ARTE: Automated Generation of Realistic Test Inputs for Web APIs

Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, *Member, IEEE,* José María García and Antonio Ruiz-Cortés, *Member, IEEE*

Abstract—Automated test case generation for web APIs is a thriving research topic, where test cases are frequently derived from the API specification. However, this process is only partially automated since testers are usually obliged to manually set meaningful valid test inputs for each input parameter. In this article, we present ARTE, an approach for the automated extraction of realistic test data for web APIs from knowledge bases like DBpedia. Specifically, ARTE leverages the specification of the API parameters to automatically search for realistic test inputs using natural language processing, search-based, and knowledge extraction techniques. ARTE has been integrated into RESTest, an open-source testing framework for RESTful APIs, fully automating the test case generation process. Evaluation results on 140 operations from 48 real-world web APIs show that ARTE can efficiently generate realistic test inputs for 64.9% of the target parameters, outperforming the state-of-the-art approach SAIGEN (31.8%). More importantly, ARTE supported the generation of over twice as many valid API calls (57.3%) as random generation (20%) and SAIGEN (26%), leading to a higher failure detection capability and uncovering several real-world bugs. These results show the potential of ARTE for enhancing existing web API testing tools, achieving an unprecedented level of automation.

Index Terms—Test data generation, automated testing, web APIs, Web of Data

1 Introduction

Web *Application Programming Interfaces (APIs)* allow heterogeneous software systems to talk to each other over the network [1], [2]. Modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style, being referred to as *RESTful* web APIs [3]. RESTful web APIs typically allow applications to interact by exchanging JSON messages sent over HTTP. In practice, this allows, for example, checking the result of a football match (BeSoccer API [4]), posting a tweet (Twitter API [5]), booking a hotel room (Amadeus API [6]), translating a text (DeepL API [7]), or finding a route between two locations (Openroute API [8]). RESTful APIs are commonly described using languages such as the OpenAPI Specification (OAS) [9]. An OAS document provides a structured specification of a RESTful web API that allows both humans and computers to discover and understand the capabilities of a service without requiring access to the source code or additional documentation. In what follows, we will use the terms RESTful web API, web API, or just API interchangeably.

Testing web APIs adequately requires using realistic test inputs such as country names, codes, coordinates, or addresses. As an example, the hotel search operation in the Amadeus API [6] requires users to provide valid hotel names (e.g., "`Hotel California`"), hotel chains (e.g., "`Hilton`"), IATA airport codes (e.g., "`BUE`" for Buenos Aires), ISO currency codes (e.g., "`EUR`" for Euro), and ISO language codes (e.g., "`FR`" for French), among others. Generating meaningful values for these types of parameters randomly is rarely feasible. Even if a test data generator could bypass the syntactic validation generating values with the right format, the chances of constructing API requests that return some results—and therefore exercise the core functionality of the API—would be remote. To address this issue, most test case generation approaches resort to *data dictionaries*: sets of input values collected by the testers, either manually [10] or, when possible, automatically [11]. This means a major obstacle for automation since data dictionaries must be created and maintained for each non-trivial input parameter, on each API under test. Other authors propose using the default or sample values included in the API specification as test inputs, if any, but those are solely intended to explain the behavior of the API, and they are insufficient to test it thoroughly [12].

Several authors have addressed the problem of generating realistic test inputs for desktop, web, and mobile apps using semantic knowledge discovery techniques [13], [14]. Specifically, they propose to query knowledge bases like DBpedia [15] to discover realistic test input values for the graphical user interface (GUI) elements of the application under test. For example, if the GUI includes a text field with the label "`DOI`", their approaches would search for "DOI" identifiers in DBpedia. To the best of our knowledge, this strategy—based on querying knowledge bases for realistic test inputs—has not been applied to the context of web APIs yet, despite its potential.

Automatically generating realistic test data for web APIs requires facing some unique challenges since, unlike GUIs, APIs are intended for developers, rather than for users. To start with, unlike GUI labels, API parameters may follow many different naming conventions. Hence, for example, different APIs could refer to the concept *country code* using very different parameters' names such as `country`

- *J.C. Alonso, A. Martin-Lopez, S. Segura, J.M. García, and A. Ruiz-Cortés are with the Smart Computer systems Research and Engineering Lab (SCORE) and the Research Institute of Informatics Engineering (I3US), Universidad de Sevilla, Spain. E-mail: {javalenzuela, alberto.martin, sergiosegura, josemgarcia, aruiz}@us.es*

(Asos API [16]), `countryCode` (DHL API [17]), `country_code` (Numverify API [18]), `cc` (Foursquare API [19]), `c`, or even a less explanatory term, such as `location` (Domainr API [20]) or `excludedCountryIds` (GeoDBCities API [21]). This may make it necessary to resort to the description of each parameter in the search for further information that helps to identify the key concept. However, descriptions are given in natural language and, as names, they can be very heterogeneous both in style and length. Finally, API specifications occasionally include further information such as expected patterns (i.e., regular expressions) and sample values. Exploiting all these aspects for the generation of realistic test data is the key challenge addressed in our work.

In this article, we present an approach for the Automated generation of Realistic TEst inputs (ARTE) for web APIs. Specifically, our approach focuses on RESTful APIs as the de facto standard, but it could be applied to any web API provided that there is an API specification. ARTE leverages semantic knowledge discovery for the generation of realistic test inputs. In particular, ARTE exploits the specification of the various elements of the API under test, such as the name and description of its parameters, by querying knowledge bases to automatically generate realistic test inputs. In contrast to related approaches, ARTE includes a novel step for the automated inference of regular expressions from previously generated inputs, increasing the accuracy of the semantic queries and the overall performance of the approach. Furthermore, ARTE has been integrated into RESTest [22], a state-of-the-art tool for black-box test case generation for RESTful APIs, making our approach fully automated and publicly available.

We evaluated the effectiveness of ARTE by comparing its performance with existing random techniques and the related tool SAIGEN—recently proposed in the context of automated testing of mobile apps [14]—in two scenarios: (1) on the generation of realistic test inputs for 48 web APIs; and (2) on the generation of valid API calls, API coverage, and the detection of failures in 6 industrial web APIs. Experimental results show that ARTE can generate meaningful valid inputs for 64.9% of the target API parameters (137 out of 211), outperforming SAIGEN (31.8%). More importantly, ARTE supported the generation of over twice as many valid API calls (57.3%) as random generation (20%) and SAIGEN (26%). As a result, ARTE achieved higher coverage and revealed more failures in more APIs, detecting confirmed bugs in the web APIs of Amadeus [6] and DHL [17] not detected by related techniques. These results show the potential of ARTE to enhance current specification-driven web API testing tools.

To summarize, after introducing the background on RESTful APIs and the Web of Data (Section 2), this paper provides the following original research and engineering contributions:

- ARTE, a novel approach for the automated extraction of realistic test inputs for web APIs from knowledge bases like DBPedia (Section 3).
- Integration of ARTE into the open-source testing framework RESTest, making our approach readily applicable in practice (Section 4).
- An empirical comparison of ARTE with random data generation techniques and the state-of-the-art approach SAIGEN [14] on the generation of realistic test inputs and its impact on testing real-world APIs (Section 5).
- A publicly available replication package [23] containing the source code and datasets discussed in the article. We trust that this will also serve as a benchmark for future contributions in the topic.

We discuss the threats to validity in Section 6, related work in Section 7, and conclude the paper in Section 8.

2 Background

This section introduces key concepts to contextualize our proposal, namely, RESTful APIs and the Web of Data.

2.1 RESTful APIs

Modern web APIs typically follow the REpresentational State Transfer (REST) [3] architectural style, being referred to as RESTful web APIs [1]. RESTful web APIs are usually decomposed into multiple RESTful web services [2], each of which implements one or more create, read, update, and delete (CRUD) operations on a resource (e.g., a playlist in the Spotify API [24]). These operations can be invoked by sending specific HTTP requests to specific API endpoints. For example, a POST HTTP request to the URI `https://api.spotify.com/v1/users/42/playlists` would create a playlist for the user with ID `42` in the Spotify API. RESTful APIs can be described in the OAS language [9], arguably the current industry standard. Listing 1 depicts an excerpt of the OAS specification of the DHL API [17]. As illustrated, an OAS document describes the API mainly in terms of the operations supported, as well as their input parameters and the possible responses. The operation shown in Listing 1 allows to search for DHL service point locations (lines 1-5) in JSON format (lines 6-7). The operation receives nine input parameters (lines 8-43). Successful responses should include a 200 status code and the set of results matching the input filters (lines 44-48). Note that it is possible to specify constraints such as regular expressions for strings and min/max values for numbers (line 27).

Motivated by their critical role in software integration, many researchers have addressed the challenge of automatically generating test cases for RESTful web APIs [11], [12], [22], [25], [26], [27], [28]. Most approaches in this domain follow a black-box strategy, where test cases are automatically derived from the API specification, typically in OAS format [11], [12], [22], [25], [27]. Roughly speaking, these approaches generate (pseudo-)random API calls by assigning values to the API input parameters, and then checking whether the API responses conform to the API specification. For the generation of test inputs, some authors resort to random values (fuzzing) [27] or default values [12], but these are rarely enough to test the APIs thoroughly. Hence, most approaches resort to data dictionaries: sets of predefined input values [10]. For example, we could create a list of valid postal codes from which to select test inputs for the parameter `postalCode` in DHL (Listing 1). However, creating and maintaining data dictionaries for each non-trivial input parameter is a costly manual endeavour—this is the problem that motivates our work.

```yaml
paths:
 '/find-by-address':
  get:
   operationId: findByAddress
   description: Find DHL locations based on an address.
   produces:
    - application/json
   parameters:
    - name: countryCode
      in: query
      description: 'A two-letter ISO 3166-1 alpha-2 code.'
      required: true
      type: string
      x-example: 'DE'
    - name: postalCode
      in: query
      description: 'Postal code for an address.'
      required: false
      type: string
      x-example: '53113'
    - name: limit
      in: query
      description: 'Maximum of results to return.'
      required: false
      type: number
      default: 15
      maximum: 50
      x-example: 20
    - name: locationType
      in: query
      description: 'Type of the DHL Service Point location.'
      required: false
      type: string
      enum:
        - servicepoint
        - locker
        - postoffice
        - postbank
    - name: streetAddress ...
    - name: serviceType ...
    - name: radius ...
    - name: addressLocality ...
    - name: providerType ...
   responses:
    '200':
      description: 'List of DHL Service Point locations.'
      schema:
       $ref: '#/definitions/supermodelIoLogisticsPUDOLocations'
```

Listing 1. OAS excerpt of the DHL API.

2.2 Web of Data

The Web of Data is a global data space in continuous growth that contains billions of interlinked queryable data published following the Linked Data principles [29]. According to these principles, resources are identified using Uniform Resource Identifiers (URIs) [30] and resource relationships are specified using the Resource Description Framework (RDF) [31]. RDF is a standard that specifies how to identify relationships between resources in the form of triples composed by a subject, a predicate, and an object, denoted as <*subject*, *predicate*, *object*>. The predicate specifies the relationship (or *link*) that holds between the subject and object entities. For example, the triple <http://dbpedia.org/resource/George_R._R._Martin, http://dbpedia.org/ontology/author, http://dbpedia.org/resource/A_Game_of_Thrones> indicates that George R.R. Martin (subject) is the author (predicate) of "A Game of Thrones" (object). Subjects and predicates are URIs representing the entities and link types, respectively. Objects can be either resource URIs or literals, i.e., data values.

SPARQL [32] is a query language aimed at performing queries to datasets represented as RDF triples. Knowledge bases [33] like DBpedia [15] and Wikidata [34] consist of RDF graphs where triples generated from various sources are interlinked and can be explored using SPARQL queries. Listing 2 shows a sample SPARQL query to search for book titles with their corresponding ISBN and number of pages. The clause `FILTER(condition)` is used to restrict the results to those satisfying the given Boolean condition. This clause can be used, for example, to obtain values that match a regular expression or arithmetic conditions (such as minimum or maximum values). In the example, only data belonging to entities that contain the target predicates (title, ISBN, and number pages) matching the regular expression (codes consisting of 5 groups of an undefined number of digits separated by '-') and with 100 or more pages will be returned.

```sparql
SELECT DISTINCT ?title ?pages ?isbn WHERE {
 ?subject <http://dbpedia.org/property/title> ?title ;
       <http://dbpedia.org/ontology/numberOfPages> ?pages ;
       <http://dbpedia.org/ontology/isbn> ?isbn .
 FILTER (?pages >= 100)
 FILTER regex(str(?isbn), '^([0-9]*[-| ]){4}[0-9]*$')
}
```

Listing 2. SPARQL query to search for book titles with their ISBN and number of pages.

3 ARTE

In this section, we present ARTE, an approach for the Automated generation of Realistic TEst inputs for web APIs. Specifically, ARTE leverages the information in the API specification to search for syntactically and semantically valid test data in knowledge bases like DBpedia. We define syntactically and semantically valid inputs as follows:

Definition 1 (Syntactically valid input)**.** An input value is *syntactically valid* if it satisfies the syntactic constraints defined in the API specification and it is accepted by the API under test without returning an error. For example, `"Germany"` is not a syntactically valid value for the parameter `countryCode` in the API of DHL (Listing 1) because, although it conforms to the specification (string value), the API only accepts two-letter ISO 3166-1 alpha-2 codes, as explained in the description of the parameter. Conversely, the value `"DE"` (ISO 3166-1 alpha-2 code for Germany) is syntactically valid.

Definition 2 (Semantically valid input)**.** An input value is *semantically valid* if it is coherent with the API domain. For example, `"Berlin"` is a semantically valid test input value for the parameter `addressLocality` in the API of DHL (Listing 1), whereas `"dog"` is not.

A value can be syntactically valid, but semantically invalid, and vice versa. For example, `"dog"` is a syntactically valid value for the parameter `addressLocality` in the API of DHL—it conforms to the specification (string value) and is processed by the API without errors—but it is not coherent with the semantics of the parameter. Conversely, `"DEU"` (ISO 3166-1 alpha-3 code for Germany) is a semantically valid value for the parameter `countryCode`, but it is not syntactically valid, since the API expects two-letter codes, i.e., `"DE"` for Germany.

In what follows, we describe the main steps of ARTE, highlighted in Figure 1. A more formal description of our approach using pseudocode is provided as supplemental material.

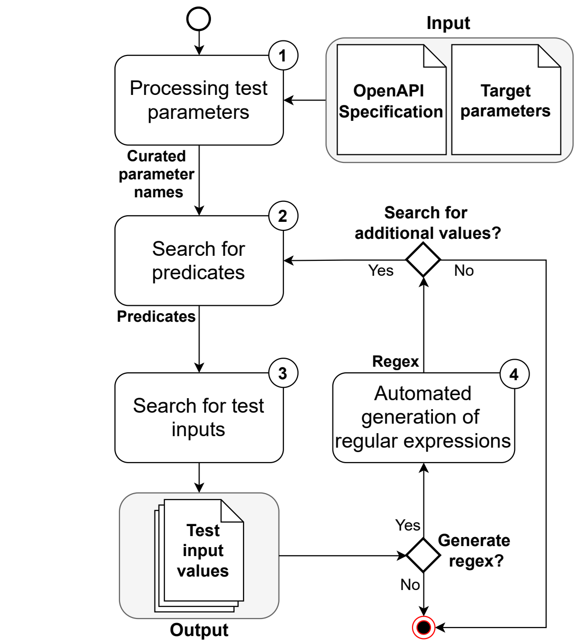

Fig. 1. Workflow of ARTE.

3.1 Processing test parameters

In this first step, the information of the test parameters is collected from the input API specification including their name, description and, if available, extra syntactic constraints like regular expressions and min/max values. As an additional input, the user must specify the list of parameters for which ARTE should generate meaningful data. This is helpful to exclude trivial parameters like dates, or domain-specific identifiers (e.g., a YouTube video ID), unlikely to be found in general-purpose knowledge bases like DBpedia. For such parameters, a different test data generation strategy may be used (e.g., data dictionaries).

Some popular APIs include parameters with a single character in their names. This is the case, for example, of the Open Movie Database (OMDb) API [35] and the dblp computer science bibliography API [36]. When this happens, it is very common that this single character is the first letter of the full name of the parameter, e.g., ‘t’ for the movie *title* in the OMDb API. Based on this idea, ARTE tries to infer the full parameter name from its description using natural language processing (NLP) techniques [37], [38]. After using part-of-speech tagging [39], stop words removal [40], and lemmatization [41], we are left with only the nouns included in the description. We observed that the full parameter name often matches the shortest noun from the list that begins with the same letter as the one-character original name of the parameter e.g., ‘q’ for search *query* in the DBLP search API [36] given the description “The query string to search for”. Thus, we used this as the default heuristic for selecting the full parameter name from the list of nouns. When multiple candidate words have the same length, ARTE selects the first one in alphabetical order, and if no candidate word is found, the letter is used as keyword. Implementing other heuristics would be straightforward.

3.2 Search for predicates

In this step, a user-defined knowledge database (e.g., DBpedia) is queried to obtain predicates that are representative of the target input parameters. For each target parameter, the search for predicates is performed in two iterative steps. First, a representative keyword of the target parameter is generated from the name and the description of the parameter, by applying the matching rules presented later on in this section. Second, for each candidate keyword, a SPARQL query is constructed as shown in Listing 3, where the string `keyword` is replaced by the selected keyword. This query conducts a search for predicates (line 2) that contain the provided keyword (filter by regular expression, line 3, where the flag `'i'` means “case-insensitive”) and ordered by length in ascending order (line 4).

```
1 SELECT DISTINCT ?predicate WHERE {
2    ?predicate a rdf:Property
3    FILTER regex(str(?predicate), 'keyword', 'i')
4 } ORDER BY strlen(str(?predicate))
```

Listing 3. SPARQL query used to search for predicates.

After executing the query, if successful, a list of predicates containing the keyword is returned. As an example, the following are a subset of the predicates found in DBpedia when using the keyword “currency”:

- `http://dbpedia.org/property/`**`currency`**
- `http://dbpedia.org/ontology/`**`currency`**
- `http://dbpedia.org/property/`**`currency`**`Iso`
- `http://dbpedia.org/ontology/`**`currency`**`Code`

Predicates are ordered by length in ascending order, with all of them containing the exact keyword. ARTE selects the first n predicates returned by the query (5 in our experiments), computing the support of each of them. Similarly to Mariani et al. [13], we define the *support* of a predicate as the number of unique RDF triples that contain it. The support of a predicate is obtained by running a SPARQL query as the one shown in Listing 2 including the COUNT set function (see supplemental material for an example). This query will include filters with the syntactic constraints of the API specification (regex and min/max values), if any, to exclude invalid values. A predicate is accepted only if its support is greater than a user-defined threshold (20 in our evaluation).

To identify relevant keywords, several *matching rules* are applied to the name and description of the parameters. These rules are ordered by priority, meaning that, as soon as a keyword is found with a rule, it is used to search for predicates. If a predicate is accepted (i.e., its support is greater than the configured threshold), the search for predicates for that parameter terminates, and the process starts over again for the next parameter, otherwise ARTE tries to identify new keywords with the remaining rules. In our preliminary experiments, we found that the description of the parameter, and not only its name, often includes key information for the generation of meaningful test inputs. Therefore, the matching rules implemented in ARTE exploit the description of the parameter first (rules 1-3 below) and then its name (rules 4-6).

In what follows, we describe the matching rules currently used in ARTE, ordered from highest to lowest priority:

1) If the name of the parameter appears in its description followed by the words "code" or "id", both words are used concatenated. For example, if the parameter name is `country` and its description is "A valid country code", this rule will be matched and the keyword `countryCode` will be used for the search for predicates.
2) If a word K with the same initial characters as the name of the parameter appears in the description followed by the words "code" or "id", both the name of the parameter and K are concatenated with "code" or "id" and used for the search of predicates. For example, if the parameter name is `lang` and its description is "A language code", the keywords `langCode` and `languageCode` will be used in the search for predicates.
3) If a noun or an unknown word (e.g., an acronym or a non-English word) is found in the description of the parameter, followed by the words "code" or "id", both words are concatenated and used for the search of predicates. If multiple matches are found, the keywords are considered in order of appearance. For example, if the parameter name is `origin` and its description is "A valid country code or airport code", the keywords `countryCode` and `airportCode` will be used for the search for predicates.
4) This rule has no condition. Whenever it is reached, the unmodified parameter name is used for the search for predicates, e.g., `streetAddress` (Listing 1).
5) If the name of the parameter is in snake case format (i.e., word1_word2) or in kebab case format (i.e., word1-word2), the parameter name is converted to camel case format (i.e., word1Word2) and used in the search for predicates. For example, the operation for matching businesses in the Yelp API [42] contains a parameter called `zip_code`; using this as a keyword (rule 4) returns 0 candidate predicates in DBpedia, whereas using `zipCode` returns a list of 5 predicates.
6) If the parameter name is in snake case, kebab case or camel case format, it is split into multiple words, and each one is used as a keyword to search for predicates. For example, several operations of the Great Circle Mapper API [43] contain a parameter called `icao_iata`, which accepts both IATA and ICAO codes. Using the unmodified parameter name (rule 4) produces 0 results, and so does converting it to camel case format (rule 5). However, when searching for predicates with the keywords `icao` and `iata`, multiple predicates are obtained.

Matching rules 4 to 6 are based on the common naming convention for web API parameters, whereas rules 1 to 3 are based on collocations we found during our preliminary work with 10 randomly selected APIs from different domains [23] (not included in our evaluation dataset to avoid overfitting). We remark that the list of matching rules is not exhaustive and new patterns could be readily targeted in the future by adding new rules. If no predicates are found for any of the matching rules, the parameter is ignored and no input values will be generated for it.

3.3 Search for test inputs

At this stage, the predicates obtained during the previous phase are used to drive the search for meaningful test

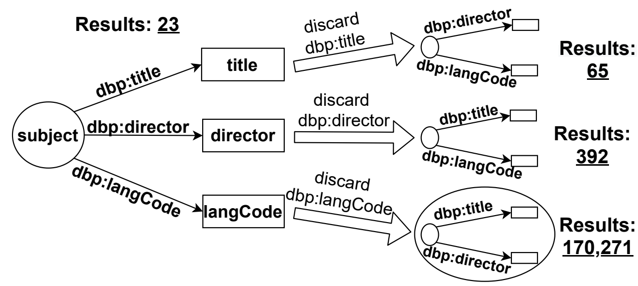

Fig. 2. Graphical representation of the search for test inputs.

inputs. When searching test inputs for a specific parameter, it is important to consider it in the context of all the parameters supported by the API operation. For example, a search for `title` may return hundreds or even thousands of title-type inputs belonging to very diverse entities (e.g., movies, people, videogames, books, etc.). However, if we accompany such predicate with another semantically related one such as `director`, the search is significantly narrowed, increasing the chances of generating semantically valid test inputs, e.g., movie titles. For this reason, instead of evaluating the predicates in isolation, our approach starts by looking for entities that contain all the predicates simultaneously and, if not enough inputs are obtained, then the predicates are progressively discarded.

As an example, Listing 4 shows the SPARQL query constructed for the search of test inputs (i.e., objects in RDF triples) for the parameters `title`, `director` and `langCode`. This query returned 23 matches in DBpedia. Each match contains three values, one for each parameter (i.e., `title`, `director` and `langCode`). However, some values may be repeated, therefore less than 23 *unique* values could be obtained for each parameter. A threshold is established to define the minimum number of unique values that should be obtained (100 in our experiments). Suppose that, for a subset of parameters, the threshold is not reached (e.g., there are enough values for `langCode`, but not for `director` and `title`). Then, ARTE would repeat the query depicted in Listing 4 considering only the predicates related to the parameters for which the threshold has not been achieved (i.e., `director` and `title`), thus widening the search.

```
SELECT DISTINCT ?title, ?director, ?langCode WHERE {
 ?subject <http://dbpedia.org/property/title> ?title ;
    <http://dbpedia.org/property/director> ?director ;
    <http://dbpedia.org/property/langcode> ?langCode .
}
```

Listing 4. SPARQL query for searching test inputs.

Figure 2 (left-hand side) depicts an example for the query in Listing 4. A search is performed for entities including the predicates `director`, `title`, and `langCode`, obtaining 23 results. Suppose that such search does not return the minimum threshold for any of the three parameters. This being the case, ARTE would execute a number of queries equal to the number of parameters, each containing all predicates but one (right-hand side of Figure 2). As illustrated, if `title` or `director` are discarded, 65 and 392 results are returned, respectively, but if `langCode` is discarded, the resulting query would return 170,271 results. Values for the `title`

TABLE 1
Example of automatically generated regular expression.

Valid values	Invalid values	Regex
CN	ITA	
ES	DOM	
IT	BGR	^\w[^\d]$
JP	92	
AD	HUN	

TABLE 2
ARTE configuration parameters and default values.

#	Parameter	Value
1	Number of predicates selected when searching for predicates	5
2	Min predicate support	20
3	Min number of unique parameter values	100
4	Min recall for accepting a regex	90%
5	Min number of valid and invalid values for generating a regex	5
6	Max number of extra predicates to leverage using regex	3
7	Max number of consecutive attempts to generate a regex	2

and `director` parameters would be extracted from this last query (since it is the one returning the highest number of results), while values for the `langCode` parameter would be obtained as a result of using its predicate in isolation.

This query-decomposition process continues until all parameters acquire the established threshold of input values, or until all predicates are executed in isolation.

3.4 Automated generation of regular expressions

Finally, ARTE can optionally generate regular expressions for each target parameter based on the input values obtained in the previous phases, by integrating the tool RegexGenerator++ [44], [45], [46] (details in Section 4). This allows to refine the search for syntactically valid inputs, improving the effectiveness of the approach. Table 1 depicts an example of this process for the parameter `market` in the Spotify API [24]. Specifically, the regular expression in Table 1 matches strings that are two-characters long, where the second character is not a digit; this pattern conforms to the format of ISO alpha-2 country codes, and allows to filter out values of different length (e.g., "ITA") and numerical values (e.g., "92"). The basic steps of the process are as follows:

1) A set of input values is generated as described in the previous sections, and classified as syntactically valid or invalid (first and second columns in Table 1). Inputs can be classified as valid or invalid either manually or, as in our work, automatically based on the API responses (see Section 4 for details). The number of valid and invalid values must reach a minimum threshold defined by the user (5 in our experiments).
2) Based on the generated valid and invalid inputs, a regular expression is generated (column "Regex" in Table 1). This regular expression should match and not match as many valid and invalid inputs as possible, respectively.
3) If the percentage of valid values matching the regular expression (i.e., recall) exceeds a minimum threshold (90% in our evaluation), the generation of the expression is considered successful. In that case, the list of input values generated until that moment is filtered such that only those values matching the regular expression are kept. Invalid values matching the regular expression (i.e., false positives) may still be helpful to identify potential bugs. For instance, the regular expression generated for the parameter `countryCode` in the DHL API matched both `"UK"` and `"FR"`, however, the former was rejected by the API (i.e., classified as invalid). This may reveal unintended behavior, e.g., codes that should be supported by the API but are not.

If a regular expression is successfully generated, ARTE extracts more input values by using the predicates that have not been leveraged yet (obtained as described in Section 3.2) adding the generated regular expression as a filter clause in the SPARQL query (Listing 4). These inputs will match the regular expression and therefore they should be more likely to be syntactically valid. On the contrary, if a regular expression cannot be inferred, the process restarts in step 1, i.e., ARTE will try to generate a regular expression with more input values classified as valid or invalid.

The process ends once the maximum number of consecutive attempts to generate regular expressions is reached (2 in our evaluation), or after ARTE has extracted input values from a maximum number of extra predicates leveraging the generated regular expressions (set to 3 in our experiments).

4 Tooling

We implemented ARTE in Java, leveraging existing libraries for specific tasks, namely: (1) Jena [47], for the creation of SPARQL queries; (2) Stanford CoreNLP [48], for NLP related tasks; and (3) RegexGenerator++ [44], [45], [46], for the generation of regular expressions. RegexGenerator++ uses search-based techniques for automatically generating context-aware regular expressions based on strings tagged as *valid* or *invalid* (i.e., *matching* or *not matching* the regular expression, respectively) within a corpus, namely, a text that provides some context. We slightly modified RegexGenerator++ to make it not context-aware, by adding the anchors '^' and '$' at the start and end of the generated regular expressions, respectively. This modification significantly improves the performance of ARTE for the generation of regular expressions.

Table 2 summarizes the configuration parameters of ARTE, described in the previous section, and the default values used in our evaluation. These values were selected based on our preliminary work with the tool. They provide a balance between performance and effectiveness. Hence, for example, setting low values for parameters 1-3 would result in faster execution, but lower probabilities of finding good values. Analogously, setting low values for parameters 4 and 5 would make it easier for ARTE to generate a regular expression, with the risk of it not being sufficiently accurate. On the contrary, setting a high value for parameters 6 and 7 would result in a slower execution with the risk of obtaining overfitted regular expressions. We refer the reader to the documentation of the project in GitHub [49] for more details about the parameters and their impact on the performance of ARTE.

We integrated ARTE into RESTest [22], a state-of-the-art black-box test case generation framework for RESTful APIs. More precisely, ARTE automatically generates data dictionaries (i.e., sets of valid and invalid values) for the selected API parameters, releasing testers from that burden. This makes ARTE easily applicable to any of the test case generation algorithms implemented in RESTest.

For the generation of regular expressions, input values must be labeled as valid or invalid. By integrating ARTE into RESTest, we automate this step as follows:

- An input parameter value is classified as *valid* if it was present in an API call for which a successful response (2XX status code) was obtained.
- An input parameter value is classified as *invalid* if it was present in an API call for which a "client error" response (4XX status code) was obtained and either: (1) it was the only parameter used in the API call; or (2) it was accompanied by other parameters whose values were already classified as valid.
- Input values not meeting any of the previous constraints are labeled as *unclassified* and ignored when inferring regular expressions.

RESTest generates and executes a fixed number of test cases by iterations. After each iteration, ARTE automatically classifies the generated values as described above and tries to infer a regular expression (Section 3.4).

5 EVALUATION

We aim to answer the following research questions:

- **RQ1:** *How effective is ARTE in generating realistic test inputs for web APIs?* We aim to measure the effectiveness of ARTE in generating syntactically and semantically valid test inputs for real-world web APIs.
- **RQ2:** *What is the impact of ARTE on the automated generation of test cases for web APIs?* The final goal of our approach is to improve current state-of-the-art methods for test case generation of web APIs. Therefore, we wish to compare the effectiveness of existing methods and ARTE in terms of valid API requests generated and test coverage achieved.
- **RQ3:** *Does ARTE improve the failure-detection capability of existing test case generation techniques?* We aim to investigate whether input values generated by ARTE reveal more failures than those generated by related techniques.

In the next sections, we explain the two experiments performed for answering the research questions. In both cases, we used the default configuration of ARTE, and we relied on DBpedia as the selected knowledge base, more specifically, the 2016-10 core dataset.[1]

The experiments were performed in a desktop machine equipped with Intel i7-6700 CPU@3.40GHz, 16GB RAM, and 125GB SSD running Windows 10 Pro 64 bit and Java 8.

5.1 Baselines

We compared ARTE against four baselines, three (pseudo)-random test data generation strategies (experiment 2) and SAIGEN [14] (experiments 1 and 2), described below.

1. http://downloads.dbpedia.org/2016-10/

5.1.1 Random test data generation techniques

We implemented three related, but different random test data generation approaches recently used in the context of automated test case generation for web APIs. All the generators were implemented using RESTest. Next, we present them, from the most naive to the most sophisticated one.

5.1.1.1 Fuzzing: This technique aims at finding implementation bugs (especially security-related) by using random, malformed or unexpected input data [50]. In our experiments, we use the *fuzzing* test case generator implemented in RESTest [22], which generates random values including out-of-range numerical values, long strings with unusual characters, empty strings, and null values.

5.1.1.2 Data dictionaries: This approach proposes to use a small set of predefined values for each parameter type. Specifically, we used the small data dictionaries proposed by Atlidakis et al. [27], namely: "sampleString" and "" (empty string) for string parameters, "0" and "1" for integers, and "1.23" for doubles.

5.1.1.3 Data generators: This approach uses specific test data generators for each parameter type [22], [25]. We used the default test data generators integrated into RESTest, which also leverage the regular expressions and min/max constraints included in the API specification, if any. Basically, we generated random English words for string parameters, numbers between 1 and 100 for integers, and floating numbers between 1 and 100 for doubles.

5.1.2 SAIGEN

To the best of our knowledge, our work is the first to leverage the Web of Data for driving test data generation for web APIs. However, semantic information retrieval techniques have already been successfully applied in the context of GUI testing of desktop, web, and mobile applications [13], [14]. In this article, we compare ARTE against the most recent of these contributions, SAIGEN (Semantic Aware Input GENerator) [14], a related tool for the automated generation of realistic test inputs for mobile apps.

Both approaches, ARTE and SAIGEN, exploit knowledge bases for the generation of realistic test inputs, but with significant differences. First, SAIGEN searches for test inputs based on the GUI labels and potential synonyms, whereas ARTE exploits the API specification, applying NLP techniques to the names and descriptions of the parameters. This means that ARTE can leverage further information than SAIGEN, but in practice this also imposes new challenges since, as explained in Section 3, parameters can use different naming conventions (e.g., parameters with a single letter) and very heterogeneous descriptions in natural language. Second, SAIGEN searches for predicates by adding the selected keyword to the namespace prefix of the knowledge base, whereas ARTE constructs specific SPARQL queries. In practice, this means that searches in SAIGEN are restricted to predicates containing exactly the specified keyword (e.g., `currency`), whereas ARTE widens the search space by looking for predicates containing the keyword (e.g., *`currency`*`Iso`). Lastly, ARTE integrates a novel step for the refinement of the test inputs extracted with automatically generated regular expressions (Section 3.4).

Since SAIGEN targets GUI elements only, in our experiments we ran the tool using the name of the API parameters as if they were GUI labels. We used the default settings of SAIGEN, configuring it to work with the selected version of the knowledge base.

5.2 Experiment 1: Generation of realistic test inputs

In this experiment, we aim to answer RQ1 by evaluating the effectiveness of ARTE in generating realistic test inputs for real-world web APIs. In what follows, we describe the setup and the results of the experiment.

5.2.1 Experimental setup

For this experiment, we resorted to RapidAPI [51], a popular online repository containing more than 35K web APIs, classified in 46 different categories. Specifically, we created a dataset of 40 APIs as follows. We selected the first 10 APIs of each category, i.e., $46 \times 10 = 460$ APIs. APIs and categories were sorted in the same order as they were displayed on the platform at the time of performing the search on May, 2021. Then, we selected the first 40 APIs of the list, filtering out those meeting any of the following exclusion criteria: (1) APIs containing exclusively domain-specific parameters (e.g., database IDs) or trivial parameters (such as `limit`, `offset` or enumerated values), for which random generation would suffice, (2) APIs for which there was no validation for any of the parameters (i.e., any value would be considered valid), (3) APIs containing only confidential parameters (e.g., username, password, email, or credit card number), (4) APIs not returning any response, (5) APIs without parameters, (6) paid APIs (so-called *premium* in RapidAPI), and (7) APIs whose documentation was not written in English. The resulting dataset contains 40 APIs from 11 different categories. Table 3 shows, for each API, its name, category, number of operations, and number of parameters used in our evaluation. Parameters used in more than one operation within the same API are considered only once, e.g., `iso_a2` in the Referential API [52]. In total, the dataset includes 173 different parameters from 122 API operations. These parameters span multiple concepts such as website URLs, city names, ingredient names, and currency codes, among many others.

RapidAPI does not provide a publicly available OAS specification for the APIs in the repository. To address this issue, we generated the OAS specification of each API using the web scraping libraries Selenium [53] and BeautifulSoup [54] on the web user interface of RapidAPI, which displays, among others, the name and type of each input parameter for every API operation.

To further evaluate our approach, we used a second dataset of 8 industrial popular RESTful APIs from different domains, depicted in Table 4. The dataset includes 38 parameters from 18 API operations. The OAS specification of Spotify [24] was downloaded from the APIs.guru repository [55]. The specifications of Yelp Fusion [42] and RESTCountries [56] were manually created from the documentation available on the official website. The OAS specification of the remaining APIs were downloaded from their official websites. The specification of the Amadeus Hotel API [6] was the only one including regular expressions for some of its parameters.

We compared ARTE against SAIGEN, since it is the only baseline specifically tailored for the automated generation of realistic test inputs. Additionally, we measured the performance of ARTE before applying the refinement with automatically generated regular expressions, denoted as ARTE NR (ARTE No Regex).

ARTE and SAIGEN may generate hundreds or even thousands of values for each parameter, making it infeasible to manually test all of them. To evaluate whether syntactically and semantically valid values were generated, we tested the APIs with 10 randomly selected values per parameter, manually checking whether at least one of them was valid. A similar approach was followed by the authors of SAIGEN [14]. We refer the reader to Section 3 for the definition of syntactically and semantically valid test inputs.

5.2.2 Experimental results

The results of ARTE and SAIGEN in the RapidAPI dataset are shown in the last nine columns of Table 3. On average, ARTE found syntactically valid inputs for 78% of the parameters (135 out of 173), whereas SAIGEN generated syntactically valid inputs for 42.8% of them (74 out of 173). Analogously, ARTE generated semantically valid inputs for 64.2% of the parameters (111 out of 173), while SAIGEN generated semantically valid inputs for 31.8% (55 out of 173). Regarding realistic values (both syntactically and semantically valid), ARTE generated realistic values for 63% of the parameters (109 out of 173), whereas SAIGEN only generated realistic values for 30.6% of them (53 out of 173). ARTE generated realistic inputs for 100% of the parameters in 9 out of 40 APIs (4 with SAIGEN), and 50% or more in 23 of them (13 with SAIGEN). It is noteworthy that the generation of regular expressions allowed to increase the percentage of realistic test inputs in 11 out of the 25 APIs for which ARTE did not initially obtain 100% of syntactically valid parameters. Among others, ARTE automatically generated regular expressions for parameters such as `season` (API-FOOTBALL and API-BASKETBALL APIs), `currency` (Skyscanner and Alpha Vantage APIs) and `state` (RedLine and Realty Mole APIs).

Table 4 shows the results for the set of industrial APIs, where the results of ARTE are even better than those obtained with the RapidAPI dataset. As illustrated, ARTE generated syntactically and semantically valid inputs for 73.7% of the parameters (28 out of 38), in contrast with SAIGEN, which generated realistic inputs for only 36.8% of the parameters (14 out of 38). Furthermore, ARTE generated realistic inputs for 100% of the target parameters in 3 out of 8 APIs (DHL, OMDb and RESTcountries), and over 50% in 6 of them, whereas SAIGEN did not manage to generate realistic inputs for all the parameters in any of the APIs, generating 50% or more in 3 of them (DHL, RESTCountries and Yelp). The automated generation of regular expressions yielded an improvement for the parameter `currency` of the RESTCountries API. Additionally, it improved the automated generation of valid test cases in the APIs of RESTCountries, Spotify and DHL, as explained in Section 5.3

In total, considering both the RapidAPI dataset and the industrial APIs, ARTE and SAIGEN generated realistic inputs for 64.9% (137 out of 211) and 31.8% (67 out of 211)

TABLE 3
Per API breakdown of input values generation for the RapidAPI dataset. O = Operations, P = Parameters.

API	Category	O	P	Syntactically valid (%)			Semantically valid (%)			Syntactically and semantically valid (%)		
				SAIGEN	ARTE NR	ARTE	SAIGEN	ARTE NR	ARTE	SAIGEN	ARTE NR	ARTE
AeroDataBox	Transportation	6	8	37.5	62.5	75	37.5	62.5	75	37.5	62.5	75
Airport info	Transportation	1	2	50	100	100	50	100	100	50	100	100
AirportIX	Transportation	6	6	33.3	16.7	50	16.7	0	33.3	16.7	0	33.3
Alpha Vantage	Finance	3	5	0	0	40	0	0	40	0	0	40
API-BASKETBALL	Sports	5	6	83.3	83.3	100	33.3	16.7	33.3	33.3	16.7	33.3
API-FOOTBALL	Sports	5	4	50	75	100	50	50	75	50	50	75
Astronomy	Science	1	2	100	100	100	100	100	100	100	100	100
Aviation Reference Data	Transportation	5	6	66.7	66.7	83.3	66.7	66.7	83.3	66.7	66.7	83.3
CarbonFootprint	Science	1	3	33.3	33.3	33.3	33.3	33.3	33.3	33.3	33.3	33.3
Countries Cities	Location	4	4	75	100	100	75	100	100	75	100	100
Currency Converter	Finance	2	4	50	75	75	0	50	50	0	50	50
Domainr	Business	2	3	66.7	66.7	66.7	0	33.3	33.3	0	33.3	33.3
Face Detection	Visual Recognition	1	1	0	0	0	0	0	0	0	0	0
Fixer Currency	Finance	3	4	0	100	100	0	100	100	0	100	100
Football Prediction	Sports	1	1	0	0	0	0	0	0	0	0	0
GeoDB Cities	Data	4	8	0	62.5	75	0	62.5	75	0	62.5	75
Google Maps Geocoding	Location	2	3	66.7	66.7	66.7	33.3	33.3	33.3	33.3	33.3	33.3
Great Circle Mapper	Travel	3	3	0	33.3	33.3	0	33.3	33.3	0	33.3	33.3
Hotels	Travel	3	3	66.7	100	100	0	33.3	33.3	0	33.3	33.3
Hotels Com Provider	Travel	1	2	100	100	100	100	100	100	100	100	100
Movie Database (Imdb alt.)	Entertainment	2	2	50	100	100	0	50	50	0	50	50
NAVITIME Route (car)	Transportation	2	3	0	0	0	0	0	0	0	0	0
NAVITIME Route (totalnavi)	Transportation	1	2	0	0	0	0	0	0	0	0	0
Periodic Table of Elements	Science	1	1	0	0	0	0	0	0	0	0	0
Priceline com Provider	Travel	2	4	50	100	100	50	100	100	50	100	100
Realty in US	Business	4	8	62.5	87.5	100	50	75	87.5	50	75	87.5
Realty Mole Property	Business	3	7	57.1	71.4	85.7	85.7	85.7	100	57.1	71.4	85.7
Recipe - Food - Nutrition	Food	8	7	14.3	71.4	71.4	14.3	85.7	85.7	14.3	71.4	71.4
RedLine Zipcode	Location	6	8	25	75	87.5	25	75	87.5	25	75	87.5
Referential	Data	8	18	27.8	72.2	72.2	27.8	44.4	44.4	27.8	44.4	44.4
Rent Estimate	Data	1	3	100	100	100	100	100	100	100	100	100
Restb.ai Watermark Detection	Visual Recognition	1	1	0	100	100	0	100	100	0	100	100
Skyscanner Flight Search	Transportation	8	5	40	40	80	0	0	40	0	0	40
Spott	Location	1	4	100	100	100	50	50	50	50	50	50
Subtitles for YouTube	Data	1	1	100	100	100	100	100	100	100	100	100
TrailAPI	Travel	2	5	40	100	100	40	80	80	40	80	80
Travel Advisor	Transportation	3	8	50	100	100	25	87.5	87.5	25	87.5	87.5
UPHERE.SPACE	Science	2	2	50	50	50	0	0	0	0	0	0
US Restaurant Menus	Food	6	5	80	100	100	60	80	80	60	80	80
Yahoo Finance	Finance	1	1	0	0	0	0	0	0	0	0	0
Total		**122**	**173**	**42.8**	**69.9**	**78**	**31.8**	**56.1**	**64.2**	**30.6**	**54.9**	**63**

TABLE 4
Per API breakdown of input values generation for the industrial APIs. O = Operations, P = Parameters.

API	O	P	Syntactically valid (%)			Semantically valid (%)			Syntactically and semantically valid (%)		
			SAIGEN	ARTE NR	ARTE	SAIGEN	ARTE NR	ARTE	SAIGEN	ARTE NR	ARTE
Amadeus Hotel	2	7	42.9	85.7	85.7	42.9	85.7	85.7	42.9	85.7	85.7
Deutschebahn StaDa	1	4	0	25	25	0	25	25	0	25	25
DHL Location Finder	2	6	50	100	100	50	100	100	50	100	100
Marvel	1	5	60	100	100	40	40	40	40	40	40
OMDb	1	3	33.3	100	100	0	100	100	0	100	100
RESTCountries	4	4	75	75	100	100	100	100	75	75	100
Spotify	5	4	25	75	75	25	75	75	0	50	50
Yelp Fusion	2	5	60	80	80	80	80	80	60	80	80
Total	**18**	**38**	**44.7**	**81.6**	**84.2**	**44.7**	**76.3**	**76.3**	**36.8**	**71.1**	**73.7**

of the parameters, respectively. It is worth recalling that SAIGEN was specifically designed for mobile apps, and therefore its poor performance on web APIs was expected. This supports the ability of ARTE to address the specific characteristics of web APIs.

Regarding performance, ARTE took on average 21.3 seconds (standard deviation σ = 30.6 seconds) to generate the set of input values for all the parameters of each API, whereas SAIGEN required 11.8 seconds on average per API (σ = 11.2 seconds).

In view of these results, we can answer RQ1 as follows:

RQ1: ARTE is effective in generating realistic test inputs for real-world web APIs. With a sample of 10 values per parameter, ARTE generated syntactically and semantically valid inputs for 64.9% of the parameters (137 out of 211), approximately twice as many as the baseline, SAIGEN, 31.8% (67 out of 211).

5.3 Experiment 2: Automated testing

In this experiment, we aim to answer RQ2 and RQ3 by evaluating how ARTE can contribute to the automated generation of valid API calls, API coverage, and detection of failures. Next, we describe the setup and the results of the experiment.

TABLE 5
Per API breakdown of valid calls, coverage, and failures detected. P = “Parameters”, VC = “Valid calls”, C = “Coverage”, F = “Failures”.

		Fuzzing			Data dictionaries			Data generators			SAIGEN			ARTE NR			ARTE		
API - Operation	**P**	**VC (%)**	**C (%)**	**F**	**VC (%)**	**C (%)**	**F**	**VC (%)**	**C (%)**	**F**	**VC (%)**	**C (%)**	**F**	**VC (%)**	**C (%)**	**F**	**VC (%)**	**C (%)**	**F**
Amadeus Hotel - Find hotels	7	0	3.3	0	1.6	68.4	0	8.1	72.2	0	3.7	73.6	0	8.6	75	7	**9.6**	**75.5**	9
Amadeus Hotel - View hotel rooms	2	0	3.6	0	17.5	65.1	0	43.4	64.1	0	30.8	**66.1**	0	51.5	65.6	0	**62.2**	65.1	0
Deutschebahn StaDa - Get stations	4	0.3	96.3	3	9.4	**97**	0	15.5	**97**	0	10.9	**97**	0	17.2	**97**	0	**19.2**	**97**	0
DHL - Find by address	4	0	3.8	0	0	3.8	0	0.1	24.1	0	0	3.8	0	13.7	**81**	138	**59.8**	**81**	140
DHL - Find by geo	2	0	3.9	0	0	3.9	0	0	3.9	0	94.1	**80.5**	0	**98.9**	**80.5**	0	97.7	**80.5**	0
RESTCountries - Capital	1	0	25	0	0	25	0	0.5	**100**	0	**19.9**	**100**	0	4.5	**100**	0	4.6	**100**	0
RESTCountries - Code	1	11.4	**100**	0	0	22.2	0	1.8	88.9	1	41.4	**100**	0	91.1	**100**	0	**99.2**	**100**	0
RESTCountries - Currency	1	0	22.2	0	0	22.2	0	0.7	88.9	0	0	22.2	0	3	**100**	1	**67.3**	88.9	0
RESTCountries - Language	1	0	22.2	0	0	22.2	0	0.2	88.9	0	16.5	88.9	0	18.3	**100**	2	**48.1**	88.9	0
Spotify - Get album	1	0	3.7	0	49.4	76.8	0	50.4	76.8	0	50.6	81.7	0	56.5	**89**	0	**92**	**89**	0
Spotify - Get category	2	0	20	0	47.5	80	0	49.3	85	0	53.4	80	0	53.6	85	0	**86.9**	**90**	0
Spotify - Get featured playlists	3	5.4	86.3	0	24.4	86.3	0	26.6	86.3	0	26.3	86.3	0	28	**88.2**	0	**42.6**	**88.2**	0
Yelp Fusion - Search business	5	0	8.8	0	18.3	94.1	0	37.9	94.1	0	16.9	94.1	0	**48.9**	**97.1**	0	44.8	94.1	0
Yelp Fusion - Search transactions	3	0	16.7	0	0	16.7	0	45.8	**72.2**	0	0	16.7	0	65.1	**72.2**	0	**67.7**	**72.2**	0
TOTAL	**37**	**1.2**	**29.7**	**3**	**12**	**48.8**	**0**	**20**	**74.5**	**1**	**26**	**70.8**	**0**	**39.9**	**87.9**	**148**	**57.3**	**86.5**	**149**

5.3.1 Experimental setup

For this experiment, we used 14 operations from 6 industrial APIs, depicted in the first column of Table 5.[2] For each operation, we generated and executed 1K API calls (20 iterations of 50 test cases) with each data generation strategy, leading to 14 (operations) $\times$ 6 (strategies) $\times$ 1K = 84K calls in total. Then, we computed the percentage of valid API calls generated, the API coverage achieved and the number of failures detected by each approach. For the computation of valid API calls generated, we resorted to the REST best practices [2], which dictate that valid API calls should obtain 2XX HTTP status codes, whereas invalid calls should obtain 4XX status codes.For computing the API coverage, we considered the test coverage criteria for RESTful web APIs defined in [57]. These criteria are classified into input criteria— elements covered by the API requests (e.g., operations and parameter values)—and output criteria— elements covered by the API responses (e.g., status codes and response body properties). Input elements (e.g., a parameter) are considered covered if they are included in at least one API request obtaining a successful response (i.e., 2XX status codes).

We used RESTest [22] for the generation and execution of test cases. In this experiment, each test case comprises a single API call. Web APIs often impose inter-parameter dependencies that restrict the way in which parameters can be combined to form valid API requests. For instance, the use of a parameter may require the inclusion of some other parameter. To handle these dependencies, we used the constraint-based test case generator integrated into RESTest [10], which supports the generation of API calls satisfying all the inter-parameter dependencies of the API operation under test [58], [59]. Therefore, when obtaining an error response, we are confident that it is due to individual input values, and not due to violation of the dependencies. The only exception is fuzzing, where dependencies are intentionally ignored to test the API with any potential input combination. Failures were automatically detected using the built-in test oracles in RESTest, mostly based on the detection of server errors (5XX status codes) and the identification of inconsistencies between the API specification and the API responses.

2. We excluded the Marvel API and the OMDb API from this experiment because the Marvel API always returns successful responses and the OMDb API does not use 4XX status codes for error responses.

Test data was generated in six fashions: using the four baselines (Section 5.1), ARTE without automated generation of regular expressions (again denoted as ARTE NR), and ARTE. Exceptionally, we used small data dictionaries (15-20 values) for domain-dependent parameters (e.g., album identifier in Spotify) in all the test data generation strategies excluding fuzzing.

Whenever ARTE or SAIGEN could not generate values for a parameter, it was assigned a random value using the data generators integrated into RESTest for a fair comparison among all the techniques under study. In practice, however, it may be sensible omitting the parameter from the API call, assuming it is optional, to increase the chances of generating a valid API request. Overall, ARTE could not generate values for 3 out of 37 parameters, whereas SAIGEN could not generate values for 9 of them.

5.3.2 Experimental results

We next describe the experimental results related to the automated test case generation (RQ2) and the detection of failures (RQ3).

5.3.2.1 *Automated test case generation*: Table 5 shows the percentage of valid calls generated and the coverage achieved by each test data generation strategy (columns “VC (%)” and “C (%)”, respectively). The highest values of each row are highlighted in boldface. On average, fuzzing achieved 1.2% of valid API calls, data dictionaries achieved 12%, data generators achieved 20%, SAIGEN achieved 26%, ARTE NR achieved 39.9%, and ARTE achieved 57.3%. ARTE obtained better results in 13 out of 14 API operations. The results obtained for the DHL API are especially significant: ARTE obtained between 59.8% and 98.9% of valid API calls, whereas the random approaches got 0%. This is because its `latitude` and `longitude` parameters are defined as `string` instead of `float`, making random generation useless. SAIGEN was able to generate valid values for `latitude` and `longitude`, but it did not generate valid values for country codes, which explains the 0% of valid calls generated for the operation “Find by address”. The automated generation of regular expressions played a key role in the generation of valid API calls in 6 out of 14 API operations under test, with a difference of up to 64.3% in the RESTCountries API. Regular expressions showed to be effective in generating test inputs for parameters such as country codes, markets, currency codes and language names.

In terms of API coverage, ARTE achieved an average coverage of 86.5% and 87.9% with and without the use of regular expressions, respectively, outperforming all other approaches by a margin of up to 58.2% (compared to fuzzing), 32% on average. This difference is mainly due to two reasons: (1) the poor performance of the related techniques in generating valid requests (e.g., fuzzing), a prerequisite to cover most API elements; and (2) the less varied data used in API requests, which results insufficient to cover some output elements. This was the case, for example, of the Spotify API, where ARTE covered between 8% and 12% more response body properties than data dictionaries, data generators, and SAIGEN.

Test case generation and execution using data dictionaries and fuzzing took, on average, about 11 minutes per API operation, data generators 19 minutes, SAIGEN 14 minutes, ARTE NR 18 minutes, and ARTE 17 minutes. These times are mainly influenced by the response time of the APIs under test, usually longer in valid calls. Thus, approaches generating more valid calls typically took longer to execute.

In view of these results, RQ2 can be answered as follows:

> **RQ2:** ARTE improved the automated generation of test cases for 13 out of the 14 web API operations under test. Specifically, ARTE generated over twice as many valid requests (57.3%) as SAIGEN (26%) and about three times as many as the best of the random approaches (20%). The superiority of ARTE was also reflected in the API coverage (86.5%).

5.3.2.2 *Failure detection*: Table 5 shows the number of failures detected by each approach. Data dictionaries and SAIGEN uncovered no failures, data generators uncovered 1 failure, fuzzing uncovered 3 failures, and ARTE uncovered a total of 149 failures (one less without using regular expressions). Failures can occur due to multiple reasons, for example, 5XX status codes (server errors), inconsistencies between the API responses and the API specification, or client error responses (i.e., 4XX status codes) obtained after valid API calls, or vice versa, when an invalid input value results in a 2XX response code. Data generators and fuzzing only uncovered failures in the form of 5XX status codes, whereas ARTE also uncovered failures in the form of inconsistencies in the API and unexpected API responses.

ARTE uncovered two issues not detected by any of the other approaches. In the "Find hotel rooms" operation of the Amadeus API, the documentation states that the `hotelName` parameter should contain 4 keywords maximum. However, there were 16 API calls violating this condition which obtained successful responses, revealing a fault. This bug was reported and confirmed by the API providers. Also, in the "Find by address" operation of the DHL API, we found that the documentation was not exhaustive, since the API accepted 27 country codes not listed in the API documentation. Additionally, the API returns a 400 code when using the country code UK, despite this being a valid ISO 3166-1 alpha-2 code, as indicated in the API documentation. After reporting these issues to DHL, they confirmed that it was the intended behavior and updated the documentation to reflect it. This shows the potential of ARTE to reveal disconformities between the API specification (or documentation) and its behavior.

In view of these results, we can answer RQ3 as follows:

> **RQ3:** ARTE revealed more failures in more APIs than related approaches. In particular, it uncovered 149 failures in 2 API operations, unveiling issues not detected by SAIGEN and random approaches.

5.4 Discussion

In what follows, we further explore the results and what they tell us about the research questions.

5.4.1 RQ1: Generation of realistic test inputs

The results of the first experiment show that, with just a sample of 10 values per parameter, ARTE managed to generate syntactically and semantically valid values for 64.9% of them, outperforming SAIGEN (33.1%). We may remark that the results obtained in this experiment are a pessimistic approximation, since we are only considering 10 input values per parameter, out of the hundreds or thousands of values generated by ARTE. Had we considered a larger sample (e.g., 100 values), we would have probably obtained better results. It is noteworthy that in all APIs, except for those of Recipe - Food - Nutrition and Realty Mole Property, the number of syntactically valid values is greater than or equal to the number of semantically valid values. This confirms that the main difficulty in test data generation lies in the generation of semantically valid inputs.

One of the distinctive features of ARTE compared to previous approaches is the automated generation of regular expressions (Section 3.4). Regular expressions did have a positive impact in the generation of test inputs for 11 out of 25 APIs from the RapidAPI dataset in which ARTE did not achieve 100% of syntactically valid values. In fact, automatically generated regular expressions were key to equal or outperform the results of SAIGEN in the APIs of AirportIX and API-BASKETBALL. Regular expressions worked well for parameters with unambiguous names, whose values follow a specific format (e.g., `countryCode`). However, they were not so effective for parameters with a more generic name (e.g., `code`) or not following any particular format (e.g., `hotelName` in the Amadeus API), therefore more research will be required for improving the generation of realistic inputs in those cases.

There are two main reasons why ARTE failed to generate valid inputs for some parameters: (1) the name and the description of the parameter are not descriptive enough; and (2) the parameter is too specific. The first reason mostly applied to the APIs from the RapidAPI dataset. We noticed that, in many cases, the parameters expecting a code do not explicitly state it (e.g., textually in the description or by using a name such as `countryCode`). In this scenario, ARTE may generate both names and codes, but these may not follow the correct format or simply be invalid. Industrial APIs, on the other hand, generally provide a more exhaustive documentation, but they tend to have very specific parameters such as `Ril-100` (identifier of German train stations in the Deutschebahn StaDa API) or ean (European Article Number in the Marvel API), hard to find in general-purpose knowledge bases like DBpedia. Despite this limitation, it is worth highlighting that ARTE successfully generated realistic inputs for German federal state names, ingredients names, website URLs, addresses, postal codes, country codes, currency codes, and language codes, among others.

Domain-specific parameters (e.g., database identifiers) are unlikely to be found in general-purpose knowledge bases. For such parameters, we resorted to manually-created data dictionaries, and therefore we could not achieve full automation. This limitation is shared by related techniques [10], [25], [60]. We see potential in combining ARTE with learning input values from previous responses [11], [61].

Regarding performance, ARTE took about 10 seconds more than SAIGEN (21.3 vs. 11.9) to generate test inputs for all parameters of each API, on average. This is because ARTE applies NLP techniques not only to the name of the parameter but also to its description. In addition, ARTE searches for predicates in DBpedia once for each matching rule (Section 3.2). However, we deem this as a negligible toll considering the gain in effectiveness, i.e., ARTE generated input values for 205 out of 211 parameters, as opposed to SAIGEN, which generated values for 144 of them.

5.4.2 RQ2: Automated generation of test cases

ARTE generated between 2 and 48 times more valid calls than the baselines. However, the improvement in the Amadeus Hotel API and the Deutschebahn StaDa API was not as significant. There were several reasons behind these results worth mentioning, since they could be extrapolated to other APIs, namely:

1) *Regular expressions in the specification*. The OAS specification of the Amadeus API contains regular expressions describing the format of some parameters (e.g., an ISO code for parameter `lang`). Therefore, the approach using data generators can also leverage these regular expressions, thus obtaining similar results.
2) *REST bad practices*. When an API call uses invalid input data, it should obtain a "client error" response. Conversely, when it uses valid data, it should obtain a successful response. This is not the case in the Amadeus API. Some parameters require inputs such as language codes, but the API returns successful responses as long as the value used conforms to the expected format (e.g., two capital letters).
3) *Semantic inter-parameter dependencies*. These are dependencies that constrain the values that different parameters can take based on their meaning. For instance, even if ARTE generates valid values for the parameters `cityCode` and `hotelName`, the API will return an error if there is no hotel with the specified name in the provided city and vice versa. These errors may also arise for trivial parameters, not targeted by ARTE. For example, if the number of rooms provided as input (an integer parameter) is greater than those available at the specified hotel.

ARTE outperformed all the baseline approaches in 13 out of 14 API operations under test. SAIGEN obtained better results in the operation "Capital" of the RESTCountries API because it leveraged a predicate that yielded mainly country capitals (those accepted by the API), whereas ARTE leveraged one that yielded capitals of both countries and regions. ARTE also outperformed the related approaches in terms of coverage, by a margin that ranged between 13.4% and 58.2%.

5.4.3 RQ3: Failure detection

The increase in the number of valid API calls generated by ARTE translates into more diverse tests that exercise different parts of the API under test and, consequently, uncover more failures. However, different test data generation strategies may uncover different types of failures, and therefore they are complementary, rather than exclusive. As an example, fuzzing uncovered a 500 status code in the Deutschebahn StaDa API, caused when using parameter values containing unexpected characters such as '%'. This bug could not have been uncovered by techniques leveraging realistic input data exclusively.

The integration of ARTE into RESTest enables leveraging not only valid test inputs but also invalid ones, since RESTest can automatically classify the values generated by ARTE into valid or invalid, according to the API responses. For instance, in the Spotify API, ARTE detected that the API returned a successful response for 6 markets that were not present in the documentation. Conversely, the value `"LY"` (Libya) for parameter `country` was classified as invalid, since it was rejected by the API. The same happened with 24 language names in the RESTCountries API. Are these values really not supported by the API, or may these responses be caused by a bug in their implementation? ARTE can reveal unexpected behaviors like these.

6 Threats to Validity

In this section, we discuss the possible internal and external validity threats that may have influenced our work, and how these were mitigated.

Internal validity. *Are there factors that might affect the results of our evaluation?* For the experiments, we used the OAS specifications of the APIs under test. When possible, we resorted to the API specifications publicly available. However, for the RapidAPI dataset and the Yelp and RESTCountries APIs such specifications were missing. We generated them manually (Yelp and RESTCountries) and using web scraping (RapidAPI), based on the information provided in the online documentation of each API. It is therefore possible that some of the OAS specifications have errors and deviate from the API documentation. To mitigate this threat, each of the OAS specification files was carefully reviewed by at least two authors.

The results obtained for the first experiment—ability of ARTE to generate realistic inputs—are based on a sample of 10 values per parameter. Given the random nature of this experiment, it should have been repeated several times (e.g., 10-30) and analyze the results with statistical tests. However, this was a manual process involving 211 (parameters) $\times$ 10 (values) $\times$ 2 (approaches) = 4,220 API calls, and so repeating it would be an extremely costly endeavor. Furthermore, we emphasize that the results obtained are simply a pessimistic approximation of what could be achieved with ARTE when considering all the inputs generated per parameter (instead of simply 10 values). The same threat applies to the second experiment. Industrial APIs impose restrictive quota limitations [10], [62], thus making it infeasible to execute thousands of requests without exceeding the quota and rate limits of the services under test. In spite of this, the high number of test cases generated and executed (84K) make us confident about the validity of the results.

Faults in the implementation of the tools used used—RESTest and SAIGEN—could compromise the validity of the results. To mitigate this threat, we carefully checked and tested the implementation of each test data generator and their results leveraging the existing regression test suites of RESTest and SAIGEN.

External validity. *To what extent can we generalize the findings of our investigation?* We evaluated our approach on a subset of APIs and therefore our conclusions could not generalize beyond that. To mitigate this threat, we evaluated ARTE on a set of 140 operations from 48 real-world RESTful APIs, including popular industrial APIs with millions of users worldwide. Additionally, we selected APIs belonging to different application domains and various sizes in terms of number of operations and parameters.

ARTE applies several heuristics to infer realistic values based on the name and description of API parameters. In particular, we proposed six matching rules to generate predicates that are likely to return semantically valid test inputs (Section 3.2). These rules are based on common naming conventions for API parameters and our work with a subset of APIs, and therefore could not generalize further. However, we may remark that this does not invalidate our results and that new matching rules could be readily added in the future.

7 Related Work

To the best of our knowledge, our work is the first to leverage the Web of Data for improving test data generation in web APIs. Nevertheless, semantic information retrieval techniques have already been applied in the context of GUI testing. Mariani et al. [13] presented Link, an approach to retrieve realistic test inputs for web, desktop, and mobile applications from DBPedia. Wanwarang et al. [14] introduced SAIGEN, which follows the same principles of Link, but is specifically tailored for mobile apps. Evaluation results on 12 mobile applications showed that SAIGEN was able to find inputs for 50% of the GUI labels on average. Out of these, 94% of inputs were semantically valid [14]. Our work shares similarities with both papers, but also clear differences, as detailed in Section 5.1.2. Link and SAIGEN exploit GUI labels, whereas ARTE exploits the API specification, including the name and the description of input parameters. In theory, this gives an advantage to ARTE since it can exploit further information. However, in practice, we found that this also implies new challenges since parameter names and descriptions tend to be very heterogeneous. This explains why ARTE resorts to NLP techniques, for instance, when the name of a parameter does not provide sufficient information (e.g., parameter `t` in the OMDb API [35]) and the most helpful information is contained in its description (e.g., "a movie title"). On the other hand, we proposed a novel approach to iteratively refine test inputs by automatically generating regular expressions conforming to them (Section 3.4), and we integrated ARTE into RESTest (Section 4), providing a fully automated semantic-aware testing tool for RESTful APIs. The results of our evaluation show that ARTE represents a significant improvement over related approaches in the context of web API testing.

Besides semantic-enabled approaches, other authors advocate for extracting realistic test inputs from other sources. Shahbaz et al. [63], [64] proposed generating valid and invalid string test data based on Web searches and predefined regular expressions (e.g., following the format of an email address). Bozkurt and Harman [65] relied on web service composition [66] to generate realistic test data, i.e., by finding a web service that returns as output the data required by a different service. Clerissi et al. [67] presented DBInputs, an approach to testing web applications by reusing data stored in the system database (e.g., a resource identifier). Compared to these techniques, ARTE is specifically tailored for web APIs, and it does not require access to the source code or the database of the system under test [63], [64], [67], nor to other web services [65], just its specification.

Our work is very much related to RESTful API testing, a thriving research field nowadays. Approaches can be divided into black-box and white-box. In black-box testing, the API specification—generally an OAS document [9]—drives the generation of test inputs. Several strategies with varied degrees of thoroughness and automation have been proposed in the literature: (1) Ed-douibi et al. [12] proposed extracting default and example values from the OAS specification of the API under test; (2) Atlidakis et al. [27] used fuzzing dictionaries for each data type (e.g., `0` and `1` for *integer* parameters); (3) other authors [10], [25] advocate for using customizable test data generators for each parameter under test (e.g., a generator of real coordinates for a mapping API); (4) lastly, it may be possible, in some cases, to extract input values from previous API responses [11], [61], e.g., when some API operation returns data required as input by a different API operation. Approaches (1) and (2) are automated, but fall short for generating realistic test inputs and testing web APIs thoroughly. Approach (3) can generate varied and realistic values, but it requires manual work for the development of test data generators. Approach (4) offers the best tradeoff but may not always be applicable. Compared to prior work, ARTE generates realistic test inputs in a highly automated fashion, solely based on the API specification. More importantly, as a test data generation approach, ARTE is complementary to existing test case generation techniques for RESTful APIs.

White-box techniques for RESTful API testing are less common than black-box, since the source code of the system is required. Arcuri [28] is the only author who advocates for white-box testing. He proposed an evolutionary approach where test inputs are randomly sampled at the beginning of the search and subsequently mutated, aiming to maximize code coverage and fault finding. In this sense, our contributions may be complementary: the test inputs generated by ARTE could be used as a seed and subsequently evolved, thus providing a bootstrap for the search [68].

8 Conclusions

This article presented ARTE, an approach for the automated generation of realistic test inputs for web APIs. ARTE analyzes the specification of the API under test to extract semantically related concepts for every API parameter. Then, those concepts are used to query a knowledge base from which to extract test inputs. As a distinctive feature, ARTE implements an iterative process for the refinement of test inputs through the automatic generation of regular expressions. Valid and invalid parameter values—those accepted and rejected by the API, respectively—are used to create regular expressions according to them. This allows ARTE to filter undesired values in subsequent queries to the knowledge base. ARTE has been integrated into RESTest, a black-box testing framework for RESTful APIs. In practice, this allows to automate the whole testing process: test data generation using ARTE, test case generation, test case execution, and assertion of test outputs. Evaluation results on 140 operations from 48 web APIs show the effectiveness of ARTE to generate realistic test inputs and its potential to boost the fault detection capability of test case generation tools for web APIs.

There are several potential lines of future work. It would be interesting to explore new heuristics for the search of more effective predicates, for example, by applying more advanced NLP techniques to the name and description of input parameters. Refining the feedback loop for generating regular expressions from previous responses would also be a natural extension. Finally, we plan to develop a web API to ease the integration of ARTE into third party tools.

Verifiability

For the sake of replicability, we provide a supplementary package containing the source code of the tools, the datasets used, and the results generated [23].

Acknowledgments

This work has been partially supported by the European Commission (FEDER) and Junta de Andalucia under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación) under project HORATIO (RTI2018-101204-B-C21) and the Excellence Network SEBASENet 2.0 (RED2018-102472-T), and the Spanish Ministry of Education and Vocational Training (FPU17/04077).

References

[1] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011.

[2] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] "BeSoccer API," https://www.besoccer.com/api/, accessed May 2021.

[5] "Twitter API," https://developer.twitter.com/en/docs, accessed May 2021.

[6] "Amadeus API," https://developers.amadeus.com/, accessed April 2021.

[7] "DeepL API," https://www.deepl.com/docs-api/, accessed May 2021.

[8] "Openroute API," https://openrouteservice.org/dev/#/api-docs, accessed May 2021.

[9] "OpenAPI Specification," https://www.openapis.org, accessed April 2020.

[10] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2020, pp. 459–475.

[11] E. Viglianisi, M. Dallago, and M. Ceccato, "RestTestGen: Automated Black-Box Testing of RESTful APIs," in *International Conference on Software Testing, Verification and Validation*, 2020.

[12] H. Ed-douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference*, 2018, pp. 181–190.

[13] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Link: Exploiting the web of data to generate test inputs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, p. 373–384.

[14] T. Wanwarang, N. P. Borges, L. Bettscheider, and A. Zeller, "Testing Apps With Real-World Inputs," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, p. 1–10.

[15] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "DBpedia-A crystallization point for the Web of Data," *Journal of web semantics*, vol. 7, no. 3, pp. 154–165, 2009.

[16] "Asos API," https://rapidapi.com/apidojo/api/asos2, 2020, accessed May 2021.

[17] "DHL Location Finder API," https://developer.dhl.com/api-reference/location-finder, accessed May 2021.

[18] "Numverify API Documentation," https://numverify.com/documentation, accessed November 2021.

[19] "Foursquare API," https://developer.foursquare.com/places-api, accessed April 2021.

[20] "Domainr API," https://rapidapi.com/domainr/api/domainr, accessed November 2021.

[21] "GeoDBCities API Documentation," https://rapidapi.com/wirefreethought/api/geodb-cities, accessed November 2021.

[22] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Automated Black-Box Testing of RESTful Web APIs," in *International Symposium on Software Testing and Analysis*, 2021.

[23] "Replication package," http://doi.org/10.5281/zenodo.5792081.

[24] "Spotify Web API," https://developer.spotify.com/web-api/, accessed February 2021.

[25] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs," in *International Conference on Software Testing, Validation and Verification*, 2020, pp. 131–141.

[26] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic Testing of RESTful Web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.

[27] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758.

[28] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, 2019.

[29] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," *Int. J. Semantic Web Inf. Syst.*, vol. 5, pp. 1–22, 2009.

[30] T. Berners-Lee, R. Fielding, L. Masinter *et al.*, "Uniform resource identifiers (uri): Generic syntax," 1998.

[31] R. Guha and D. Brickley, "RDF vocabulary description language 1.0: RDF schema," W3C, W3C Recommendation, 2004, https://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[32] "SPARQL 1.1 overview," https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/, accessed April 2021.

[33] S. Krishna, *Introduction to database and knowledge-base systems*. World Scientific, 1992, vol. 28.

[34] D. Vrandečić and M. Krötzsch, "Wikidata: A Free Collaborative Knowledge base," *Commun. ACM*, vol. 57, no. 10, p. 78–85, 2014.

[35] "OMDb API," http://www.omdbapi.com, accessed May 2021.

[36] "dblp API," https://dblp.org/faq/13501473.html, accessed April 2021.

[37] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[38] C. Manning and H. Schutze, *Foundations of statistical natural language processing*. MIT press, 1999.

[39] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, 2003, pp. 173–180.

[40] S. Vijayarani, M. J. Ilamathi, and M. Nithya, "Preprocessing techniques for text mining-an overview," *International Journal of Computer Science & Communication Networks*, vol. 5, no. 1, pp. 7–16, 2015.

[41] V. Balakrishnan and E. Lloyd-Yemoh, "Stemming and lemmatization: a comparison of retrieval performances," *Lecture Notes on Software Engineering*, pp. 262–267, 2014.

[42] "Yelp Fusion API," https://www.yelp.com/developers/documentation/v3, accessed May 2021.

[43] "Great Circle Mapper," https://www.greatcirclemapper.net/en/api.html, accessed May 2021.

[44] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, 2016.

[45] A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, and M. Virgolin, "Evolutionary learning of syntax patterns for genic interaction extraction," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1183–1190.

[46] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Can a machine replace humans in building regular expressions? a case study," *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 15–21, 2016.

[47] "Apache Jena," https://jena.apache.org/index.html, accessed April 2021.

[48] "Stanford CoreNLP," https://stanfordnlp.github.io/CoreNLP/, accessed April 2021.

[49] "ARTE documentation," https://github.com/isa-group/RESTest/wiki/Test-configuration-files#semanticgenerator-arte.

[50] P. Godefroid, "Fuzzing: Hack, Art, and Science," *Commun. ACM*, vol. 63, no. 2, p. 70–76, 2020.

[51] "RapidAPI API directory," https://rapidapi.com/marketplace, accessed May 2021.

[52] "Referential API," https://rapidapi.com/referential/api/referential, accessed May 2021.

[53] "Selenium with Python," https://selenium-python.readthedocs.io/, accessed May 2021.

[54] "Beautiful Soup," https://www.crummy.com/software/BeautifulSoup/, accessed May 2021.

[55] "APIs.guru," https://apis.guru, accessed May 2021.

[56] "RESTCountries API," https://restcountries.com, accessed November 2021.

[57] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Test Coverage Criteria for RESTful Web APIs," in *A-TEST*, 2019, pp. 15–21.

[58] ——, "A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2019, pp. 399–414.

[59] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Transactions on Services Computing*, 2021.

[60] O. Banias, D. Florea, R. Gyalai, and D.-I. Curiac, "Automated Specification-Based Testing of REST APIs," *Sensors*, vol. 21, no. 16, 2021.

[61] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API Data Fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 725–736.

[62] A. Gamez-Diaz, P. Fernandez, A. Ruiz-Cortés, P. J. Molina, N. Kolekar, P. Bhogill, M. Mohaan, and F. Méndez, "The Role of Limitations and SLAs in the API Industry," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1006–1014.

[63] M. Shahbaz, P. McMinn, and M. Stevenson, "Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing," in *2012 12th International Conference on Quality Software*, 2012, pp. 79–88.

[64] ——, "Automatic Generation of Valid and Invalid Test Data for String Validation Routines Using Web Searches and Regular Expressions," *Science of Computer Programming*, vol. 97, pp. 405–425, 2015.

[65] M. Bozkurt and M. Harman, "Automatically Generating Realistic Test Input from Web Services," in *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. IEEE, 2011, pp. 13–24.

[66] S. Dustdar and W. Schreiner, "A Survey on Web Services Composition," *International journal of web and grid services*, vol. 1, no. 1, pp. 1–30, 2005.

[67] D. Clerissi, G. Denaro, M. Mobilio, and L. Mariani, "Plug the Database & Play With Automatic Testing: Improving System Testing by Exploiting Persistent Data," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 66–77.

[68] A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortés, "Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?" in *International Symposium on Software Reliability Engineering*, 2021.

Juan C. Alonso is a PhD candidate at the SCORE Lab, University of Seville, Spain. His current research interests lie in the areas of software testing, natural language processing, repository mining and deep learning. Contact him at javalenzuela@us.es.

Alberto Martin-Lopez is a PhD candidate at the SCORE Lab, University of Seville, Spain. He received his MsC from this university. His current research interests focus on automated software testing and service-oriented architectures. He is a Fulbright fellow at the University of California, Berkeley, and the winner of the ACM Student Research Competition held at ICSE 2020. Contact him at alberto.martin@us.es.

Sergio Segura is an Associate Professor of software engineering at the University of Seville, Spain. He is a member of the SCORE Lab, where he leads the research lines on software testing and search-based software engineering. His current research interests include test automation and AI-driven software engineering. Contact him at sergiosegura@us.es.

José María García is an Associate Professor of software engineering at the University of Seville, Spain. He is a member of the SCORE Lab. His research focus is on blockchain, semantic web technologies, service-oriented architectures and linked data. Contact him at josemgarcia@us.es.

Antonio Ruiz-Cortés is a Full Professor of software and service engineering and elected member of the Academy of Europe. He heads the SCORE Lab at the University of Seville. His current research focuses on service-oriented computing, business process management, testing and software product lines. He is an associate editor of Springer Computing. Contact him at aruiz@us.es.