International Journal of Cooperative Information Systems
Vol. 26, No. 2 (2017) 1742001 (37 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218843017420011

Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns

Francis Palma*
Department of Electrical and Computer Engineering
Concordia University
1515 St. Catherine West, Montréal, QC, Canada H3G 2W1
francispalmaphd@gmail.com

Javier Gonzalez-Huerta
Software Engineering Research Lab Sweden
Blekinge Institute of Technology
Campus Gräsvik, SE-371 79 Karlskrona, Sweden

Mohamed Founi, Naouel Moha and Guy Tremblay
Département d'informatique
Université du Québec à Montréal, C. P. 8888, Succ. Centre-ville
Montréal, QC, Canada H3C 3P8

Yann-Gaël Guéhéneuc
Ptidej Team, Département de Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, C.P. 6079, Succ. Centre-ville
Montréal, QC, Canada H3C 3A7

Received 15 December 2016
Accepted 21 December 2016
Published 16 May 2017

Identifier lexicon may have a direct impact on software understandability and reusability and, thus, on the quality of the final software product. Understandability and reusability are two important characteristics of software quality. REpresentational State Transfer (REST) style is becoming a *de facto* standard adopted by software organizations to build their Web applications. Understandable and reusable Uniform Resource Identifers (URIs) are important to attract client developers of RESTful APIs because *good* URIs support the client developers to understand and reuse the APIs. Consequently, the use of proper lexicon in RESTful APIs has also a direct impact on the quality of Web applications that integrate these APIs. *Linguistic antipatterns* represent poor practices in the naming, documentation, and choice of identifiers in the APIs as opposed to *linguistic patterns* that represent the corresponding best practices. In this paper, we present the Semantic Analysis of RESTful APIs (SARA) approach that employs both syntactic and semantic analyses for the detection of linguistic patterns and antipatterns in RESTful APIs. We provide detailed definitions of 12 linguistic patterns and antipatterns and define and apply their detection algorithms on 18 widely-used RESTful APIs, including `Facebook`, `Twitter`, and `Dropbox`. Our detection results show that linguistic patterns and antipatterns do occur in major RESTful APIs in particular in the form of poor

*Corresponding author.

documentation practices. Those results also show that SARA can detect linguistic patterns and antipatterns with higher accuracy compared to its state-of-the-art approach — DOLAR.

Keywords: RESTful APIs; linguistic antipatterns; patterns; detection; semantic analysis; Latent Dirichlet Allocation (LDA); second-order similarity; documentation.

1. Introduction

Service-Oriented Architecture (SOA) has changed the way software systems are developed, deployed, and consumed.[11] The REpresentational State Transfer (REST) architectural style[12] is becoming a *de facto* standard, adopted by large software organizations like `Facebook`, `Twitter`, `Dropbox`, and `YouTube`, for developing and publishing their services, also known as their RESTful APIs.

In REST, understandable and reusable Uniform Resource Identifiers (URIs) facilitate the task of RESTful API developers, firstly with the comprehension of APIs, and secondly, with the maintenance and evolution of RESTful APIs. Moreover, RESTful APIs with consistent naming practices may attract client developers more than poorly designed or named ones[24] because client developers must understand the providers' APIs while designing and developing their Web-based applications that use these APIs. Therefore, in the design, development, and use of RESTful APIs understandability and reusability are two major quality factors.

Source code lexicon impacts the understandability, reusability, and, overall, the quality of software systems.[20] APIs designers tend to use relevant identifiers to name software entities.[19] In REST, linguistic relations among resources, services, and parameters are crucial[13] and the lack of such linguistic relations and/or poor naming may degrade the overall design of RESTful APIs and translate into *linguistic antipatterns* (LAs).

In the context of REST, *LAs are poor solutions to common recurring naming problems, which may hinder* (1) *the consumption of RESTful APIs by client developers and* (2) *the maintenance and evolution of RESTful APIs by API developers. Linguistic patterns, in contrast, represent good solutions to common naming problems and facilitate the consumption and maintenance of APIs.*

A number of *best* and *poor* linguistic practices for RESTful APIs design have been reported in the literature[6,13,24] but they do not provide clear and detailed descriptions to allow their automated detection. In our previous work,[30] we redefine those best and poor practices as patterns and antipatterns, respectively. For example, ✗*Contextless Resource Names*[13] is a linguistic antipattern that describes a URI composed of nodes from different semantic contexts as in the URI www.example.com/newspaper/player where "`newspaper`" and "`player`" do not belong to the same semantic context or domain. On the contrary, ✓*Contextualized Resource Names*[13] is a linguistic pattern describing a URI composed of nodes that belong to the same semantic context, which help developers to better understand the resources or the interaction context with the server and, thus, increase the understandability and reusability of a given API. The URI www.example.com/newspapers/media is an example of this pattern, since "`newspapers`" and "`media`" belong to the same semantic context or domain. In RESTful APIs development, the automatic detection of such linguistic patterns and antipatterns is a means to assess their understandability and reusability. However, no previous work except Detection of Linguistic Antipatterns in REST (DOLAR), the approach presented in our previous work,[30] analyzed linguistic patterns and antipatterns in RESTful APIs.

DOLAR[30] relies on WordNet[a] and Stanford's CoreNLP,[b] two general-purpose English dictionaries, to perform syntactic and semantic analyses of RESTful APIs for detecting linguistic patterns and antipatterns in RESTful APIs. In addition, in that work we also defined a set of five linguistic antipatterns and their corresponding patterns inspired from the literature[6,13,24] by using a consistent template. We applied DOLAR in an empirical study to assess the linguistic quality of 15 well-known RESTful APIs including `Facebook`, `Twitter`, `Dropbox`, and `YouTube`. Finally, we also empirically validated the precision and recall of DOLAR, in which we obtained, on average, precision and recall higher than 75%. Indeed, the detection accuracy of DOLAR was higher than 95% for those patterns and antipatterns that require only syntactic analysis. However, our semantic analysis suffered of lower detection accuracy, just over 60% since: (1) it used general-purpose English dictionaries (i.e. *WordNet and Stanford CoreNLP*) instead of using domain-specific dictionaries; and (2) it applied limited semantic analysis techniques, e.g. used only the sets of synonyms for two identifiers to decide their semantic similarity.

In this paper, we extend our previous work to overcome some of the observed limitations. We revisit the DOLAR approach and propose Semantic Analysis of RESTful APIs (SARA), a detection approach capable of performing improved semantic analysis of RESTful APIs. In addition to using WordNet and Stanford's CoreNLP general-purpose English dictionaries, SARA combines the Latent Dirichlet Allocation (LDA)[8] topic modeling technique from the natural language processing (NLP) domain and employs second-order semantic similarity metric[16,17] based on the distributional similarity between URI nodes to decide their semantic similarity. SARA improves DOLAR by capturing the proper context (e.g. in the form of topics) for the nodes in a URI from its documentation. In addition, we defined a new linguistic (anti)pattern — *Pertinent versus Nonpertinent Documentation* — inspired from the object-oriented (OO) programming domain.[3] Finally, we also provide a more detailed and up-to-date description of related works, showing the lack of a comprehensive approach for the semantic analysis of RESTful APIs. Finally, we describe the Service-Oriented Framework for Antipatterns (SOFA) framework showing the architecture of the integration of SARA and SOFA.

[a]wordnet.princeton.edu.
[b]nlp.stanford.edu/software/corenlp.shtml.

In addition, we report the results of an empirical study that aims to assess SARA's detection accuracy, in which we use the same validation dataset as in our previous work.[30] The comparison of the detection accuracies of SARA and DOLAR confirms a significant improvement and shows that SARA overcomes DOLAR's limitations related to semantic analysis. We also applied SARA to analyze 12 REST linguistic patterns and antipatterns on the set of 18 well-known RESTful APIs — including `Facebook`, `Twitter`, `Dropbox`, and `YouTube`.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the 12 linguistic patterns and antipatterns in REST. Section 5 introduces the SARA approach. Section 6 presents a validation of SARA and a comparison with the DOLAR approach. Finally, Sec. 7 concludes the paper and sketches future work.

2. Related Work

Researchers have used linguistic analysis techniques to detect linguistic antipatterns and to check for consistency between source code and comments in OO systems.[1,4,5,21,25,34] Various NLP techniques have been applied in software engineering, in particular, for assessing the quality of OO software systems documentation[15] and Web APIs[18,23,24,28,31,40] documentation.

In the following, we discuss some relevant researches done on assessing the linguistic quality of source code and documentation, which employ both the syntactic and semantic analyses.

2.1. *Syntactic and semantic analyses of source code*

Abebe *et al.*[1] present the first set of lexicon bad smells in OO source code and a toolsuite that uses semantic analysis techniques for their detection. Arnaoudova *et al.*[4] present the first definition of *linguistic antipatterns*, define 17 linguistic antipatterns in OO programming (i.e. recurring poor practices related to inconsistencies among the naming, documentation, and implementation of a software entity), and implement their detection algorithms. The authors search for differences between the identifiers used for software entities (e.g. method names and return types) and their implementation and/or documentation. For example, one antipattern is called "*Is*" *returns more than a Boolean*, which analyzes the name of a method starting with "Is" and checks if the method returns a Boolean.[4]

Semantic analyses are also applied to Web services design and development.[25,36] Rodriguez *et al.*[36] present a study on poor linguistic practices identified on a set of WSDL descriptions and provide a catalog of Web services discoverability antipatterns. These antipatterns focus on the comments, elements names, or types used for representing the data models in WSDL documents. Mateos *et al.*[25] present a tool to detect a subset of the antipatterns presented in Ref. 36.

Other researchers also use semantic analyses in different aspects of the software development life cycle.[5,21,34] For example, Lu *et al.*[21] define an approach to improve code searches by identifying relevant synonyms using the WordNet English lexical database. Arnaoudova *et al.*[5] perform analyses of identifiers renaming in OO systems and classify them. Finally, Rahman and Chanchal[34] present an approach to automatically suggest relevant search terms based on the textual descriptions of software change tasks. These approaches are tailored to OO identifiers and their consistencies with comments[1,4] or to traditional SOAP-based Web services interfaces.[25,36] Therefore, they cannot be applied to RESTful APIs due to the peculiarities of their development life cycle and their consumption nature. For example, the invocation of RESTful services relies on a uniform interface formed using HTTP methods to access or modify resources via URIs.

In addition, some researchers try to deal with the linguistic aspect of RESTful APIs. For example, Hausenblas[14] performs a subjective analysis of RESTful APIs to assess the quality of the URIs naming. Yet, he does not perform an automatic nor a systematic analysis. Parrish[32] also performs a subjective lexical comparison between two well-known RESTful APIs, e.g. `Facebook` and `Twitter`. He analyzes, for example, the use of verbs and nouns in URIs naming and concludes that developers should use nouns instead of verbs while designing REST URIs.

Although the above approaches deal with linguistic aspects of RESTful APIs, they only rely on the subjective view of a set of good linguistic practices and recommendations. Thus, there is no dedicated approach to automatically assess the linguistic quality of RESTful APIs by detecting poor and best practices.

2.2. *Semantic analysis of software documentation*

High quality and self-descriptive source code comments help in developing highly maintainable systems. Khamis *et al.*[15] propose the `JavadocMiner` approach for assessing the quality of inline documentation relying on heuristics both in terms of language quality and consistency between source code and comments. The defined heuristics belong to two different categories: (1) *internal* (*natural language quality*) *comment analysis* where quality of natural language used are assessed (e.g. *Words Per Javadoc Comment Heuristic* checks if the methods or classes are under- or over-documented); and (2) *code/comment consistency analysis* where the consistency between source code and comments are checked (e.g. *Documentable Item Ratio Heuristic* checks if a method documents all its aspects including return type, parameters, and thrown exceptions).

RESTful APIs are being adopted by large software companies, such as `Facebook`, `Dropbox`, or `Twitter`, to develop and offer their services. Client developers must follow well-documented RESTful APIs to properly consume the services and resources offered by those APIs. However, there are only a few standards and guidelines that guide RESTful API development process.[24,28]

In a book on RESTful APIs design principles, Masse[24] proposes an exhaustive list of RESTful APIs design principles including those related to the design of

URIs, proper use of HTTP methods and HTTP status codes, metadata design, and best practices for resource representations. Also, the Open Mobile Alliance (OMA) provides guidelines for designing RESTful APIs exhibiting *REST*fulness and for properly documenting the APIs, for example, not using verbs as resource identifiers or specifying API versioning within URIs.

Several works (e.g. Refs. 18, 23 and 31) deal with the generation of semantically richer documentations of RESTful APIs because the majority of the RESTful APIs are textually described in PDF or HTML documents that are not machine-readable and useful only to developers. Kopecky *et al.*[18] propose HTML for RESTful Services (hRESTS) to facilitate the generation of machine-readable APIs documentation. hRESTS focuses only on service's operations, inputs, and outputs.

Panziera and Paoli[31] propose a set of best practices for building self-descriptive RESTful services, which can be both human-readable and machine-processable (e.g. by using a common vocabulary for RESTful resources). The authors also propose a framework to collect documentation information for semiautomatically generating complete and updated descriptions of RESTful services. They evaluate their framework and report the accuracy of identifying resources correctly with precision and recall of 72% and 77%, respectively. Maleshkova *et al.*[23] devise an approach to formally describe RESTful APIs with the goal of enabling their wider adoption. The authors relied on hRESTS[18] and developed a tool called `SWEET` that supports users in automatically creating the semantic descriptions and adding semantic annotations to RESTful APIs to facilitate resource and service discovery, their composition, and the invocation of RESTful APIs.

Finally, Treude *et al.*[40] develop a search-based approach for automatically extracting tasks (i.e. a set of specific programming actions to be undertaken) from software documentations. The authors try to minimize the gap between the information needs of software developers and the documentation structure/content and, thus, assist developers in documentation navigation. Using the proposed approach, which utilizes NLP techniques, the authors extract more than 70% tasks from two large corpuses of software documentation. However, unlike our goal in this paper, the goal of this work is not to assess the linguistic quality of software documentations.

2.3. *Semantic analysis of web interfaces*

Studies have been performed to investigate and analyze services interfaces to measure their linguistic quality, in particular for SOAP Web services[7,41] and for RESTful APIs.[33,35]

For example, Wei *et al.*[41] present a novel framework and algorithms to analyze service interfaces, the SOAP Web services, in particular. The authors target the large and overloaded services with the goal to ease their integration and interoperability. The proposed framework proposes to refactor large interfaces and is validated with real commercial logistic systems like FedEx. However, the goal of this work is far from ours: we target the linguistic design quality of Web interfaces, namely for RESTful APIs.

In another work, Bertolino *et al.*[7] model the SOAP Web service behavior protocol, i.e. how clients should behave while interacting with the service. The authors propose the StrawBerry[7] method to automatically derive the Web service behavior protocol from its WSDL interface. The proposed StrawBerry method relies on data-type analysis and is evaluated on the Amazon e-commerce service. The main goal of this work is to facilitate the automatic discovery of the behavior protocol of a Web service, and, if required, to compensate the lack of information through its behavior model created by the StrawBerry.

Works have also been done in the domain of RESTful APIs to perform their automatic analysis. For example, Petrillo *et al.*[33] perform a thorough survey on the REST literature and report a collection of 73 best practices in designing RESTful APIs. The assumption is that those best design practices facilitate APIs understandability and reusability. The authors perform a study with three well-known RESTful APIs from three cloud providers, namely Google Cloud Platform, OpenStack, and Open Cloud Computing Interface (OCCI). The goal of this work is to evaluate those three APIs based on the identified best practices and results show that Google Cloud Platform, OpenStack, and the OCCI, respectively, conform to 66%, 62%, and 56% of the identified best design practices from the literature. However, this work does not target the semantic analysis of RESTful APIs and analyzed a limited number of RESTful APIs.

Moreover, Rodríguez *et al.*[35] analyze high volume of REST HTTP traffic, i.e. HTTP requests, to evaluate how well or bad REST developers implement their APIs in practice. The authors compare the wellness with theoretical Web engineering principles and guidelines. The authors rely on heuristics and metrics to measure the implementation quality by means of antipatterns. Results are evident with a gap between the theory and practice. However, the work focuses only on REST requests and not responses. Moreover, the linguistic design quality of REST requests/responses is out of the scope of their work.

2.4. *Discussion*

The analysis of the aforementioned studies allows us to identify some limitations:

(1) There exist several works that deal with the linguistic assessment or consistency of identifiers in the OO domain (e.g. Refs. 4 and 15), however, these approaches are not applicable to the domain of Web and to the RESTful APIs documentation.

(2) The guidelines (e.g. Refs. 24 and 28) for designing RESTful APIs in a proper RESTful manner proposed an exhaustive list of best and poor design guidelines but do not discuss how, in practice, they can be detected.

(3) The approaches in the domain of REST (e.g. Refs. 4 and 15 and Web APIs[2,18,22,23,31]) contributed to the automatic generation of richer and simpler representation and organization of RESTful resources and their underlying access methods, however, there is still a lack of methods and tools to assess the existing documentation of RESTful APIs.
(4) Except for DOLAR,[30] there is no empirical evidence on the linguistic quality of the RESTful APIs documentation. Empirical assessment and evidences are needed because the current textual and machine-unreadable documentations of RESTful APIs already hinder the consumption of those APIs[18,22,23,31]

In the following section, we describe 12 REST linguistic patterns and antipatterns from the literature with examples. Then, we propose an automatic approach for assessing the linguistic quality of REST URIs and their API documentation. The goal of assessing the documentations of RESTful APIs is to check the consistency between the API documentation and the resources they describe.

3. REST Linguistic Patterns and Antipatterns

The following subsections present the 12 linguistic patterns and antipatterns that we consider in this paper, which we extracted from existing literature.[6,13,24,39] We summarize their definitions and provide good and bad examples of each one.

3.1. *Contextualized versus contextless resource names*

URIs should be *contextual*, i.e. nodes in URIs should belong to semantically-related contexts. Thus, the *Contextless Resource Names* antipattern appears when URIs are composed of nodes that do not belong to a same semantic context.

Example. ✖`www.example.com/newspapers/players?id=123` is a ✖*Contextless Resource Names* antipattern because "newspapers" and "players" do not belong to same semantic context. ✔`www.example.com/newspapers/media/page?id=123` is a ✔*Contextual Resource Names* pattern because "soccer", "team", and "players" belong to a same semantic context.

Consequences. *Contextless Resource Names* do not provide a clear context for a request, which may mislead the API clients by decreasing the understandability of the API.[13]

3.2. *Hierarchical versus nonhierarchical nodes*

Each node forming a URI should be related hierarchically to its neighbor nodes. In contrast, *Nonhierarchical Nodes* is an antipattern that appears when at least one node in a URI is not related hierarchically to its neighbor nodes.

Example. ✖`www.example.ca/profs/university/faculty/` is a ✖*Nonhierarchical Nodes* antipattern because "profs", "faculty", and "university are not in a hierarchical relationship. ✔`www.example.com/university/faculty/professors/` is a

✓*Hierarchical Nodes* pattern because "university", "faculty", and "professors" are in a hierarchical relationship.

Consequences. Using nonhierarchical names may confuse users on the real purpose of the API and hinders its understandability and, therefore, its usability.[13]

3.3. *Tidy versus Amorphous URIs*

REST resource URIs should be tidy and easy to read. A *Tidy URI* is a URI with appropriate lower-case resource naming, no extensions, underscores, or trailing slashes. The *Amorphous URI* antipattern appears when URIs contain symbols or capital letters that make them difficult to read and use. As opposed to good practices,[24] a URI is amorphous if it contains: (1) upper-case letter (except for Camel Cases[26]), (2) file extensions, (3) underscores, and (4) a final trailing-slash.

Example. ✗`www.example.com/NEW_Customer/_photo01.jpg/` is an ✗*Amorphous URI* antipattern because it includes a file extension, upper-case resource names, and underscores. ✓`www.example.com/customers/1234` is a ✓*Tidy URI* pattern because it only contains lower-case resource naming, without extensions, underscores, or trailing slashes.

Consequences. (1) Upper/lower-case names may refer to different resources, RFC 3986.[6] (2) File extensions in URIs violate RFC 3986 and affect service evolution. (3) Underscores are hidden when highlighting URIs, decreasing readability. (4) Trailing-slashes mislead users to provide more resources.

3.4. *Verbless versus CRUDy URIs*

Appropriate HTTP methods, e.g. GET, POST, PUT, or DELETE, should be used in *Verbless URIs* instead of using CRUDy terms (e.g. create, read, update, delete, or their synonyms).[13] The use of such terms as resource names or requested actions is highly discouraged.[24]

Example. ✗`POST www.example.com/update/players/age?id=123` is a ✗*CRUDy URIs* antipattern because it contains a CRUDy term "update" while updating the user's profile color relying on an HTTP POST method. ✓`POST www.example.com/players/age?id=123` is a ✓*Verbless URIs* pattern because this is an HTTP POST request without any verb.

Consequences. Using CRUDy terms in URIs can be confusing for API clients, i.e. in the best cases they overload the HTTP methods and in the worst cases they go against HTTP methods. CRUDy terms in a URI confuse and prohibit users to use proper HTTP methods in a certain context and may introduce another REST antipattern, *Tunneling through GET/POST*.[39]

3.5. *Singularized versus pluralized nodes*

URIs should use singular/plural nouns consistently for resources naming across an API. When clients send PUT/DELETE requests, the last node of the request URI should be singular. In contrast, for POST requests, the last node should be plural. Therefore, the *Pluralized Nodes* antipattern appears when plural names are used for PUT/DELETE requests or singular names are used for POST requests. GET requests are not affected by this antipattern.[13]

Example. The first example URI is a POST method that does not use a pluralized resource, thus leading to ✗*Pluralized Nodes* antipattern. On the other hand, for the ✓*Singularized Nodes* pattern, the DELETE request acts on a single resource for deleting it:

✗`DELETE www.example.com/team/players` or

✗`POST www.example.com/team/player`;

✓`DELETE www.example.com/team/player` or
✓`POST www.example.com/team/players`.

Consequences. If a plural node for PUT (or DELETE) request is used at the end of a URI, the API clients cannot create (or delete) a collection of resources, which may result in, for example, a `403 Forbidden` server response. In addition, even if the resources can be filtered through query-like parameters, it confuses the user if one or multiple resources are being accessed/deleted.[13]

3.6. *Pertinent versus nonpertinent documentation*

The *Nonpertinent Documentation* linguistic antipattern occurs when the documentation of a REST resource URI is in contradiction with its structure (e.g. nodes separated by slashes in URIs). This antipattern applies to both a resource URI and its corresponding documentation. In contrast, a well-documented URI should properly and clearly describe its purpose using semantically-related terms.

Example. ✗`https://api.twitter.com/1.1/favorites/list` — Returns the 20 most recent Tweets liked by the authenticating or specified user. Note: The like action was known as favorite before 3, November 2015; the historical naming remains in API methods and object properties. This URI–documentation pair from `Twitter` shows no semantic similarity between them and, thus, appears as ✗*Nonpertinent Documentation* antipattern. ✓`https://instagram.com/media/media-id/comments` — Gets a list of recent comments on a media object. The public_content permission scope is required to get comments for a media that does not belong to the owner of the access_token. In contrast to the previous example, this URI–documentation pair shows a high relatedness and is a ✓*Pertinent Documentation* pattern.

Consequences. Developers can make wrong assumptions on resource URIs, which may hinder their understandability and reusability. Overall, this is misleading because the client developer might be unsure whether to follow the documentation or the URI structure when trying to understand its purpose. Moreover, for the API providers, it might cause comprehension difficulties during the maintenance and evolution of APIs.

4. Background

This section introduces the second-order semantic similarity[16,17] and the LDA[8] that are useful in our SARA approach as we propose in Sec. 5.

4.1. *Second-order semantic similarity*

Second-order distributional similarity[16,17] metric allows obtaining distributionally the most similar words for an input word and computes similarity scores among them based on the second-order word vectors. Two words are distributionally similar if they have multiple co-occurring words in the same syntactic relations, in contrast to the distributional relatedness metric that uses a *bag-of-words* to capture these distributional relations.[9] Taking as input a big corpus, for each word, it builds a vector of the collocated words that appear together within a window of *window-size*. To compare the similarity between two words, the vectors of collated words are analyzed, calculating the extent to which those two words appear in the corpus together with the same collated words.

As shown in Table 1, for example, let us analyze the occurrences of the word *newspaper*. With $window\_size = \pm 3$, we have two occurrences of the word *newspaper*. If we take into account the position, *newspaper* has eight different features (by omitting stop words, e.g. "are", "the", "and") as shown in Table 2 in the WPT column. If we do not take into account the window position, then the word *newspaper* has seven different features (after omitting stop words), as shown in Table 2 under

Table 1. Window setup for the calculation of Window Position Triples (WPT).

−3	−2	−1	—	+1	+2	+3
radio	news	and	newspapers	are	more	accurate
TV	radio	and	newspapers	are	the	most

Table 2. Examples of the WPT and co-occurrences for the example in Table 1.

WPT		Co-occurrence	
$\langle newspaper, -3, radio\rangle$	1	$\langle newspaper, radio\rangle$	2
$\langle newspaper, -3, TV\rangle$	1	$\langle newspaper, TV\rangle$	1
$\langle newspaper, -2, news\rangle$	1	$\langle newspaper, news\rangle$	1
$\langle newspaper, -2, radio\rangle$	1	$\langle newspaper, are\rangle$	1
$\langle newspaper, +1, are\rangle$	2	$\langle newspaper, more\rangle$	2
$\langle newspaper, +2, more\rangle$	1	$\langle newspaper, accurate\rangle$	1
$\langle newspaper, +3, accurate\rangle$	1	$\langle newspaper, most\rangle$	1
$\langle newspaper, +3, most\rangle$	1		

the "co-occurrences" column. Moving the window over the corpus gives the word vectors for each word. With these word vectors and by normalizing the counts,[17] the distributionally similar words used in the similar context are calculated. For example, if *newspaper* co-occurs with {*radio*, *TV*, *news*, and *print*} and the *media* co-occurs with the same three words, we will conclude that they are distributionally similar. However, the two words are distributionally similar does not necessarily mean that they appear together. Distributional similarity allows capturing the different senses that a word has, and, therefore, mixes up the different similar words for all the senses that a word might have.

This technique allows one to obtain the list of the n most similar words for a given input word. The list is then used as the *second-order* word vector for the given word, which contains the words that occur together in the similar contexts. Applying the same principle the technique allows comparing the *second-order* word vectors to compute the *second-order distributional similarity.*[16,17]

Distributional semantic similarity strategies allows one to go beyond *is-a* relationships between nouns and verbs as allowed by WordNet-based approaches[9,17] that only benefit from the synonym (warm–hot), meronym (car–wheel), and antonym (hot–cold) relations. Second-order similarity has been demonstrated to hold a higher correlation with semantic similarities derived from WordNet than the latent semantic analysis (LSA) and Web-based PMI-IR.[17]

4.2. *Latent Dirichlet allocation*

In machine learning and NLP, topic models are defined based on the idea that a document is a mixture of latent topics and each topic is characterized by a distribution over words.[8] The LDA is a generative probabilistic model of a corpus based on the idea of topic models.[8] The LDA allows extracting the different topic models from a corpus that can be used as a low-dimensional representation for the content of the set of documents constituting the corpus.

The dimensionality k for the Dirichlet distribution is assumed to be known and fixed, and should be provided as input to build the desired topic model. The LDA model, as opposed to many other clustering models that restrict a document to be associated with a single topic, allows a document to be associated with multiple topics, giving the probability of the document belonging to a particular topic.

The large vocabulary size inherent to the majority of document corpora is one of the drawbacks of topic models due to the problems of sparsity.[8] A new document, we want to classify, will contain unobserved words that did not appear in the documents of the training corpus. This problem together with the *bag-of-words* assumptions that allow words that should appear or be generated by the same topic to be allocated in several different topics[8] drives us to define a hybrid approach by combining LDA topic modeling to obtain the low-dimensional representation of the corpus and the distributional semantic similarity to measure the semantic similarity between the words.

In the next section, we describe the SARA approach that takes the advantage of second-order semantic similarity metric and LDA topic modeling technique to perform semantic analysis of RESTful APIs for the detection of linguistic (anti)patterns.

5. The SARA Approach

Now that we defined 12 linguistic patterns and antipatterns, we present an approach to identify their occurrences.

In this section, we describe the SARA approach for the analysis and detection of linguistic patterns and antipatterns in RESTful APIs. SARA takes advantage of a second-order semantic similarity metric and LDA topic modeling technique to detect linguistic patterns and antipatterns.

SARA consist of four steps (as shown in Fig. 1), which are briefly described below and presented in more details in the following subsections.

Step 1. (*Analysis of linguistic patterns and antipatterns*). This manual step consists in analyzing the description of REST linguistic patterns and antipatterns from the literature to identify the properties relevant to their detection. We use these relevant properties to define *algorithmic rules* for patterns and antipatterns.

Step 2. (*Implementation of interfaces and detection algorithms*). This manual step involves the implementation of detection algorithms for patterns and antipatterns based on the rules defined in step 1 and the service interfaces for RESTful APIs, which include the list of methods to be invoked.

Step 3. (*REST methods invocation*). This automatic step deals with the consumption of RESTful APIs by calling their methods to access their underlying methods automatically from the interfaces defined in step 2.

Step 4. (*Detection of linguistic patterns and antipatterns*). The last automatic step deals with the semantic analysis of resource URIs and API documentations by applying automatically the detection algorithms (implemented in step 2) on resource URIs and documentations of RESTful APIs obtained in step 3 for the detection of linguistic patterns and antipatterns.

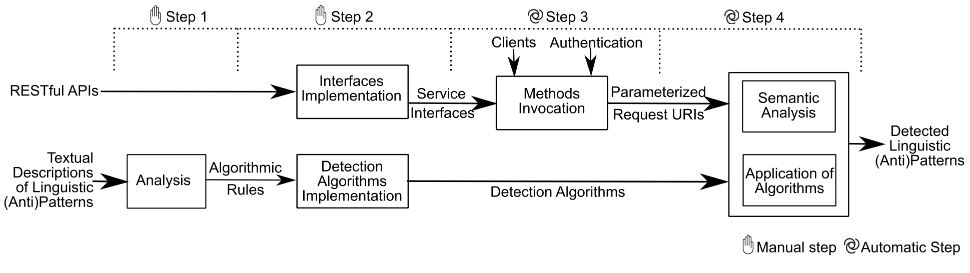

Step 1
Step 2
Step 3
Step 4
Clients
Authentication
RESTful APIs
Interfaces Implementation
Service Interfaces
Methods Invocation
Parameterized Request URIs
Semantic Analysis
Application of Algorithms
Detected Linguistic (Anti)Patterns
Textual Descriptions of Linguistic (Anti)Patterns
Analysis
Algorithmic Rules
Detection Algorithms Implementation
Detection Algorithms
Manual step
Automatic Step

Fig. 1. Overview of the SARA approach.

5.1. *Step 1 — Analysis of linguistic patterns and antipatterns*

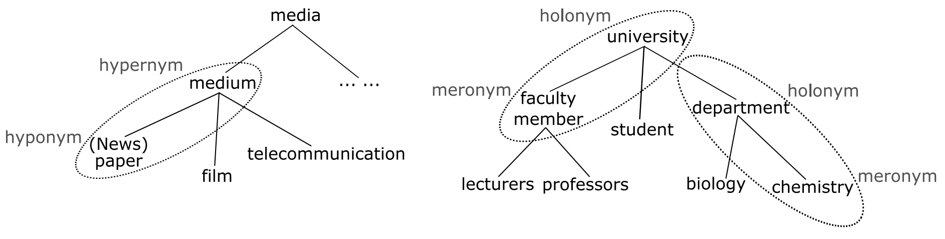
We analyze the definitions of the patterns and antipatterns defined in Sec. 3 to identify their linguistic aspects. For example, a linguistic aspect for the detection of the *Contextless Resource Names* antipattern is to check if a URI node belongs to the same semantic context.

Figure 2 shows the algorithmic rule that we define for the *Contextless Resource Names* antipattern based on WordNet. We compare the context of every pair of nodes or resources in a URI using WordNet (lines 3–6). We report a URI as an occurrence of this antipattern if we find at least one contextless relation among all possible resource pairs. Conversely, we report an occurrence of the corresponding pattern *if and only if* all possible resource pairs share at least one common context and are relevant for that particular URI.

WordNet is a widely-used lexical database, which groups nouns, verbs, and adjectives into sets of cognitive synonyms — *synsets* — each representing unique concepts that can be used interchangeably in a context. WordNet is useful in finding semantic similarity between words using its underlying *hypernym–hyponym* and *meronym–holonym* relations as depicted in Fig. 3. In Fig. 3(a), `medium` is one of 11 synsets of "media" and there exist different types of `medium` including `newspaper`, `film`, `telecommunication`, and so on, defined in WordNet. Based on WordNet, `medium` is thus the *hypernym* of `newspaper` and `newspaper` is the *hyponym* of `medium`. Such relations also exhibit contextual relevance between words and can be useful for analyzing *Contextless Resource Names* antipattern[13] in URIs.

```
1: CONTEXTLESS-RESOURCE-NAMES(Request-URI)
2:     URINodes ← EXTRACT-URI-NODES(Request-URI)
3:     for each index = 1 to LENGTH(URINodes)-1
4:         Set1 ← CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index}$)
5:         Set2 ← CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index+1}$)
6:         if $Set1 \cap Set2 = \emptyset$
7:             return true
8:         end if
9:     end for
10:    return false
```

Fig. 2. Algorithmic rule of the *Contextless Resource Names* antipattern (using WordNet).



(a)Hypernym–Hyponym relation (b) Holonym–Meronym relation

Fig. 3. Hypernym–hyponym and *meronym–holonym* relations in WordNet.

In addition, there exist *part-of* relations, i.e. *holonym–meronym*, between words defined in WordNet [see Fig. 3(b)]. For example, a `university` consists of `faculty member`, `student`, and `department` and the `department` may include `biology` and `chemistry`. Thus, `university` is a *holonym* for `faculty member` and `faculty member` is a *meronym* of `university`. Such hierarchical relations defined in WordNet between words are useful in analyzing the *Nonhierarchical Nodes* antipattern.[13] Moreover, Stanford's CoreNLP annotates nodes (after splitting CamelCase nodes) with its underlying part-of-speech (POS) tagger to differentiate verbs (i.e. actions) and nouns (i.e. resources). We thus define algorithmic rules for 11 other linguistic patterns and antipatterns.

WordNet and Stanford's CoreNLP dictionaries are general-purpose English dictionaries and do not provide any domain-specific knowledge (i.e. they are not suited to identify domain-specific semantic relationships between concepts). We overcome such limitations of WordNet and Stanford's CoreNLP by using well-known LDA topic modeling[8] technique and a second-order semantic similarity metric.[16,17] The use of these two techniques helps us capture the proper context (from URIs documentation) and check semantic relatedness between words and/or identifiers.

Figure 4 shows the algorithmic rule that we defined for the *Contextless Resource Names* antipattern based on the LDA and a second-order similarity metric. With WordNet-based analysis, we perform pairwise similarity of URI nodes to check if they have any *hypernym–hyponym* and *meronym–holonym* relations. In contrast, with LDA-based analysis we do not rely on general relations; instead, we extract domain knowledge from the API documentation to check if the URI nodes are in the same context. To measure similarity between URI nodes, we rely on (DISCO)-based extracting DIStributionally similar words using CO-occurrences[16] second-order similarity metric. DISCO uses an English Wikipedia precomputed database of word similarities to compute the similarity. The database contains more than 420,000 words and uses a *window_size* of 3. We report a URI as an occurrence of this antipattern if we find the similarity value to be less than a predefined threshold. Conversely, we report an occurrence of the corresponding pattern for the similarity value being equal to or higher than that threshold. We set the *threshold* value as 0.3 to determine if two words are semantically related based on the second-order similarity metric. For DISCO, Kolb[17] showed the threshold value of 0.3 as a *gold standard* with a good accuracy in terms of semantic relatedness.

```
1: CONTEXTLESS-RESOURCE-NAMES(Request-URI, API-Documentation)
2:     TopicsModel ← EXTRACT-TOPICS(API-Documentation)
3:     URINodes ← EXTRACT-URI-NODES(Request-URI)
4:     Similarity-Value ← CALCULATE-SECOND-ORDER-SIMILARITY(URINodes, TopicsModel)
5:     if Similarity-Value < threshold
6:         return true
7:     end if
8:     return false
```

Fig. 4. Algorithmic rule of the *Contextless Resource Names* antipattern (using LDA topic modeling and second-order similarity metric).

5.2. Step 2 — Implementing interfaces and detection algorithms

This step includes the implementation of the interfaces of the services to be assessed and the implementation of detection algorithms of linguistic patterns and antipatterns. For each RESTful API to be assessed, we implemented its service interface using Java, which contains the methods callable to access or modify its underlying resources. Each interface method is mapped to a HTTP method. Using the appropriate HTTP methods, our DOLAR approach sends HTTP requests to the real RESTful APIs and receives HTTP responses. Linguistic patterns and antipatterns, for example, *Amorphous URIs* (or *Tidy URIs*), require the fully-parameterized request URIs to be detected, which can only be obtained after HTTP requests are made. For each RESTful API, the details required to implement its service interfaces (i.e. resources, HTTP actions to perform on its resources, and parameters for each HTTP request) can be found in its online documentation as shown in Table 5. For other linguistic patterns and antipatterns, we can perform the analysis just with the URIs extracted from the documentation of the RESTful APIs.

Like the RESTful APIs interfaces, the detection algorithms for linguistic patterns and antipatterns are also written in Java. We manually transform the algorithmic rules defined in the previous subsection into executable programs. This is because some of the detection algorithms can be defined as the combination of some building blocks (i.e. various segments of algorithmic rules) where the gluing of these blocks is currently done manually. However, in the future version of our approach, we will explore the use of a domain-specific language (DSL) and also try obtaining through model-to-text transformations automatically the executable code of the algorithmic rules.

5.3. Step 3 — REST methods invocation

For each RESTful API, besides its interface, we also implement a client to call the methods in its interface, methods that perform read, write, update, or delete operations on resources. These explicit calls are done at detection time to obtain fully-parameterized request URIs sent to the servers, which are required for detecting antipatterns like *Amorphous URI*. In REST, a resource may be related to multiple Java methods because any of the four basic operations (GET, POST, PUT, and DELETE) can be performed. As for the clients' authentication, large companies often require clients' authentication to accept secured HTTP requests. Thus, we also implement the *OAuth* 2.0 authentication protocol. This step results in sets of parameterized request URIs and their responses for the APIs being assessed.

5.4. Step 4 — Detection of linguistic patterns and antipatterns

After having defined the algorithmic rules for antipatterns, implemented the interface for the RESTful APIs, derived the detection algorithms for antipatterns, and concretely invoked the REST services using resource URIs, we can now perform

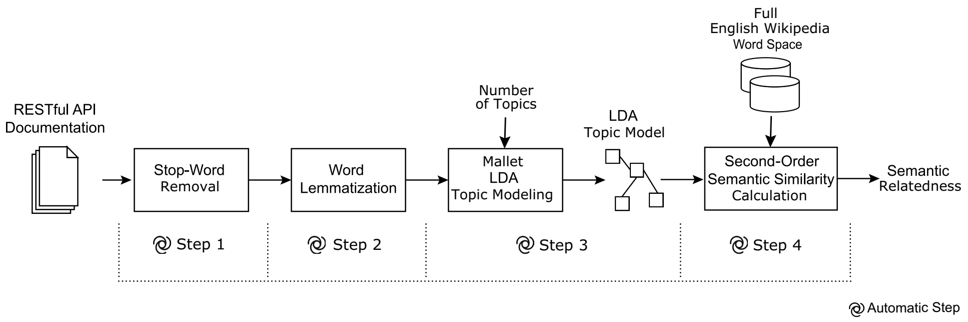


Fig. 5. The semantic analysis part of SARA approach.

the detection of the linguistic antipatterns on those resource URIs, which involves two phases (Fig. 1, step 4). The first phase involves performing the semantic analysis of the URIs and their corresponding documentation, whereas the second phase involves applying detection algorithms using the semantic relatedness (e.g. similarity values) determined from the semantic analysis. We discuss these two phases in the following sub-subsections.

5.4.1. *Semantic analysis of RESTful APIs*

The semantic analysis of REST URIs and documentation involves four automatic steps (as shown in Fig. 5): (1) collection of APIs documentation corpuses and performing some preprocessing, e.g. exclusion of stop words; (2) truncating URI nodes to their base form, i.e. lemmatization, using Stanford's CoreNLP; (3) based on the collected corpora, performing topic model extraction relying on LDA; and (4) measuring the second-order similarity between the acquired topic model and nodes in a URI.

In the following, we illustrate the semantic analysis showing the detection of our running example antipattern *Contextless Resource Names* (and, vice versa, *Contextual Resource Names* pattern) using LDA[8] topic modeling and second-order semantic similarity.[16,17]

Semantic analysis procedure. To infer the contextual relationships between resource identifiers, we rely on the Mallet LDA topic modeling tool-set.[c] LDA builds a model for a given text corpora that represents a short description of the members of the corpus and preserves the essential relationships critical for classification and/or summarization.[8] The LDA topic model therefore is a low-dimensional representation of the content of a set of documents.[38] We use Mallet for building a topic model taking the descriptions of the RESTful APIs resources as input, which exclude the list of parameters, response formats, and example code snippets.

[c]http://mallet.cs.umass.edu.

Table 3. Top 10 words for `Twitter` topic model with $k = 18$ (we only show the first five topic sets).

Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
list	search	user	profile	user
member	place	authenticate	user	id
add	save	block	banner	collection
remove	create	follow	image	cursor
create	query	post	account	follow
note	geo	friendship	update	order
rapidly	request	account	url	request
post	match	set	upload	navigate
membership	location	relationship	post	information
user	longitude	screen	authenticate	give

We first remove the stop words and extend the acronyms by using API-specific acronym lists. Then, we perform the lemmatization process to obtain the base form of each word by using Stanford's CoreNLP. We derive the LDA topic model accumulating k topics using Mallet. The total number of unique end-points for an API is the k number of topics in our LDA topic model. These unique end-points are the group of most significant concepts on the API as they are on the top of the URI hierarchy, if we consider the hierarchical organization of URIs as suggested in Ref. 13.

Table 3 shows an excerpt of the LDA topic model created using Mallet for the whole `Twitter` documentation corpus. The complete topic model consists of 18 topics and we consider the 10 most relevant words in each topic. This set of topics derived from an API corpus can be used to measure similarity between resource identifiers or words, i.e. if two words are semantically related, then they appear in the same topic.[38]

To calculate the semantic similarity between identifiers, we use the second-order semantic similarity metric. We rely on the distributional second-order similarity because the nodes in the URI can be slightly different from the text used for their description in terms of structure and form. A pair of words is said to be distributionally similar if it has common co-occurring words (i.e. words that appear frequently with the same set of words as neighbors). As described in Section 4.1, the calculation of this distributional similarity is based on a corpus, which we analyze to find the words that occur together within a context of $\pm$*window_size* words. The resulting word matrix is then processed to build word vectors that represent the distribution of a word in the corpus and show the words sharing a maximum number of co-occurrences. These vectors are used to compare two words by analyzing the extent to which these two words have similar second-order word vectors.[17] We use DISCO[16] to compute such distributional similarity between identifiers.

Determining antipatterns. To compare the contexts of every pair of nodes in a URI, we measure the second-order semantic similarity between each node in a URI and the 10 top words of each topic. Based on the similarity value, we determine

Table 4. An example analysis of two `Twitter` URIs.

Topic 0	lists/memberships.json?...		Topic 2	account/settings.json	
	Lists	Memberships		Account	Settings
list	**2.000**	0.113	user	0.109	0.217
member	0.233	0.752	authenticate	0.118	0.066
add	0.118	0.007	block	0.013	0.156
remove	0.085	0.000	follow	0.104	0.034
create	0.234	0.043	post	0.080	0.043
note	0.440	0.024	friendship	0.190	0.224
rapidly	0.013	0.037	account	**2.000**	0.666
post	0.165	0.135	set	0.158	0.372
membership	0.113	**2.000**	relationship	0.641	**0.799**
user	0.063	0.015	screen	0.038	0.192
Average similarity: 2.000			**Average similarity: 1.399**		

to which topics a node belongs. We consider that a node belongs to a topic if the average second-order semantic similarity value is greater than a predefined *threshold*, i.e. 0.3, for any words in each topic.

If for a given nodes pair of a URI, the intersection of topic sets to which each node belongs to is empty (i.e. there is no common topic for that pair of nodes in the URI), then the URI is reported as a *Contextless Resource Names* antipattern. Otherwise, if each pair of nodes in the URI share at least one common topic, then the URI is reported as a *Contextual Resource Names* pattern.

Table 4 shows the results for two resource URIs from the `Twitter` API: (1) `https://api.twitter.com/1.1/lists/memberships.json` and (2) `https://api.twitter.com/1.1/account/settings.json`. For the first URI, the base forms of each node (i.e. `membership` and `list`) both appear in topic 0 and are representative enough in `Twitter` API. Hence, we can report the first URI as *Contextual Resource Names*. For the second URI, the word `account` appears in topic 2 but `settings` does not appear in the list of the 10 most relevant words for the same topic. However, the word `settings` is semantically related to the word `relationship` with a second-order similarity metric value equal to 0.799 (greater than the *threshold* of 0.3). Therefore, we can report the second URI as *Contextual Resource Names* pattern because both nodes are semantically related.

5.4.2. *Application of detection algorithms*

The SOFA framework[27] automatically applies the algorithmic rules in the form of detection algorithms on the parameterized request URIs from the clients, collected in the previous step. Finally, SOFA returns a set of detected RESTful linguistic patterns and antipatterns.

The SOFA framework uses a Service Component Architecture (SCA).[10] It relies on FraSCAti[37] for its runtime support. In a previous work,[29] we added 13 REST patterns and antipatterns related to the design of RESTful requests/responses. Then, we extended SOFA with detection support of RESTful linguistic patterns and antipatterns using linguistic analyses based on WordNet and Stanford's CoreNLP.[30] For this paper, we further extend SOFA for the semantic analyses of RESTful resource URIs and their documentations for the detection of REST linguistic patterns and antipatterns relying on LDA topic modeling technique[8] and DISCO-based second-order semantic similarity metric.

Specifically, we extended the `REST Handler` component to facilitate the detection of RESTful linguistic patterns and antipatterns by wrapping each RESTful API in an SCA component and applying the detection algorithms on the SCA-wrapped RESTful APIs. By wrapping each API, we can introspect each full request URI with its actual runtime parameters, relying on *FraSCAti IntentHandler*, a runtime interceptor. We invoke methods from a service interface defined with an *IntentHandler* to introspect the request details, which allows on-the-fly syntactic and semantic analyses of parameterized request URIs and their corresponding documentations. Figure 6 outlines the SOFA framework positioning the end-user who executes the framework to perform the detection.

6. Validation

In this section, we assess SARA's effectiveness by analyzing the accuracy of the defined algorithmic rules, the extensibility of SOFA, and the performance of the detection algorithms. In our previous work,[30] we reported the results of an empirical study aimed at assessing the linguistic quality (i.e. syntactic and semantic) of RESTful APIs relying on WordNet. In this paper, we perform the validation study over the same dataset (310 URIs from 15 APIs[30]). Then, we perform a comparison

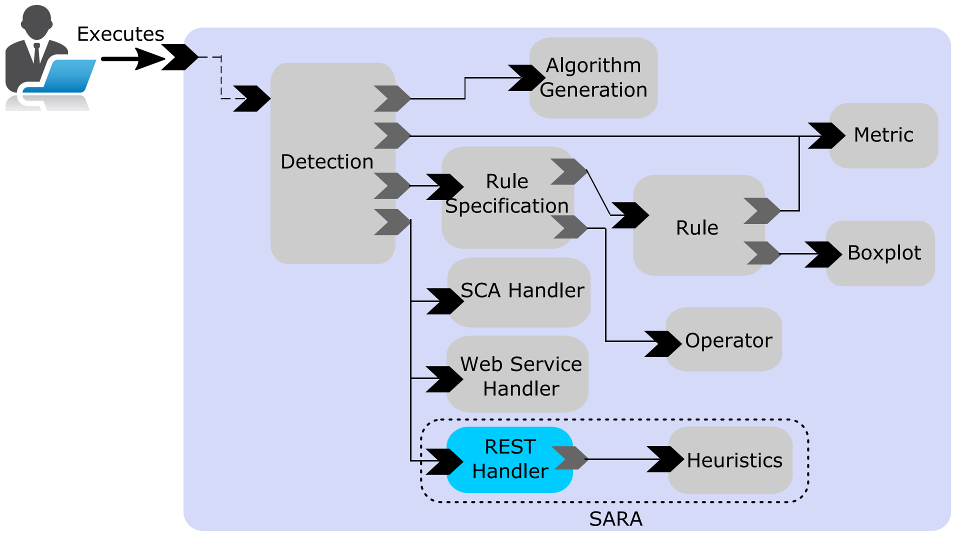


Fig. 6. The SOFA framework (the dotted box shows the position of SARA within SOFA).

of the accuracy, in terms of precision and recall, of SARA compared with our previous DOLAR approach. In addition, we perform an analysis of the 15 RESTful APIs adding also three new APIs to assess the semantic quality of RESTful APIs using the new LDA topic modeling and second-order similarity metric in addition to linguistic analyses as in our previous paper.[30]

6.1. *Hypotheses*

We formulate three hypotheses to assess SARA's effectiveness:

H_1 (Accuracy). *The set of all defined rules have an average precision and recall of more than 75%, i.e. at least three out of four are true positives and we do not miss more than one out of four of all existing patterns and antipatterns.*

H_2 (Extensibility). *Our SOFA framework is extensible and allows adding new service-oriented and REST patterns and antipatterns. In addition, SOFA facilitates the integration of new RESTful APIs.*

H_3 (Performance). *The implemented detection algorithms can perform their task with low detection times — namely, on average, in the order of seconds.*

6.2. *Subjects and objects*

The subjects of our study are the 12 REST linguistic patterns and antipatterns described in Sec. 3. The objects are 18 common and well-known RESTful APIs for which documentations are available. From our previous work[30] where we analyzed 15 APIs, in this paper, we added three new APIs, namely `GoogleBook`, `LinkedIn`, and `Walmart`. We choose APIs whose underlying HTTP methods, APIs end-points, and authentication mechanisms are well explained by RESTful API developers, e.g. `Facebook`, `Twitter`, `Dropbox`, or `YouTube`; these are summarized in Table 5.

6.3. *Validation process*

For the validation of SARA, we followed the instructions in the online documentations for the APIs and implemented related (authenticated) clients. We invoked a set of 310 REST methods from 15 RESTful APIs to access their resources. We collected all fully-parameterized request URIs from the clients and responses from the servers. Subsequently, we applied our algorithmic rules in the form of detection algorithms implemented manually on the REST request URIs and reported patterns and antipatterns detected by our SOFA framework. We validated the results in two phases: (1) all the `Dropbox` URIs and (2) four representative APIs — `Facebook`, `Twitter`, `Dropbox`, and `YouTube` — for which we randomly selected some candidate request URIs detected by DOLAR as patterns and antipatterns. We chose those four APIs based on our previous findings,[29] which concluded that `Facebook` and `YouTube` were well-designed APIs whereas `Twitter` and `Dropbox` were more problematic. We compared the accuracies between our previous DOLAR approach[30]

Table 5. List of the 18 analyzed RESTful APIs and their online documentations.

RESTful APIs	Online documentations
Alchemy	alchemyapi.com/api
BestBuy	developer.bestbuy.com/documentation
Bitly	dev.bitly.com/api.html
CharlieHarvey	charlieharvey.org.uk/about/api
Dropbox	dropbox.com/developers/core/docs
Externalip	api.externalip.net
Facebook	developers.facebook.com/docs/graph-api
GoogleBook	developers.google.com/books/
Instagram	instagram.com/developer
LinkedIn	developer.linkedin.com/docs
MusicGraph	developer.musicgraph.com/api-docs/overview
Ohloh	github.com/blackducksw/ohloh_api
StackExchange	api.stackexchange.com.docs
TeamViewer	integrate.teamviewer.com/en/develop/documentation
Twitter	dev.twitter.com/rest/public
Walmart	developer.walmartlabs.com/
YouTube	youtube.com/yt/dev/api-resources.html
Zappos	developer.zappos.com/docs/api-documentation

and SARA. In addition to the 310 REST methods tested in the DOLAR validation, we also included three new RESTful APIS. Moreover, for the newly added (anti)pattern *Pertinent versus Nonpertinent Documentation*, we needed the URIs' documentations. In total, we collected 555 REST resource URIs and their documentations to assess the documentations' pertinence.

Three professionals evaluated manually the URIs to identify the true positives and false negatives to define a ground truth for a predefined subset of the analyzed URIs and documentations. These professionals had knowledge about REST and were not involved in the detection steps. We provided them with the descriptions of the REST linguistic patterns and antipatterns, the sets of all URIs collected during the service invocations, and the documentation collected for each URI. We resolved conflicts with majority votes.

For assessing the accuracy, extensibility, and performance of DOLAR,[30] due to the large size of the datasets, we performed the validation on two sample sets because it was a laborious task to validate all APIs and all patterns and antipatterns but also because `Facebook`, `Dropbox`, `Twitter`, and `YouTube` are representative APIs.[29] In the first phase, we chose one medium-sized API, `Dropbox`, to calculate the recall on one API — the entire validation would have required 1,550 questions for 310 test methods. In the second phase, we randomly selected 50 validation questions (out of 630 possible candidates) to measure the overall accuracy.

We assessed SARA in two phases: (1) we compared the accuracy of SARA with DOLAR's accuracy[30] on the same dataset and (2) we selected a separate dataset by applying a similar technique described above for selecting test URIs (and their corresponding documentations) to measure SARA's accuracy on 18 RESTful APIs, including the three new APIs — namely, `GoogleBook`, `LinkedIn`, and `Walmart`.

We used precision and recall to measure the detection accuracy. Precision is the ratio between the true detected (anti)patterns and all detected (anti)patterns. Recall is the ratio between the true detected (anti)patterns and all existing true (anti)patterns.

6.4. *Interpretation of the results*

The mosaic plot in Fig. 7 represents the regenerated and improved detection results for each (anti)pattern on the 15 RESTful APIs using SARA. In our previous work,[30] we showed DOLAR detection results on the same dataset. Each column corresponds to a pattern and antipattern while rows represent the detected patterns and antipatterns on each API. In each row, the height of the mosaic represents the size of the method suite that we tested for an API. The most frequent patterns are *Verbless URI* and *Contextualized Resource Names* — the majority of the analyzed APIs did not include any CRUDy terms or any of their synonyms and the nodes in these URIs belong to the same semantic context. In contrast, the most frequent antipatterns are *Amorphous URI* and *Nonhierarchical Nodes* — the majority of the analyzed APIs involve at least one syntactical problem and URI nodes for those APIs were not organized in a hierarchical manner.

Table 6 presents the detailed detection results for the 10 linguistic patterns and antipatterns on 15 RESTful APIs. The table reports the patterns and antipatterns

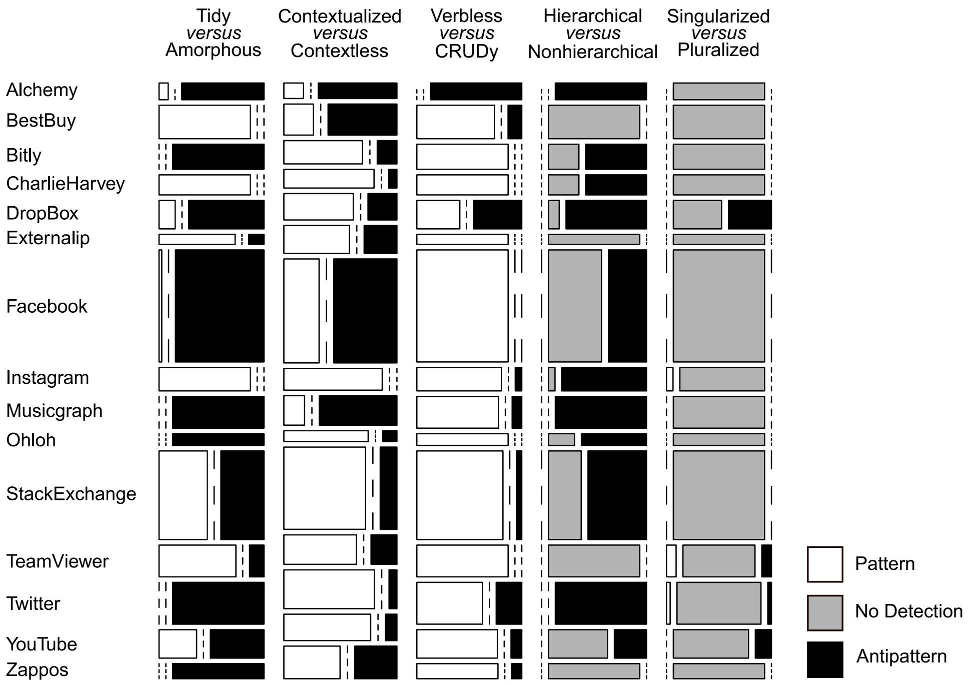


Fig. 7. Linguistic patterns and antipatterns detected in 15 RESTful APIs using SARA approach.

Table 6. Detection results of the 10 REST linguistic patterns and antipatterns (numbers in parenthesis show the number of methods tested for each API).

RESTful APIs	**(10)Alchemy**	**(20)Bestbuy**	**(15)Bitly**	**(12)CharlieHarvey**	**(17)DropBox**	**(6)Externalip**	**(67)Facebook**	**(14)Instagram**	**(19)Musicgraph**	**(7)Ohloh**	**(53)StackExchange**	**(19)TeamViewer**	**(25)Twitter**	**(17)YouTube**	**(9)Zappos**	**(310) Total**	**Detection Time(s)**
Linguistic Antipatterns/Patterns																	
✗Amorphous URI	9	0	15	0	14	1	65	0	19	7	25	3	25	10	9	**202**(65%)	0.984s
✓Tidy URI	1	20	0	12	3	5	2	14	0	0	28	16	0	7	0	**108**(35%)	0.968s
No Detection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	–
✗Contextless Resource Names	8	14	3	1	5	6	43	0	15	1	9	5	2	2	9	**123**(40%)	0.565s
✓Contextualized Resource Names	2	6	12	11	12	0	24	14	4	6	44	14	23	15	0	**187**(60%)	0.66s
No Detection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	–
✗CRUDy URI	10	3	0	0	9	0	0	1	2	0	3	0	7	2	1	**38**(12%)	0.737s
✓Verbless URI	0	17	15	12	8	6	67	13	17	7	50	19	18	15	8	**272**(88%)	0.677s
No Detection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**10**(3%)	–
✗Nonhierarchical Nodes	10	0	10	8	15	0	28	13	19	5	34	0	25	6	0	**173**(56%)	0.584s
✓Hierarchical Nodes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	0.592s
No Detection	0	20	5	4	2	6	39	1	0	2	19	19	0	11	9	**137**(44%)	–
✗Pluralized Nodes	0	0	0	0	8	0	0	0	0	0	0	2	1	3	0	**14**(5%)	0.668s
✓Singularized Nodes	0	0	0	0	0	0	0	1	0	0	0	2	1	0	0	**4**(1%)	0.656s
No Detection	10	20	15	12	9	6	67	13	19	7	53	15	23	14	9	**292**(94%)	–

in the first column followed by the analyzed RESTful APIs in the following 15 columns. For each RESTful API and for each pattern and antipattern, we report the total number of occurrences reported as positives by our detection algorithms. The last two columns show the total number of detected occurrences across the 15 APIs (with percentages) and the average detection time. The detailed detection results for all the 310 tested methods from 15 RESTful APIs are available on our project website: http://sofa.uqam.ca/sara/.

As shown in Table 6, more than 60% of the analyzed URIs (202 out of 310) show amorphousness. Exceptionally, `BestBuy` and `Instagram` have all the URIs detected as *Tidy URIs*. In contrast, all the URIs in `Twitter` have syntactic problems and all of them are detected as *Amorphous URIs*. Compared to DOLAR,[30] SARA better classifies the URIs (i.e. there are no gray boxes), which is made possible by using LDA-based topic modeling and DISCO-based second-order similarity metric measurement. There are significant numbers of APIs not using verbs in their URIs design, which, according to REST URI good design principles, is a good practice to avoid confusing API client developers.[39]

We observe detection results of inferior quality for *Hierarchical Nodes* pattern (i.e. the dictionaries could not find hierarchical relations among URI nodes). Indeed, we have zero detection for *Hierarchical Nodes* pattern because: (1) around 50% of tested URIs used only one node (excluding the base URI) in which case we cannot check the hierarchical relation and (2) more than 20% of URIs contain digits or numbers as nodes, which again do not fall under any hierarchical relations.

The occurrences of the *CRUDy URI* antipattern were detected in 12% (38 out of 310) tested URIs. In contrast, 88% (272 out of 310) of the tested URIs are *Verbless URI*. APIs designers seem aware of not mixing the definition of traditional Web service operations and resource-oriented HTTP requests in REST. In traditional Web services, operation identifiers reflect what they are doing whereas, in REST, actions to be performed on a resource should be explicitly mentioned only using HTTP methods and not within a URI through a CRUDy term. Finally, there is a significant amount of *No Detection* for *Singularized versus Pluralized Nodes* because about 90% of our tested requests used HTTP GET methods — such requests can retrieve both single and multiple resource instances. However, for the remaining 10%, the *Pluralized Nodes* antipattern appeared more frequently than the *Singularized Nodes* pattern.

Now, we discuss the *Contextless Resource Names* antipattern in detail, as it is our running example. Out of 310 tested URIs, 40% (123 occurrences) were identified as *Contextless Resource Names* antipatterns and 60% (187 occurrences) were detected as *Contextualized Resource Names* patterns. For example, in `BestBuy`, most of the URIs have a single node followed by parameters. We ignored parameters while we captured the context because they have variable values only, which are not the part of the base URI design. Thus, if there is only one node in a URI, it is not possible to capture meaningful contextual relationship. In DOLAR,[30] we faced this problem because we relied on WordNet to identify the context regardless of the underlying domain. However, SARA overcomes this limitation by creating API-specific topic models relying on LDA[8] and, then, by semantically matching the URI nodes with various topics. Therefore, in general, SARA can clearly distinguish between identifiers, and thus, in Table 6, the number of *No Detection* is zero for *Contextless versus Contextualized Resource Names*.

In contrast to the `BestBuy` API, the `Dropbox`, `Facebook`, `StackExchange`, `Twitter`, and `YouTube` APIs involve a high number of contextualized URIs naming. These good practices may help their APIs clients better understand and reuse them. For instance, the following snippet shows two request URIs from `Facebook` where the nodes in each URI are considered to be in the same semantic context:

```
1. https://graph.facebook.com/v2.2/{user_id1}/mutualfriends/{user_id2}?access_token=CAATt8...
2. https://graph.facebook.com/v2.2/{user_id1}/friendlists?access_token=CAATt8...
```

For `Facebook`, SARA reported 24 methods (out of 67) as *Contextualized Resource Names* patterns.

Figure 8 shows the comparison between DOLAR and SARA for the detection of *Contextualized versus Contextless Resource Names* on 15 RESTful APIs. The figure shows the detection improvement by SARA over DOLAR by classifying the APIs in gray boxes (Fig. 8, left). In general, a majority of the URIs from gray boxes (no conclusive detection) are moved to white boxes (pattern detected), i.e. LDA and DISCO-based analysis overcome the limitations of the WordNet English dictionary.

The mosaic plot in Fig. 9 shows the detection results for *Pertinent versus Nonpertinent Documentation* in 16 RESTful APIs — including three newly added APIs, e.g. `GoogleBook`, `LinkedIn`, and `Walmart`, but excluding `Externalip` and `Zappos` APIs since their documentations were not available online. The figure suggests that `Dropbox` and `StackExchange` document their resource URIs with higher pertinence compared to other APIs considered in this study. By contrast, `YouTube`, `GoogleBook`, and `Instagram` provide little documentation for their URIs and do not exhibit high pertinence between URIs and documentation.

Table 7 shows the detection summary for *Pertinent versus Nonpertinent Documentation* on 16 RESTful APIs. In total, we analyzed 555 URI–documentation pairs from those APIs of which 72% (400 out of 555) of tested URIs show pertinence with their corresponding documentations. By contrast, 28% (155 out of 555) of the tested URIs are not well described in their documentations according to SARA. This finding suggests major RESTful APIs developers, like `Dropbox`, `Facebook`, `StackExchange`, and `Twitter`, are concerned with documenting their APIs with consistency and expressiveness. However, SARA also reports comparatively a high number (28%, 155 out of 555) of nonpertinent documentations because LDA topic modeling underperforms (i.e. we had many false positives) for small corpus. Some URI documentations have only a single to a few lines of textual documentation, which is not enough for building a proper and correct topic modeling for any URI.[8]

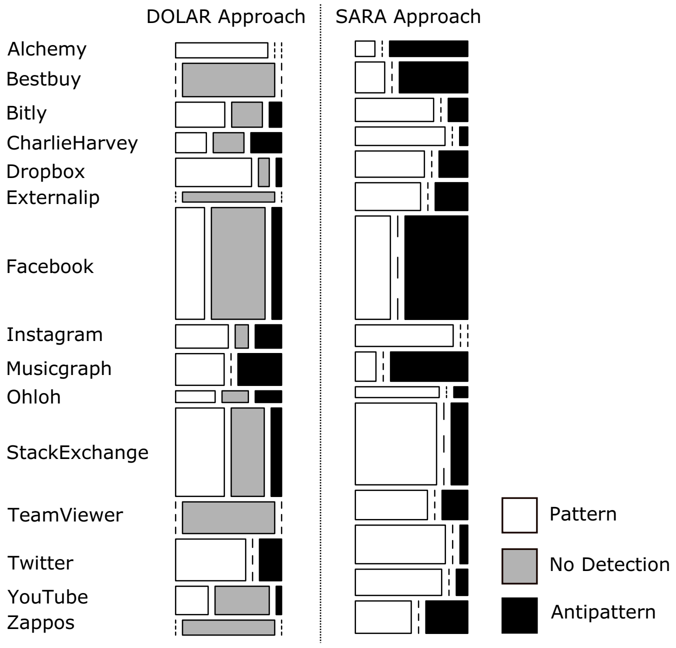


Fig. 8. Comparison between the DOLAR and SARA approaches for the detection of *Contextualized versus Contextless Resource Names.*

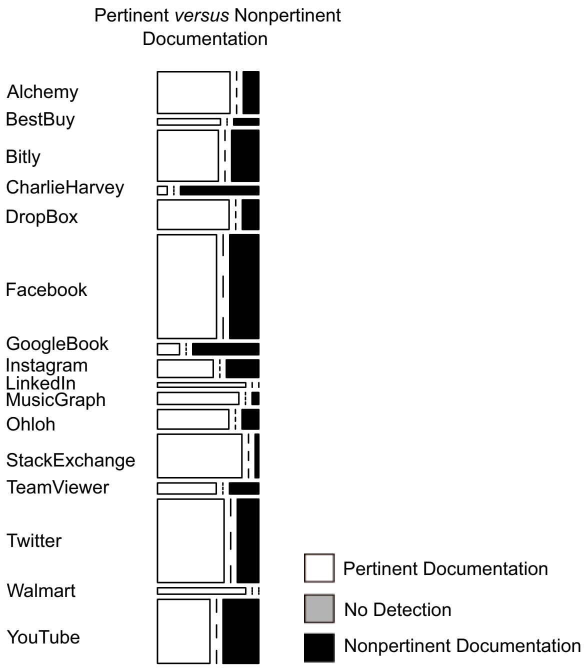


Fig. 9. Detection results for *Pertinent versus Nonpertinent Documentation* on 16 RESTful APIs.

Table 7. Detection results for *Pertinent versus Nonpertinent Documentation*: numbers in parenthesis are numbers of URI documentations analyzed for each API.

RESTful APIs	**(45)Alchemy**	**(7)Bestbuy**	**(55)Bitly**	**(9)CharlieHarvey**	**(32)DropBox**	**(112)Facebook**	**(12)GoogleBook**	**(19)Instagram**	**(5)LinkedIn**	**(13)Musicgraph**	**(21)Ohloh**	**(47)StackExchange**	**(12)TeamViewer**	**(90)Twitter**	**(7)Walmart**	**(69)YouTube**	**(555) Total**	**Detection Time(s)**
✗Nonpertinent Documentation	8	2	17	8	6	37	9	7	0	1	4	2	4	22	0	28	**155**(28%)	0.798s
✓Pertinent Documentation	37	5	38	1	26	75	3	12	5	12	17	45	8	68	7	41	**400**(72%)	0.899s
No Detection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	–

6.5. *Further discussion of the results*

In this subsection, we validate our detection results also in two phases. In the first phase, we compare the accuracy of SARA with DOLAR[30] on a complete API — namely, `DropBox` — by means of precision and recall (see Table 8, validation 1). In the second phase, we compare the overall precisions of DOLAR and SARA by randomly selecting a subset of all tested URIs (see Table 8, validation 2). In addition, we also perform the validation of detection results of newly defined antipattern on 16 RESTful APIs (see Table 9).

Table 8 shows the comparison of validation results on `Dropbox` (validation 1) and on four representative APIs (validation 2) between DOLAR and SARA. In the first validation for DOLAR, the average precision is 81.3% and recall is 78.0% for all patterns and antipatterns, whereas, for the same dataset, SARA has an average precision and recall of 80.9% and 81.0%, respectively. As for the second validation, the average precision for DOLAR is 79.7%, which is significantly improved by SARA with an average precision of 87.3%.

We illustrate with one example: in the first validation of `Dropbox` using DOLAR, two occurrences of *Verbless URI* are false positives. The terms `copy` and `search` (or their synonyms) were not considered CRUDy by our algorithm in `/1/copy_ref/dropbox/MyDropboxFolder/and/1/search/dropbox/MyDropboxFolder/`. However, the manual validation considered those terms CRUDy. Thus, on `Dropbox`, we had a precision of 100% and a recall of 75.0% for *CRUDy URI* and a precision of 80.0% and recall of 100% for *Verbless URI*. However, we improved the detection and excluded such CRUDy terms in SARA detection algorithms, which improved the detection precision of *Verbless URI* pattern by 20.0%.

The *Nonhierarchical Nodes* antipattern was detected by our detection algorithm in 14 cases whereas the manual validation suggested that only three of them actually are organized in a nonhierarchical order. We investigated the cause of such discrepancies and we found that the URIs identified as antipatterns by our algorithm were manually validated as patterns and have the following structure:

Table 8. Comparison of DOLAR and SARA validation results on `Dropbox` (validation 1) and partial validation results on `Facebook`, `Dropbox`, `Twitter`, and `YouTube` (validation 2). "P" represents the numbers of detected positives and "TP" the numbers of true positives.

Antipatterns/ Patterns	Validation 1													Validation 2								
		DOLAR						SARA							DOLAR				SARA			
	Validated	P	TP	Precision	Average Precision	Recall	Average Recall	P	TP	Precision	Average Precision	Recall	Average Recall	Validated	P	TP	Precision	Average Precision	P	TP	Precision	Average Precision
✗Amorphous URI	12	13	12	92.3%		100%		13	12	92.3%		100%		4	4	4	100%		4	4	100%	
✓Tidy URI	4	3	3	100%	96.2%	75%	87.5%	3	3	100%	96.2%	75%	87.5%	3	3	3	100%	100%	3	3	100%	100%
No Detection	0	0	0	-		-		0	0	-		-		0	0	0	-		0	0	-	
✗Contextless Resources	0	0	0	-		-		0	0	-		-		2	2	0	0%		1	1	100%	
✓Contextualized Resources	14	14	14	100%	100%	100%	100%	16	14	88%	88%	100%	100%	5	5	3	60%	53.3%	6	5	83%	91.7%
No Detection	2	2	2	100%		100%		0	0	-		-		3	3	3	100%		0	0	-	
✗CRUDy URI	8	6	6	100%		75%		8	8	100%		100%		2	2	2	100%		2	2	100%	
✓Verbless URI	8	10	8	80%	90%	100%	87.5%	8	8	100%	100%	100%	100%	9	9	9	100%	100%	9	9	100%	100%
No Detection	0	0	0	-		-		0	0	-		-		0	0	0	-		0	0	-	
✗Non hierarchical Nodes	3	14	3	21.4%		100%		14	3	21.4%		100%		3	6	1	16.7%		6	1	16.7%	
✓Hierarchical Nodes	11	0	0	-	60.71%	0%	66.7%	0	0	-	60.7%	0%	66.7%	11	0	0	-	58.3%	0	0	-	58.3%
No Detection	2	2	2	100%		100%		2	2	100%		100%		5	4	4	100%		4	4	100%	
✗Pluralized Nodes	4	6	3	50%		75%		6	3	50%		75%		4	1	1	100%		1	1	100%	
✓Singularized Nodes	2	0	0	-	60%	0%	48.3%	0	0	-	60%	0%	48.3%	6	1	1	100%	86.7%	1	1	100%	86.7%
No Detection	10	10	7	70%		70%		10	7	70%		70%		10	10	6	60%		10	6	60%	
Average				Precision	81.37%	Recall	78%			Precision	80.9%	Recall	81%				Precision	79.7%			Precision	87.3%

Table 9. Validation of *Pertinent versus Nonpertinent Documentation* and *Contextual versus Contextless Resource Names* using SARA.

Antipatterns/Patterns	SARA		Validated	Precision	Average Precision
	P	TP			
✗Nonpertinent Documentation	9	4	6	44.44%	65.97%
✓Pertinent Documentation	16	14	19	87.5%	
No Detection	0	0	0	-	
✗Contextless Resource Names	4	3	6	75%	80.36%
✓Contextualized Resource Names	21	18	19	85.71%	
No Detection	0	0	0	-	
Average Precision					**73.2%**

```
1. {baseURI}/{media|revisions|shares}/dropbox/MyDropboxFolder/...
2. {baseURI}/fileops/{copy|delete|move|create_folder}/?root=dropbox&path=...
```

Our dictionary-based analyses did not find any hierarchical relations between {`media,revisions,shares`} and {`dropbox`}, between {`MyDropboxFolder`} and {`dropbox`}, and so on. Yet, these hierarchical relations are obvious for developers and it was possible for the manual validation to infer the hierarchical relations among those pairs simply because they use a natural naming scheme.[19] Similarly, for the second example, `fileops` and {`copy,delete,move,create_folder`} are manually validated as being in a hierarchical relation while the English dictionaries could not find any hierarchical relations. Consequently, SARA considered them as *Nonhierarchical Nodes* antipattern. Therefore, for this antipattern, we had a low precision of 21.4%.

In the second validation, also for the *Nonhierarchical Nodes* antipattern, SARA faced a similar problem for `Twitter` as illustrated in these example URIs:

```
1. {baseURI}/help/privacy.json
2. {baseURI}/statuses/{show.json|user_timeline.json}?screen_name=...
```

The dictionary-based analyses did not find any hierarchical relations between `help` and `privacy` or between `statuses` and {`show,user,timeline`} and reported them as nonhierarchical. The precision for *Nonhierarchical Nodes* antipattern is therefore 16.7%.

Finally, an interesting observation for DOLAR from Table 8: it identified two occurrences of the *Contextless Resource Names* antipattern that were manually validated as *Contextualized Resource Names* pattern. Our investigation showed that the English dictionaries suggested `Canucks` and `albums` in `Facebook` and `followers` and `list` in `Twitter` to be in two different contexts. However, manual validation considered them as being in similar contexts, which drove the precision down to 0% for this antipattern in four representative APIs, with an average precision of 53.3%. SARA captures the context from URI's documentation based on LDA topic modeling and produces more precise context model. Therefore, it considers `Canucks` and `albums` in `Facebook` and `followers` and `list` in `Twitter` as contextual. With a significant improvement in capturing proper APIs context, SARA improves the detection of *Contextless versus Contextual Resource Names* by 38.4% (see Table 8, validation 2).

```
1. https://graph.facebook.com/Canucks/albums?access_token=CAA2...
2. https://api.twitter.com/1.1/followers/list.json?screen_name=...
```

More precisely, the summary of improvement in the detection accuracy from Table 8 is the following:

- Overall, for validation 1, SARA improves the average recall by 3.0% while compromising the average precision by 0.5%.
- SARA improves the recall of *CRUDy URI* antipattern by 25.0% and the precision of *Verbless URI* by 20.0%, and, thus, the average precision and recall for *Verbless versus CRUDy URI* by 10.0% and 12.5%, respectively.
- For validation 2, SARA improves the detection accuracy for *Contextual versus Contextless Resource Names* by 38.3% i.e. from 53.3% to 91.7%.
- Overall, for validation 2, SARA improves the average precision by 7.6%.

Table 9 shows the validation results obtained for the newly defined *Pertinent Documentation* pattern and *Nonpertinent Documentation* antipattern. SARA obtained an average precision of 66.0% after we randomly selected 25 URI–documentation pairs and asked three professionals to validate our detection results. We sampled and revalidated SARA with 25 new sets of REST resource URIs to verify if the nodes in URIs pertained to the same semantic context. The validation result in Table 9 shows an average precision of 80.4% for *Contextual versus Contextless Resource Names*, which confirms that SARA can capture contexts and classify identifiers with high accuracy.

We briefly describe two additional SARA detection results: SARA identified https://api.remix.bestbuy.com/v1/products/mostViewed from `BestBuy` as *Pertinent Documentation* linguistic pattern. After manually investigating its documentation,[d] which clearly describes the purpose and use of the URI, we found this detection appropriate. The manual validation also confirms this detection, i.e. all three professionals agreed on this URI–documentation pertinence.

On the other hand, https://api.twitter.com/1.1/users/suggestions/:slug/members from `Twitter` was identified as *Nonpertinent Documentation* linguistic antipattern. Our investigation of its documentation[e] and the manual validation also

[d]https://api.remix.bestbuy.com/v1/products/mostViewed — The Trending Products endpoint returns top 10 products, by rank, based on customer views of the bestbuy.com product detail-page over a rolling 3-h time period. Trending growth is based on a comparison against the previous 3-h time period. You can also pull this same information by category or subcategory. For more information about identifying category ids please refer to our Categories API documentation. Note: Minimum of 50 page views/h required for inclusion. In addition, deep subcategories may not have enough user traffic to generate trending data.

[e]https://api.twitter.com/1.1/users/suggestions/:slug/members — accesses the users in a given category of the `Twitter`-suggested user list and returns their most recent status if they are not a protected user.

confirmed this detection because the documentation does not seem to relate with its corresponding URI. The detailed detection results with all 555 URI–documentation pairs are available on our site: http://sofa.uqam.ca/sara.

6.6. *Discussion on the hypotheses*

We now discuss the hypotheses stated in Sec. 6.1.

H_1 (Accuracy). Table 8 shows the results for the first DOLAR and SARA validation on `Dropbox` API (validation 1) where we obtained an average precision of 81.3% and recall of 78.0% for DOLAR and an average precision of 80.9% and recall of 81.0% for SARA. As for the second validation (validation 2), for DOLAR, on a partial set of tested methods on `Facebook`, `Dropbox`, `Twitter`, and `YouTube` (i.e. 50 out of 125 tested methods), we obtained an average precision of 79.7%, and using SARA we further improved the detection accuracy and obtained an average precision of 87.3%.

For the validation 2, we cannot compute recall because we validated only a part of all tested methods. For the manually validated subset of URIs, we had a lower precision ranging between 16.7% and 21.4% only for *Nonhierarchical Nodes* antipattern. However, on average, we have precision of 80.9% (validation 1) and 87.3% (validation 2) and a recall of 78.0% (validation 1), with which we can support our first hypothesis on accuracy.

Finally, for the validation in Table 9, we have an average precision of 73.2% for *Pertinent versus Nonpertinent Documentation* and *Contextual versus Noncontextual Resource Names* revalidation.

H_2 (Extensibility). In our previous work,[30] we added to SOFA 10 new REST linguistic patterns and antipatterns, which required semantic analyses for their detection. Currently, SOFA can detect a set of 25 REST patterns and antipatterns from both syntactic and semantic aspects. Furthermore, we added three new RESTful APIs (i.e. `Instagram`, `StackExchange`, and `Externalip`) and more than 190 new HTTP requests from Ref. 29. To add new patterns and antipatterns, one needs to implement and integrate their detection algorithms within SOFA. To add a new RESTful API, one must add its service interface, the underlying methods of the service, an authenticated client that can invoke these methods, and a wrapper SCA component, which specifies the bindings, base URI, and runtime properties. For this paper, we added three new APIs (namely, `GoogleBook`, `LinkedIn`, and `Walmart`) and two new linguistic patterns/antipatterns (namely, *Pertinent Documentation* pattern and *Nonpertinent Documentation* antipattern) following the same process. Therefore, it is possible to add new RESTful APIs and patterns and antipatterns, which supports our second hypothesis.

H_3 (Performance). The last column of Table 6 shows the detection times for each pattern and antipattern, ranging between 0.565 s and 0.984 s, with an average of 0.709 s. In fact, the total required time also includes the execution time, i.e. sending requests and receiving responses (ranges from 2.074 s to 20.656 s, with an average of 6.920 s). We performed our experiment on an Intel Core-i7 with a processor speed of 2.50 GHz and 8 GB of memory. The reported detection times are comparatively low — on average, 10% of the total required time. However, the total required time also depends on the number of tested methods for each API. Moreover, the average detection time for the new pattern and antipattern, i.e. *Pertinent Documentation* and *Nonpertinent Documentation*, is 0.850 s, which is again in the order of seconds. With such a low average detection time of 0.709 s and execution time of 6.920 s, we can positively support our third hypothesis on performance.

6.7. *Threats to validity*

To minimize the threat to the *external validity* of our results, we performed experiments on 18 well-known APIs by invoking over 300 methods and by analyzing the documentation of over 550 URIs.

In DOLAR, we used WordNet for lexical and semantic analyses of URIs. However, one major limitation of WordNet is that it does not include information on the semantic similarity between words. In addition, the number of defined relationships among words is limited and it lacks compound words. For example, we found URIs with compound resource identifiers that, when split, may cause loosing contextual information. We minimize the threat to the *internal validity* by employing LDA topic modeling technique for capturing proper domain-specific context and by using DISCO-based second-order similarity metric for measuring similarity between identifiers.

The detection results may deviate depending on the defined algorithmic rules of linguistic patterns and antipatterns. Engineers may have their own views and levels of expertise on REST linguistic patterns and antipatterns, which may affect the definition of algorithmic rules. We tried to minimize this threat to the *construct validity* by defining all rules after a thorough review of definitions in existing literature on REST linguistic patterns and antipatterns. We also involved three professionals in the validation of the results who decided the patterns or antipatterns on majority.

Finally, to minimize the threat to *reliability validity* — the possibility to replicate this study — we gathered the details to replicate this study, including the algorithmic rules and the client request URIs, on our website.

7. Conclusion and Future Work

REST client developers need to correctly understand RESTful APIs while designing and developing their own Web-based systems. Understandability and reusability are thus two major factors that APIs providers must consider when designing such APIs. This paper presented the SARA approach, an improved version of the state-of-the-art DOLAR approach.[30] SARA is supported by the SOFA framework[27] extended with syntactic and semantic analyses for the detection of linguistic patterns and antipatterns in RESTful APIs.

We applied SARA to specify 12 linguistic patterns and antipatterns. We validated the SARA detection strategy by analyzing 18 RESTful APIs after invoking 310 methods and assessing 555 URI documentations and showed its accuracy: (1) an average precision of 80.9% and recall of 81.0% on `Dropbox` and (2) an average precision of 87.3% for a partial validation on `Facebook`, `Dropbox`, `Twitter`, and `YouTube`. We also observed that, out of the 18 analyzed RESTful APIs, most of them involved syntactical URIs design problems and did not organize URI nodes in a hierarchical manner. However, the REST APIs designers, in general, use appropriate resource names fit for a context, and they do not use verbs in URIs, which is a good URI design practice in REST.

The observed accuracy for SARA in terms of precision and recall shows that it is a feasible technique for detecting linguistic patterns and antipatterns in RESTful APIs, which can help RESTful API developers and vendors when assessing the linguistic quality of their APIs. Although we did not discuss in this paper, the proposed approach also has broader applicability for facilitating the cataloging of services, at least in part, for the purposes of service discovery and service brokerage applications.

A further investigation is required on the hierarchical relations among the resources, for example, in the case of composite RESTful resources which might not necessarily exhibit semantically meaningful hierarchical relations. As future work, we want to improve SARA by building API-specific hierarchical models to properly capture and identify the hierarchical relationships among URI nodes. We also want to perform additional validation of SARA's results with RESTful APIs developers. Since SARA does not perform well for the *Non-pertinent Documentation* antipattern with small-sized documentations while creating topic models with small corpus size, we plan to improve its detection. Moreover, with a thorough empirical evidence, we plan to confirm the useful implications for well-designed naming.

Acknowledgments

The authors thank Charlie Faucheux for initiating this study. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de Recherche du Québec — Nature et Technologie (FRQNT) Research Grants.

References

1. S. L. Abebe, S. Haiduc, P. Tonella and A. Marcus, Lexicon bad smells in software, in *Proc. 2009 16th Working Conf. Reverse Engineering* (IEEE, 2009), pp. 95–99.
2. R. Alarcón and E. Wilde, RESTler: Crawling RESTful services, in *Proc. 19th Int. Conf. World Wide Web* (ACM, 2010), pp. 1051–1052.

3. V. Arnaoudova, M. Di Penta, G. Antoniol and Y. G. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in *Proc. 2013 17th European Conf. Software Maintenance and Reengineering* (IEEE, 2013), pp. 187–196.
4. V. Arnaoudova, M. Di Penta and G. Antoniol, Linguistic antipatterns: What they are and how developers perceive them, *Empir. Softw. Eng.* **21**(1) (2015) 104–158.
5. V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol and Y. G. Guéhéneuc, REPENT: Analyzing the nature of identifier renamings, *IEEE Trans. Softw. Eng.* **40**(5) (2014) 502–532.
6. T. Berners-Lee, R. T. Fielding and L. Masinter, Uniform Resource Identifier (URI): Generic syntax (2005), https://tools.ietf.org/html/rfc3986.
7. A. Bertolino, P. Inverardi, P. Pelliccione and M. Tivoli, Automatic synthesis of behavior protocols for composable web-services, in *Proc. 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. Foundations of Software Engineering* (ACM, 2009), pp. 141–150.
8. D. M. Blei, A. Y. Ng and M. I. Jordan, Latent Dirichlet allocation, *J. Mach. Learn. Res.* **3**(4–5) (2003) 993–1022.
9. A. Budanitsky and G. Hirst, Evaluating WordNet-based measures of lexical semantic relatedness, *Comput. Linguist.* **32**(1) (2006) 13–47.
10. M. Edwards, Service Component Architecture (SCA) (2011), http://www.oasis-opencsa.org/sca.
11. T. Erl, *Service-Oriented Architecture*: *Concepts*, *Technology and Design* (Pearson Education, Boston, 2005).
12. R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California, Irvine (2000).
13. T. Fredrich, RESTful service best practices: Recommendations for creating Web services (2012), http://www.restapitutorial.com/resources.html.
14. M. Hausenblas, On entities in the web of data, in *REST from Research to Practice*, eds. E. Wilde and C. Pautasso (Springer, 2011), pp. 425–440.
15. N. Khamis, R. Witte and J. Rilling, Automatic quality assessment of source code comments: The JavadocMiner, in *NLDB 2010*: *Natural Language Processing and Information Systems* (Springer, 2010), pp. 68–79.
16. P. Kolb, DISCO: A multilingual database of distributionally similar words, in *Proc. 9th Conf. Natural Language Processing* (*KONVENS-2008*), Berlin, Germany (2008), pp. 37–44.
17. P. Kolb, Experiments on the difference between semantic similarity and relatedness, in *Proc. 17th Nordic Conf. Computational Linguistics* (*NODALIDA 2009*), Odense, Denmark (2009).
18. J. Kopecky, K. Gomadam and T. Vitvar, hRESTS: An HTML microformat for describing RESTful Web services, in *Proc. IEEE/WIC/ACM Int. Conf. Web Intelligence and Intelligent Agent Technology*, Vol. 1 (IEEE, 2008), pp. 619–625.
19. K. Laitinen, Estimating understandability of software documents, *SIGSOFT Softw. Eng. Notes* **21**(4) (1996) 81–92.
20. D. Lawrie, C. Morrell, H. Feild and D. Binkley, Effective identifier names for comprehension and memory, *Innov. Syst. Softw. Eng.* **3**(4) (2007) 303–318.
21. M. Lu, X. Sun, S. Wang, D. Lo and Y. Duan, Query expansion via WordNet for effective code search, in *Proc. 22nd IEEE Int. Conf. Software Analysis*, *Evolution*, *and Reengineering*, Montreal, Canada (2009), pp. 545–549.

22. P. A. Ly, C. Pedrinaci and J. Domingue, Automated information extraction from web APIs documentation, in *Proc. 13th Int. Conf. Web Information System Engineering* (*WISE 2012*) (2012), pp. 497–511.
23. M. Maleshkova, C. Pedrinaci and J. Domingue, Supporting the creation of semantic RESTful service descriptions, in *Proc. 8th Int. Semantic Web Conf.* (*ISWC 2009*) (2009).
24. M. Masse, *REST API Design Rulebook*, O'Reilly and Associate Series (O'Reilly Media, 2011).
25. C. Mateos, J. M. Rodriguez and A. Zunino, A tool to improve code-first web services discoverability through text mining techniques, *Softw. — Pract. Exp.* **45**(7) (2015) 925–948.
26. Microsoft, Microsoft MSDN: Capitalization Styles (2016), https://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx.
27. N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y. G. Guéhéneuc, B. Baudry and J. M. Jézéquel, Specification and detection of SOA antipatterns, in *Proc. 10th Int. Conf. Service-Oriented Computing*, Lecture Notes in Computer Science, Vol. 7636 (Springer, 2012), pp. 1–16.
28. Open Mobile Alliance, *Guidelines for RESTful Network APIs* (OMA, 2012).
29. F. Palma, J. Dubois, N. Moha and Y. G. Guéhéneuc, Detection of REST patterns and antipatterns: A heuristics-based approach, in *Service-Oriented Computing*, eds. X. Franch, A. Ghose, G. Lewis and S. Bhiri, Lecture Notes in Computer Science, Vol. 8831 (Springer, Berlin, 2014), pp. 230–244.
30. F. Palma, J. Gonzalez-Huerta and N. Moha, Are RESTful APIs well-designed? Detection of their linguistic (anti)patterns, in *Proc. 13th Int. Conf. Service Oriented Computing*, Goa, India (Springer, 2015), pp. 171–187.
31. L. Panziera and F. D. Paoli, A framework for self-descriptive RESTful services, in *Proc. 22nd Int. Conf. World Wide Web* (IW3C2, 2013), pp. 1407–1414.
32. A. Parrish, Social network APIs: A revised lexical analysis (2010), http://www.decontextualize.com/2010/04/social-network-apis-a-revised-lexical-analysis/.
33. F. Petrillo, P. Merle, N. Moha and Y. G. Guéhéneuc, Are REST APIs for cloud computing well-designed? An exploratory study, in *Proc. Int. Conf. Service-Oriented Computing* (Springer International Publishing, 2016), pp. 157–170.
34. M. M. Rahman and R. K. Chanchal, TextRank based search term identification for software change tasks, in *Proc. 22nd IEEE Int. Conf. Software Analysis*, *Evolution, and Reengineering*, Montreal, Canada (2015), pp. 540–544.
35. C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali and G. Percannella, REST APIs: A large-scale analysis of compliance with principles and best practices, in *ICWE 2016*, Lecture Notes in Computer Science, Vol. 9671 (Springer International Pubishing, 2016), pp. 21–39.
36. J. M. Rodriguez, M. Crasso, A. Zunino and M. Campo, Improving web service descriptions for effective service discovery, *Sci. Comput. Program.* **75**(11) (2010) 1001–1021.
37. L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni and J. B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, *Softw. — Pract. Exp.* **42**(5) (2012) 559–583.
38. M. Steyvers and T. Griffith, Probabilistic topic models, in *Latent Semantic Analysis*: *A Road to Meaning*, eds. T. Landauer, D. McNamara, S. Dennis and W. Kintsch (Laurence Erlbaum, 2007).
39. S. Tilkov, REST antipatterns (2008), www.infoq.com/articles/rest-anti-patterns.

40. C. Treude, M. P. Robillard and B. Dagenais, Extracting development tasks to navigate software documentation, *IEEE Trans. Softw. Eng.* **41**(6) (2015) 565–581.
41. F. Wei, A. Barros and C. Ouyang, Deriving artefact-centric interfaces for overloaded web services, in *Proc. Int. Conf. Advanced Information Systems Engineering* (Springer, 2015), pp. 501–516.