get a better understanding about the structure of REST APIs and to help designers and developers to create better REST APIs, we have developed a set of metamodels for REST APIs as shown in Fig. 4 and successfully integrated them in a model-driven approach for the design and realization of REST APIs [22,31].

The core model is the atomic resource model, which describes a REST API in terms of its basic elements like resources, methods, representations, or query parameters. The composite resource model provides higher level modeling elements that ease the modeling task and the understanding of complex models. A composite resource model based on the concept of conversations has been introduced in [23]. An important feature is the URI model. The HATEOAS constraint of REST demands that clients navigate through an

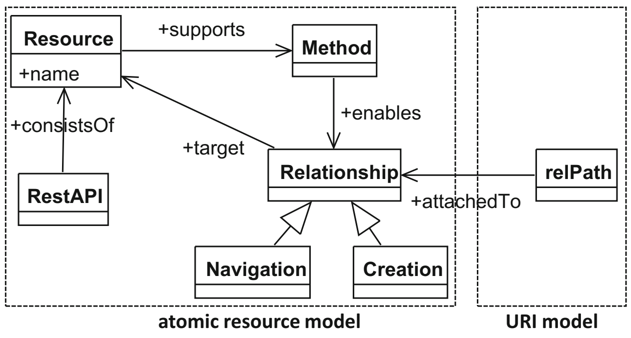


Fig. 5 Simplified metamodel for REST APIs

API independent of any specific URIs by following hyperlinks. The separation of the resource model (which contains no URIs at all) from the URI model reflects this very important aspect of REST and intends to support API designers as well as API clients in following the HATEOAS principle. Details about the other models are not relevant in the context in this paper and can be found in [22].

A simplified version of the metamodel for the atomic resource model as well as for the URI model is shown in Fig. 5. A REST API consists of a set of resources and each resource supports a set of methods. An important distinctive feature of our metamodel is that relationships between resources are not directly attached to the resource but related to the methods of a resource. The rationale behind this is as follows. When submitting for example a GET request to a resource, the response may contain hyperlinks, which then allow navigating further to other resources. Similarly, the submission of a POST request to a resource (e.g. a list resource) may result in the creation of another resource. In summary, relations between resources always depend on the usage of the methods of the source resource of a connection and our metamodel reflects this. An example instance of this metamodel is presented in Fig. 11, showing the model of an example REST API in a graphical editor we developed as part of a toolchain around our REST API metamodel.

The URI model shown in Fig. 5 is defined separately from the atomic resource model. It defines a set of relative paths that are then attached (by reference) to (a subset of) the relationships of the resource model. Regarding the analysis of the structure of an REST API, i.e. the analysis of the resources and their connections, the resource model contains all necessary information and we will therefore ignore the URI model in the following.

5 Model transformations

As discussed in the previous section, we aim at using our atomic resource model as base for the structural analysis of

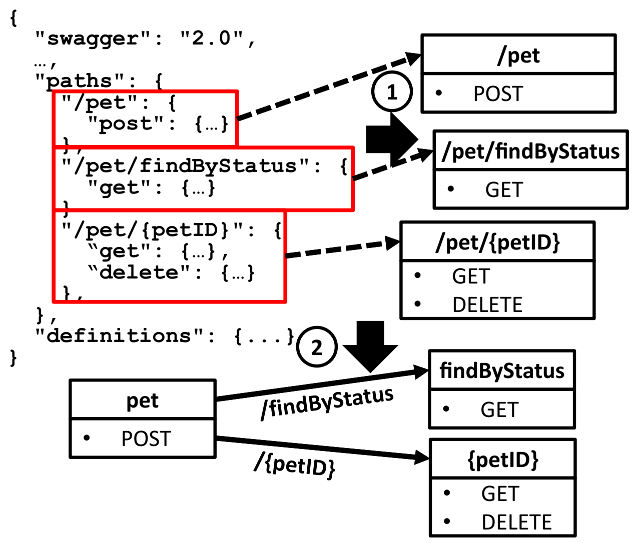

```
{
  "swagger": "2.0",
  ...,
  "paths": {
    "/pet": {
      "post": {…}
    },
    "/pet/findByStatus": {
      "get": {…}
    },
    "/pet/{petID}": {
      "get": {…},
      "delete": {…}
    },
  },
  "definitions": {...}
}
```

Fig. 6 Swagger transformation process

REST APIs. Consequently, we first have to design and realize the transformation of Swagger as well as RAML models into our metamodel. In addition, this process already provides some interesting findings about the characteristics of the description languages we considered in our work (and these characteristics may in turn influence the characteristics of the REST APIs that are designed using them).

5.1 Transformation design

The transformation of a Swagger model comprises two phases as shown in Fig. 6. First, all resources are identified and transformed including their entire detailed configuration like the supported methods, representations or query parameters. Second, the relationships between the resources are identified. Unfortunately, Swagger does not provide any means to describe links between resources explicitly. Instead, the structure of a REST API is usually given by the paths the API provides. Here it is generally accepted that these paths represent hierarchical relationships. We build on this assumption for determining the relationships between resources.

The corresponding algorithm for deriving resource relations from the associated paths, which is applied in the second step of the transformation process, is sketched in Listing 1. The general idea is to first sort all resources by levels, where the level of a resource is determined by the length of its path (in terms of the number of elements of the path). Then, the resources are processed from top level to bottom level (i.e. shortest paths first) and each resource is connected to all resources on higher levels whose path is a prefix of the current resource. Consequently, the resulting resource graph is always a tree, as Swagger is inherently limited to such structures.

Listing 1 Resource structure derivation algorithm for Swagger

```
public void deriveStructure(Swagger model) {
  //<level, paths on this level>
  Map<int, Set<Path>> paths;
  //order paths by level (i.e. length)
  for (Path p : model.getPaths()) {
    paths.get(p.getElements()).add(p);
  }
  //sort levels ascending
  Set<int> sorted = sortAsc(paths.keySet());
  //process paths from shortest to longest
  for (int i : sorted) {
    //process all paths on current level
    for (Path current : paths.get(i)) {
      //look at all higher levels
      for (int x = i-1; x >= 0; x--) {
        for (Path p : paths.get(x)) {
          //if a is a prefix of b, then a is
              parent of b
          if (current.startsWith(p)) {
            connect(p, current);
            found = true;
          }
        }
        //stop if parent resource was found
        if (found) { break; }
}}}}
```

The RAML transformation is for the most parts very similar to the Swagger transformation. The main difference is that the derivation of the resource structure is simpler because RAML already supports the modeling of nested resources structures. As for Swagger, the resulting resource graph is always a tree.

5.2 Findings

A general finding that has already been mentioned is that Swagger as well as RAML provide no explicit means for describing links between resources. In addition, Swagger as well as RAML also do not support to describe the relationship between POST requests on one resource and the resulting creation of another resource. Such relationships are very common (e.g. for list resources) and usually documented in the human-readable description text, but they cannot be included in the formal description of an API. Altogether, these drawbacks either result in APIs that do not exploit the full power of REST (especially the HATEOAS concept) or it results in incomplete API description that only cover a subset of the capabilities an API provides.

Another observation is that relationships between resources sometimes go across multiple levels, i.e. the path of a child resource extends the path of its parent resource by more than one element. For example, an API might offer the resource /api and its child resource /api/products/pid but no resource with a path of /api/products. Although an API with such an URI structure might still be fully REST compliant, it is usually considered a best practice to assume that clients might access any part of a URI and therefore to provide at least some response at any possible URI.

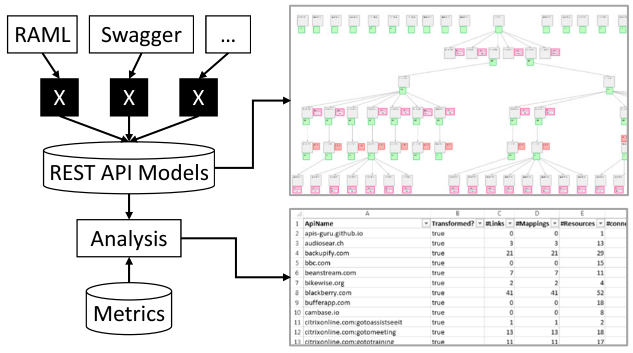
6 API governance support

The general approach of the REST API analysis we conducted is based on a framework for the structural analysis of REST APIs we presented in [8]. An overview is depicted in Fig. 7. Starting from available REST API description documents in Swagger and RAML we transform them into our canonical metamodel and store them in a model repository. This way, the following analysis steps are easily reusable for other REST API description languages, they only need to be transformed into the canonical metamodel.

The models stored in the repository can be automatically processed by the analysis component, which in turn builds on a repository of algorithms that are able to calculate the metrics we are interested in, making this part easily extendable with new metrics as desired. The results of this analysis (i.e. a set of metrics for each REST API) are written to CSV files, allowing further analysis and processing by common office tools. This kind of analysis refers to type 3 as introduced in Sect. 2.2.

The REST API models stored in the repository can in addition be visualized using a graphical editor we developed as part of a toolchain around our REST API metamodel. The graphical representation of the resources and their relationships enables domain as well as REST experts to easily understand and assess the structure of an API. This kind of analysis refers to type 4 as introduced in Sect. 2.2.

The goal of our work is to provide API governance support based on the structural analysis of REST APIs. API governance is a task mainly applied inside an organization, typically aiming at achieving a certain harmonization of APIs in terms of their non-functional properties, best-practices-support, documentation quality or rule compliance in general. Especially when considering Microservice architectures, this task may have to consider a considerably huge



	A	B	C	D	E	
1	ApiName	Transformed?	#Links	#Mappings	#Resources	#conne
2	apis-guru.github.io	true	0	0	1	
3	audiosear.ch	true	3	3	13	
4	backupify.com	true	21	21	29	
5	bbc.com	true	0	0	15	
6	beanstream.com	true	7	7	11	
7	bikewise.org	true	2	2	4	
8	blackberry.com	true	41	41	52	
9	bufferapp.com	true	0	0	18	
10	cambase.io	true	0	0	8	
11	citrixonline.com:gotoassistseeit	true	1	1	2	
12	citrixonline.com:gotomeeting	true	13	13	18	
13	citrixonline.com:gototraining	true	11	11	17	

Fig. 7 Analysis approach [8]

set of APIs. Our approach assumes that API description documents are available, and as discussed in Sect. 3 this assumption is valid in many cases.

All analyzes discussed in the following are based on a set of 286 API description documents retrieved from https://apis.guru, a web page (and API) that describes itself as Wikipedia for WEB APIs. The set includes only APIs that are publicly available (free or paid) and includes renowned providers like Microsoft Azure, Google, BBC, GitHub, Instagram, NYTimes, Spotify, and Wikimedia.

6.1 Metrics-based API analysis

As a first step towards API governance support we propose to characterize a set of APIs using metrics that are related to the structure of an API. Based on expert knowledge and experience we identified a set of potentially relevant metrics, implemented them, integrated them in the analysis framework, and calculated them for each API under investigation. An overview about the aggregated values of all metrics that were calculated during this analysis is given in Table 2. The first group of four metrics concentrates on the resources of an API. The number of resources, i.e. the size of an API, covers a range from a minimum of one resource to a maximum of 264 resources per API. The deviation between the mean value and the median indicates that the distribution is rather uneven and includes breakout values. This can in detail be seen in Fig. 8, which shows the distribution of the API size throughout the set of all APIs. The majority of APIs (53.5%) has a size of 10 or less resources (33% APIs have a size between 1 and 5 resources, and 20.5% APIs have a size between 6 and 10 resources). Another 37.5% of all APIs has a size ranging between 11 and 40 resources and the remaining 9% have a size between 41 and the maximum of 264 resources (the distribution between 111 and 270 resources has been combined into one value in Fig. 8).

The set of APIs considered in the analysis comprises two noteworthy subsets, a set of 39 API models from Microsoft Azure, and a set of 105 API models from Google. As these two sets represent a significant amount of the complete API set, Fig. 8 also shows the distribution of the API size separately for the set of Azure APIs, the set of Google APIs, and the set of all remaining APIs. These three distributions vary in parts. The share of APIs with a size up to five resources is 23% and 36% for the Azure and Google APIs respectively, and nearly 45% for all remaining APIs. In contrast, the share of APIs with a size from six up to ten resources is much smaller for the remaining APIs than for Azure and Google.

The next metric, the number of read-only resources (#ReadOnly Resources in Table 2) counts all resources that

Table 2 Aggregated metrics overview

	Min	Max	Mean	Median
#Resources	1	264	20.3	9
#ReadOnly resources	0	227	10.4	4
#POST	0	93	6.5	3
#DELETE	0	40	2.6	1
#Roots	1	227	8.1	4
#Links	0	248	12.2	4
MaxDepth	0	7	1.8	1
#Components	1	227	8.1	4
Smallest component	1	165	2.4	1
Biggest component	1	165	8.3	3.5
Avg component size	1	165	4.0	2
Biggest component coverage	0.4%	100.0%	54.0%	50.0%

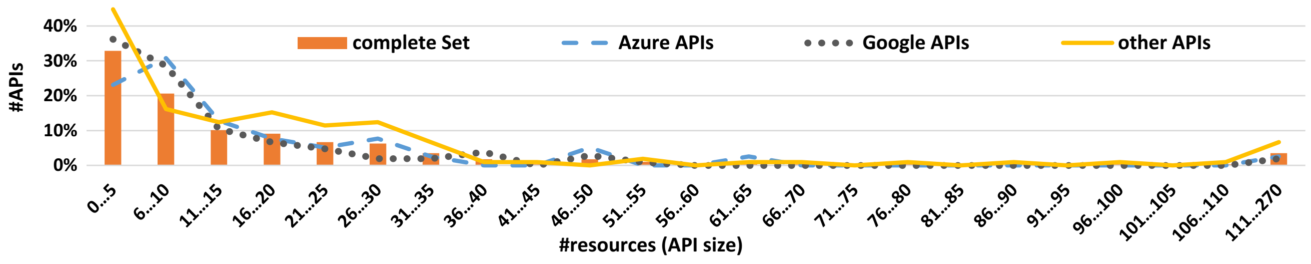


Fig. 8 Distribution of API size (number of resources)

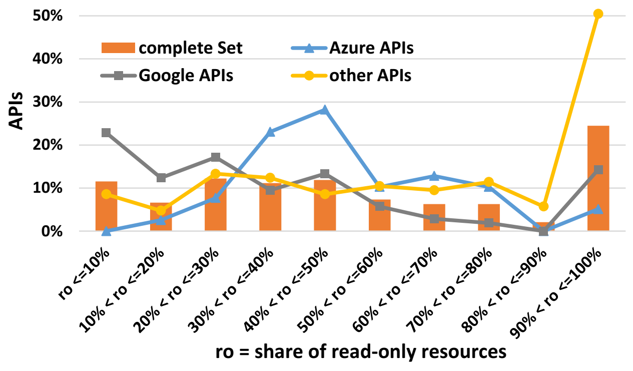


Fig. 9 Share of read-only resources in APIs

support only the GET method but no other methods. For POST and DELETE, we count all resources that support these methods (and maybe others, as usually every resources is supposed to support GET requests).

The distribution of the share of read-only resources in an API is shown in Fig. 9, again for the whole API set (bars) as well as separately for the three subsets (curves). Looking at the whole set, it is noticeable that around 24.5% of all APIs have a share of read-only resources between 90% and 100%, i.e. these APIs focus on information retrieval rather than on content creation and manipulation. Looking at the three subsets, their distributions are rather different. The majority of Azure APIs (61.5%) has a read-only share between 30 and 60%, whereas the distribution for the Google APIs is more even. For the set of all remaining APIs, 50.5% have a read-only share of 90% or more. These differences probably result from the fact, that the Azure and Google APIs provide similar functionality in their APIs (both provide common cloud services) which includes not only information retrieval but also content creation and manipulation. The set of all remaining APIs however covers a much broader spectrum of services, which evidently include a significant set of information services.

Comparing the numbers of the first four metrics in Table 2 shows that read-only resources are very common in REST APIs, in average they make up half of all resources. In addition, we can also read off that there are in general more resources supporting the POST method than resources supporting the DELETE method. This proportion can, at least partially, be explained when we have a closer look at two commonly used resource types, list resources and command resources [24]. List resources are used to manage a set of child resources. The list resource supports GET for retrieving a list of references to all child resources and it supports POST for adding new child resources. Child resources support GET for retrieval, DELETE for deleting, and PUT for updating. Examples of list resources are shown in Fig. 11 (labeled 1). Altogether, list resources add the same amount of POST as well as DELETE methods to an API. The increased occurrence of POST resources is mostly added by so-called command resources. Such resources usually represent functionality that cannot be mapped to one of the other methods (GET, PUT, and DELETE). Typically, each command resource represents one functionality that is called using the POST method. Examples of command resources are shown in Fig. 11 (labeled 2).

The next group of three metrics in Table 2 adds data about links between resources to the analysis. We define any resource that has no incoming links as a root resource. A generally accepted best practice in REST API design, driven by the HATEOAs constraint, is that an API should have only one (or at least few) root resource [25]. However, the numbers in Table 2 show that todays REST APIs usually have quite some root resources. Due to the fact that all APIs that we consider in our analysis are trees, the total amount of links per API is limited by the amount of resources. The maximum depth of an API is given by the longest path of resources that are connected by links. This metric shows rather small values if we compare it to the number of resources, which indicates that APIs are in general more wide than deep. Speaking from a clients view, this means that when navigating through an API there are comparatively little possibilities to navigate deeper into the API, but at each of these steps, there are in average many alternatives to navigate further.

The last group of five metrics in Table 2 adds data about components to the analysis. In graph theory, a component of a graph is defined as a subset of a graph where each node in the subset (i.e. resource) is connected (directly or by a path) to any other node in the subset, but not to any other node outside the subset [26]. Speaking in terms of REST APIs, a component comprises a root resource together with all resources that can be accessed from this root resource. The API shown in Fig. 11 consists of three components. Following the best practice of having exactly one root resource, an API should comprise exactly one component (in the case of trees, the number of root resources and the number of components are always equal). However, the numbers in Table 2 show that APIs usually comprise several components.

If we have a closer look at the size of the components of an API, we can observe that for the average size of the smallest component as well as for the average size of the biggest component the mean value and the median again clearly differ, indicating an uneven distribution of these metrics (similar to the API size). However, this does not apply to the share of an API that is covered by the biggest component (Biggest Component Coverage in Table 2). The distribution of this metric is shown in detail by the bars in Fig. 10. One remarkable aspect is that the share of APIs for which the biggest component covers at least 90% of an API is rather high. This means that around 22% percent of the APIs we analyzed can be viewed

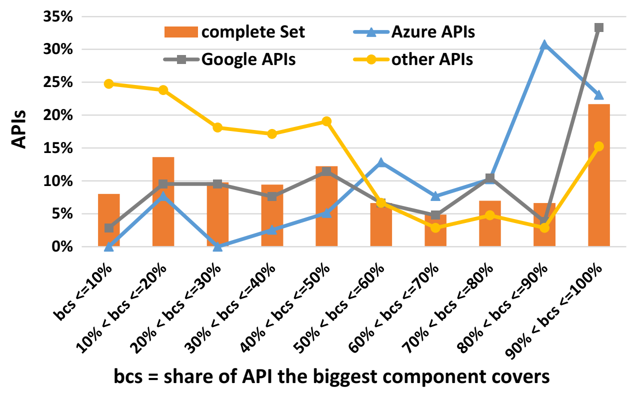


Fig. 10 Share of API covered by the biggest component

as well structured, as they are mostly accessible starting from one root resource. In fact, 52 of 286 APIs (18%) comprise exactly one component.

6.2 Measuring user-perceived API complexity

The metrics we have calculated and discussed so far can serve as a first step towards understanding and characterizing a (potentially huge) set of APIs. Such an analysis can then provide hints for further actions in the context of API governance. Considering the set of REST APIs we analyzed so far, the metrics for example show that many APIs have (considerably) more than one root resource, which violates the HATEOAS constraint of the REST style. Also, the rather uneven distribution of the API size can be a hint that it might be meaningful to investigate if the API size is relevant for API governance (and if additional governance rules are required).

All these metrics are however very low-level, and they often require a significant amount of processing and interpretation before resulting in useful information (as seen in the previous section). We therefore propose to complement the metrics-based analysis by the development of aggregated metrics representing higher-level information about the APIs under investigation. In the following, we are considering the *user-perceived complexity of an API* as an exemplary use case for such an aggregated metric.

Complexity can in general be distinguished in descriptive and perceived complexity [27]. *Descriptive complexity* assumes that there exists an *objective truth* and consequently that such a complexity can be objectively measured. In the context of REST APIs, the descriptive complexity may be determined by investigating the time it takes clients to understand an API or the amount of misunderstandings and errors that occur during its usage. *Perceived complexity* in contrast is a rather subjective concept and solely based on the perception of an observer. In the context of REST APIs, the perceived complexity of an API describes how a user judges the complexity of the API without objectively measuring it. In the following, we are looking at the user-perceived complexity of REST APIs. A high user-perceived complexity may hinder the use and dissemination of an API, which is especially relevant in case an API is offered using a marketplace or public API registry.

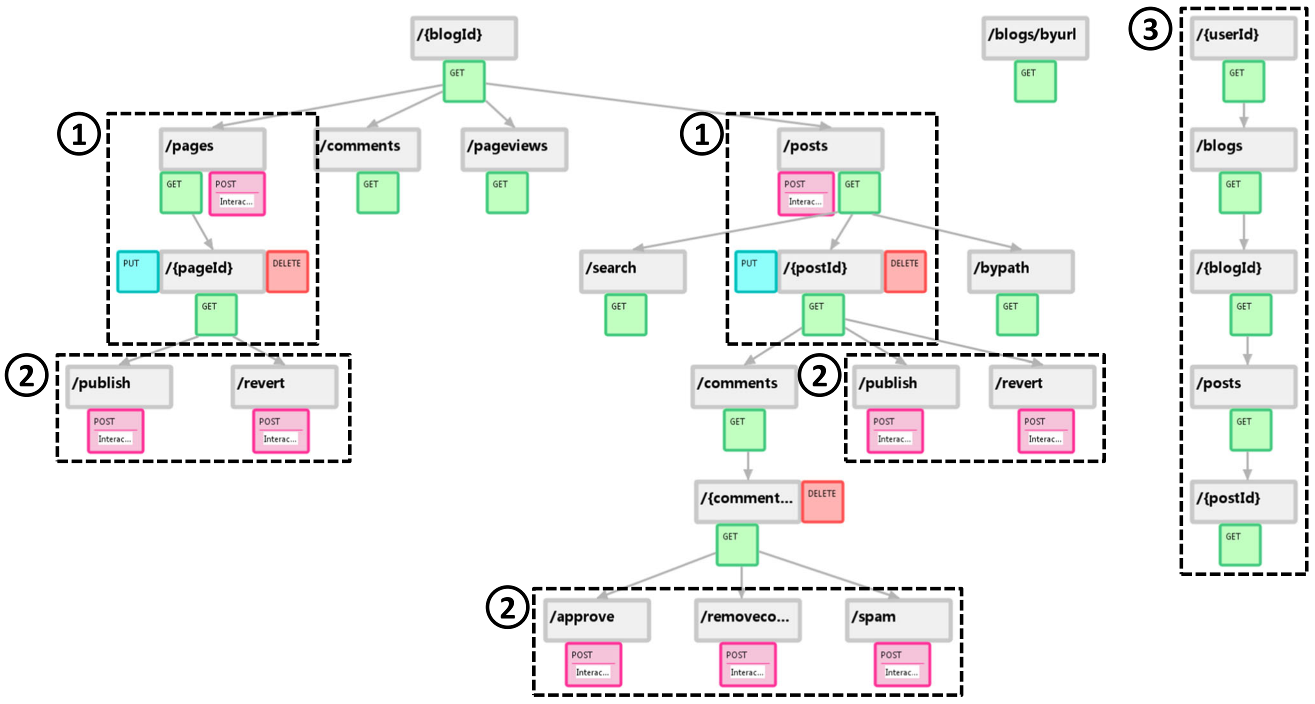


Fig. 11 Graphical representation of the Google Blogger API

In a first step, we selected a random set of ten APIs from the set of 286 APIs introduced in Sect. 6. Then we asked a group of nine software developers, each having at least one year of experience in designing and realizing REST APIs, to rank these APIs based on their complexity (as perceived by the developer). For the ranking, we applied the Analytic Hierarchy Process (AHP) [32] by carrying out a pairwise comparison of all ten APIs. For each pair of APIs, graphical representations of the APIs (as generated by the analysis framework) were shown to the developers and they were asked to decide, which one is more complex. They also had to indicate how much one API is more complex than the other using a scale (as defined by Saaty [32]) ranging from 1 (equally complex) to 9 (extremely more complex). One advantage of applying AHP is, that it provides a quantitative ranking of the APIs under investigation, meaning that it does not only show which API is perceived more complex than another but also how much more complex it is perceived. The result of the described AHP process is depicted in Fig. 12, showing the judgments of the individual developers as bars (D1 to D9) and the consolidated ranking as line.

In a second step, we selected (based on expert knowledge and experience) a subset of two basic metrics (as introduced in Sect. 6.1) as candidates for defining an aggregated metric that represents the user-perceived complexity of an API. Then, we composed out of these basic metrics new aggregated metrics, resulting in a set of five metrics candidates as shown in Eq. 1:

$$\begin{aligned}
M_1 &= \#roots \\
M_2 &= BiggestComponentCoverage \\
M_3 &= M_1 + M_2 \\
M_4 &= M_1 + M_2^2 \\
M_5 &= M_1^2 + M_2
\end{aligned} \tag{1}$$

These five metrics have then been applied to the set of ten APIs that have been ranked by the users before. To ensure comparability with the AHP results, the resulting values are normalized to a scale between 0 and 1. A graphical representation of the result of this process is shown in Fig. 13. If we consider only the order of the ranking, M1 as well as M4 result in the same ranking order as the user ranking (their curves are monotonously rising, same as the user ranking). Another approach of comparing the different metrics candidates is to calculate their mean square deviation from the user ranking. The result of this calculation is shown in Fig. 14, indicating M3 as the best candidate as it deviates least from the user ranking. M3 however, as shown in Fig. 13, would result in a different ranking order than the user ranking.

For the use case under investigation, the assessment of the user-perceived complexity of an API, we were able to derive suitable metrics based on structural API analysis. It is obvious, that this approach gets more complex when considering more metrics and also additional metric combinations. However, the presented process can easily be automated, including the combination of metrics to aggregate as well as the evaluation of their deviation from the expected result.

7 Threats to validity

The results of our work presented in the previous sections are subject to several threats to their validity. In the following, we will discuss how we proceeded in our work in order to minimize these threats.

7.1 Internal validity

The model-to-model transformations from Swagger and RAML into the canonical resource metamodel might dis-

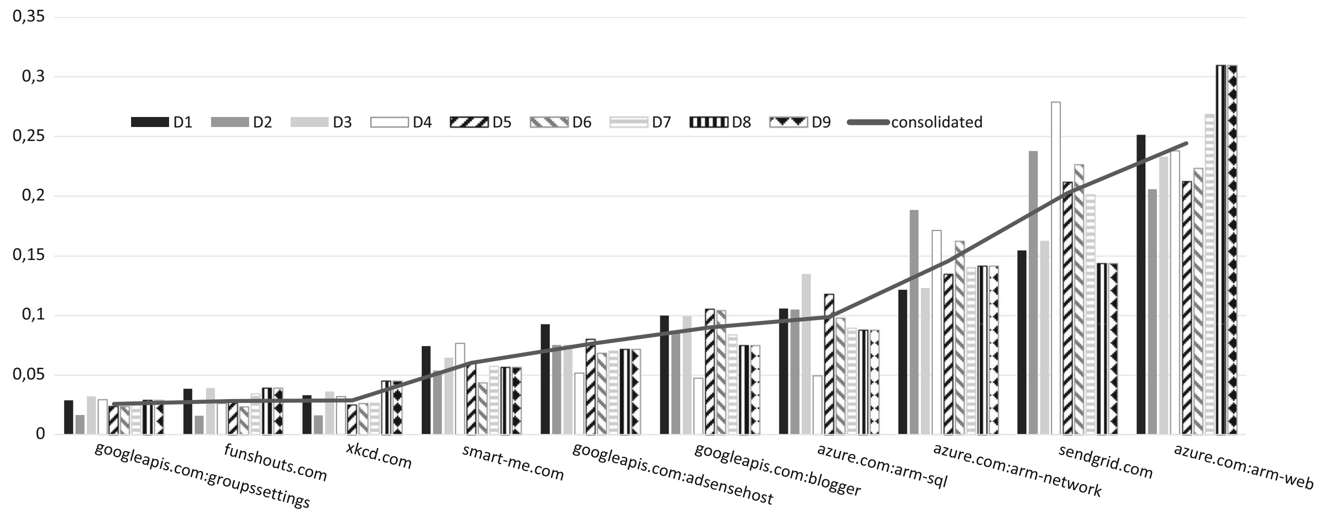


Fig. 12 Perceived API complexity according to AHP results

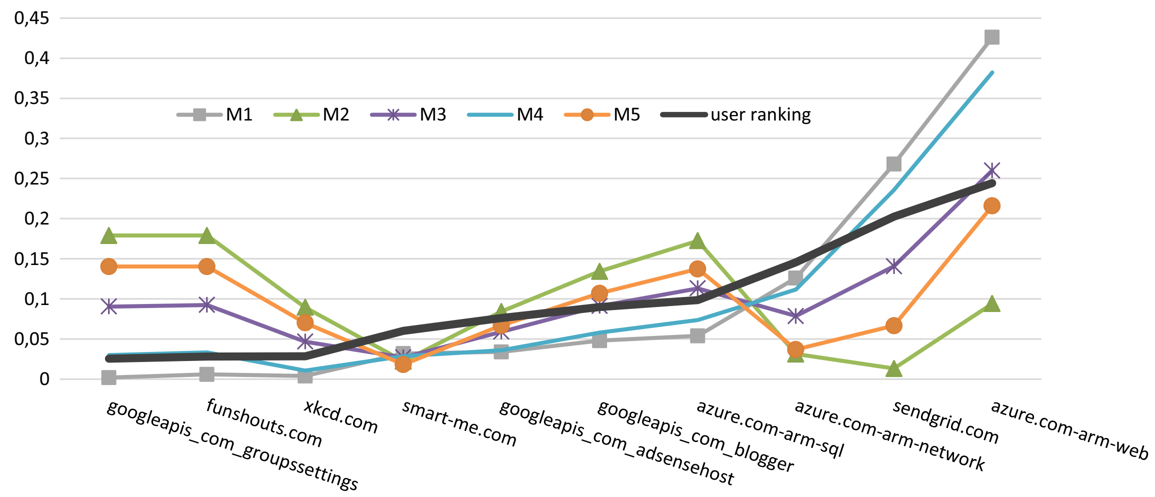


Fig. 13 Perceived API complexity according to metric candidates

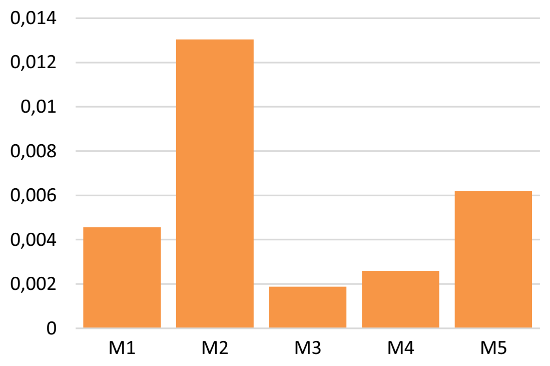


Fig. 14 Mean square deviation from user ranking

tort the structure of an API description, and therefore also the analysis results, in case they do not follow the conceptual mapping described in Sect. 5 correctly. We tackled this threat by manually checking a random sample of source and target models for their transformation being correct and coherent in terms of content. We randomly selected a sample of 30 model pairs out of a total of 286, and 29 of the reviewed transformations were found to be correct. The remaining transformation showed some errors due to some very unusual URI naming that could not be properly processed by the transformation algorithm.

Many of the metrics used in the analysis of the resource structure are based on graph algorithms. Errors in these graph algorithms (or their implementation) might result in incorrect values for the metrics and therefore distort the analysis results. We reduced this threat by following software engineering best practices. We reused existing and mature graph libraries (JGraphT), we implemented unit tests for all analysis algorithms, and we manually validated the correctness of the analysis for a random sample of 30 APIs.

7.2 External validity

The general significance of the analysis results presented in Sect. 6 depends on the set of analyzed REST API descriptions. We minimized this threat by analyzing a set of 286 descriptions, all of them describing publicly available real-world REST APIs, including well-known ones like for example several Google APIs, the GitHub API, or the Instagram API. In addition, we decided to examine Swagger and RAML as they are among the most popular and common REST API description languages in practice. This contributes to the relevance of our analysis results as the set of analyzed REST API descriptions reflects the state-of-the-art of todays REST APIs.

Our analysis is based on the assumption that the REST API descriptions that were transformed and analyzed describe the real APIs completely and correctly. Incomplete and faulty descriptions might distort the analysis results, as they would only apply to the descriptions but not to the real APIs. We minimized this threat by manually comparing a random sample of 64 REST API descriptions with the APIs they describe. We found out, that all of the checked API descriptions are correct and complete. Another indication for the correctness and completeness of the analyzed REST API description is given by the fact, that the Swagger files we used are serving as the base for other API related services like https://any-api.com and https://datafire.io.

It should however be noted that Swagger and RAML, in contrast to our metamodel, do not support the explicit modeling of hyperlinks between resources at all. If a REST API provides hyperlinks that do not follow the URL structure of an API, then these relationships between resources cannot be explicitly represented in the corresponding Swagger or RAML description document. This can then lead to analysis results that are still correct with respect to the API description document, but not necessarily correct with respect to the API itself. This fact has however no or only little influence to the results presented in this paper, as only very few APIs provide such hyperlinks at all. Regarding the random sample of 64 REST API descriptions that we manually compared with their underlying APIs, we found no API that contains this kind of hyperlinks.

All the data our analysis is based on as well as the implementation of the model-to-model transformation and the analysis are available online at http://www.iaas.uni-stuttgart.de/rest. This enables anyone to replicate, review, validate, and reuse our work.

8 Discussion and outlook

Our work we presented in this paper applies structural analysis of REST APIs for supporting API governance tasks. Advantages of this are that it can already be applied at design-time (as it requires only API models but no implementations) and that it is easily applicable to huge sets of APIs (as being executable automatically).

The metrics-based analysis provided a high-level overview and characterization of a large set of real-world REST APIs. We discovered that the APIs under investigation are on average small with a median of nine resources per API, but that the distribution of the API size is rather uneven and that some APIs have more than 250 resources. Therefore, it seems sensible that any works on the design of REST APIs should be able to cope with such huge APIs. Another interesting result is that read-only resources are very common and that there is even a subset of APIs that are completely read-only.

The measurement of user-perceived complexity, based on the metrics-based analysis, demonstrated that aggregated metrics can provide even further benefits for API governance tasks. We systematically derived metrics for the user-perceived complexity of APIs and validated them by a survey among API designers and developers. Such metrics can then be automatically applied to APIs without requiring additional user input.

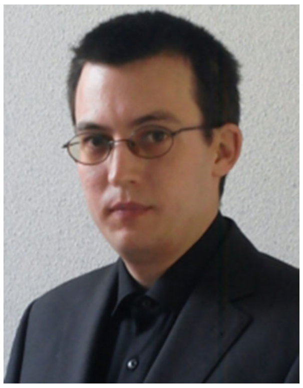
For future work, we think of automating the detection of reoccurring structures in REST APIs. Similar to existing works in the context of business process models [28] we can automatically analyze a set of REST API models for frequent structures. In a next step, these reoccurring structures can be analyzed by REST experts, which might finally result in the detection of structural REST API patterns. It might also be worth it, to investigate if existing REST API patterns [29,30] can be validated by analyzing our set of REST API models.

Acknowledgements This work has been partially funded by the German Research Foundation (DFG) within the Cluster of Excellence in Simulation Technology (EXC310/2) at the University of Stuttgart.

References

1. Webber J, Parastatidis S, Robinson I (2010) REST in practice: hypermedia and systems architecture. O'Reilly Media, Sebastopol
2. Fielding RT, Taylor RN (2002) Principled design of the modern Web architecture. ACM Trans Internet Technol 2:115–150
3. Renzel D, Schlebusch P, Klamma R (2012) Todays top RESTful services and why they are not RESTful. WISE, london
4. Maleshkova M, Pedrinaci C, Domingue J (2010) Investigating web APIs on the World Wide Web. In: The 8th IEEE European conference on web services (ECOWS 2010), 1–3, Ayia Napa, Cyprus
5. Adamczyk P, Smith PH, Johnson RE, Hafiz M (2011) REST and Web services: In theory and in practice, REST: from research to practice. Springer, New York
6. Fielding R, Reschke J (2014) Hypertext transfer protocol (HTTP/1.1): Semantics and Content", RFC 7231. http://www.ietf.org/rfc/rfc7231.txt
7. Haupt F, Fischer M, Karastoyanova D, Leymann F, Vukojevic-Haupt K (2014) Service composition for REST, In: IEEE 18th international enterprise distributed object computing conference (EDOC), pp 110–119
8. Haupt F, Leymann F, Scherer A, Vukojevic-Haupt K (2017) A framework for the structural analysis of REST APIs. In: Proceedings of the IEEE international conference on software architecture (ICSA 2017)
9. Palma F, Dubois J, Moha N, Guhneuc YG (2014) Detection of REST patterns and antipatterns: a heuristics-based approach, ICSOC 2014. Springer, Berlin
10. Palma F. Gonzalez-Huerta J, Moha N, Guhneuc Y.G, Tremblay G (2015) Are restful apis well-designed? Detection of their linguistic (anti) patterns. In: International conference on service-oriented computing. Springer, Berlin
11. Petrillo F, Merle P, Moha N, Guhneuc YG (2016) Are REST APIs for cloud computing well-designed? An exploratory study. Springer, Berlin
12. Rodriguez C, et al. (2016) REST APIs: a large-scale analysis of compliance with principles and best practices. In: International conference on web engineering, Springer
13. Fowler M (2010) Richardson maturity model: steps toward the glory of rest. http://martinfowler.com/articles/richardsonMaturityModel.html
14. Swagger. http://swagger.io/
15. RESTful API modeling language (RAML). http://raml.org/
16. Open API initiative. https://www.openapis.org/
17. API blueprint. https://apiblueprint.org/
18. I/O Docs. http://mashery.github.io/
19. Hadley MJ (2006) Web application description language (WADL). https://www.w3.org/Submission/wadl/
20. Chinnici R, et al. (2007) Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation 26
21. Dusseault L, Snell J (2010) PATCH method for HTTP, RFC 5789. https://tools.ietf.org/rfc/rfc5789.txt
22. Haupt F, Karastoyanova D, Leymann F, Schroth B (2014) A model-driven approach for REST compliant services. In: ICWS
23. Haupt F, Leymann F, Pautasso C (2015) A conversation based approach for modeling REST APIs. In: IEEE WICSA 2015
24. Allamaraju S (2010) Restful web services cookbook: solutions for improving scalability and simplicity. O'Reilly Media Inc, Sebastopol
25. Nottingham M. Home documents for HTTP APIs. https://tools.ietf.org/html/draft-nottingham-json-home-05
26. Bondy JA, Murty USR (1976) Graph theory with applications. Macmillan, London

27. Schlindwein SL, Ison R (2004) Human knowing and perceived complexity: implications for systems practice. Emerg Complex Organ 6(3):2732
28. Skouradaki M, Andrikopoulos V, Kopp O, Leymann F (2016) RoSE: reoccurring structures detection in BPMN 2.0 process model collections, OTM 2016. Springer, Berlin
29. Pautasso C, Ivanchikj A, Schreier S. RESTalk pattern language—patterns for RESTful conversations. http://restalk-patterns.org
30. Pautasso C, Ivanchikj A, Schreier S (2016) A pattern language for RESTful conversations. In: Proceedings of the 21st European conference on pattern languages of programs (PLoP), ACM
31. Vukojevic-Haupt K, Haupt F, Leymann F, Reinfurt L (2015) Bootstrapping complex workflow middleware systems into the cloud. e-Science 2015, IEEE
32. Saaty TL (2008) Decision making with the analytic hierarchy process. Int J Serv Sci 1(1):83–98



Florian Haupt is a research associate and PhD student at the University of Stuttgart. His research focuses on the design and realization of REST APIs. Florian was part of the Migrate! Project, which investigated the environmental effects of cloud computing. Besides his various teaching activities, he is involved in further projects related to cloud computing, the TOSCA standard, and Microservices.

Frank Leymann is a full professor of computer science and director of the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. His research interests include service-oriented architectures and associated middleware, workflow- and business process management, cloud computing and associated systems management aspects, and patterns. Frank is co-author of more than 300 peer-reviewed papers, more than 40 patents, and several industry standards. He is on the Palsberg list of Computer Scientists with highest h-index.

Karolina Vukojevic-Haupt is a research associate and PhD student at the University of Stuttgart. Her research focuses on the automated on-demand provisioning of workflow infrastructures and simulation services in Cloud environments, enabling the efficient execution of simulation experiments. Her work is part of the cluster of excellence in simulation technology (SimTech). Besides that, Karolina works on topics related to application system architecture, quality of services, and resiliency.