RESTTESTGEN: Automated Black-Box Testing of RESTful APIs

Emanuele Viglianisi
Fondazione Bruno Kessler
Trento, Italy

Michael Dallago
University of Trento
Trento, Italy

Mariano Ceccato
University of Verona
Verona, Italy
mariano.ceccato@univr.it

***Abstract*—RESTful APIs (or REST APIs for short) represent a mainstream approach to design and develop Web APIs using the REpresentational State Transfer architectural style. When their source code is not (or just partially) available or the analysis across many dynamically allocated distributed components (typical of a micro-services architecture) poses obstacles to white-box testing, black-box testing becomes a viable option. Black-box testing, in fact, only assumes access to the system under test with a specific interface.**

This paper presents RESTTESTGEN, a novel approach to automatically generate test cases for REST APIs, based on their interface definition (in Swagger). Input values and requests are generated for each operation of the API under test, with the twofold objective of testing nominal execution scenarios and of testing error scenarios. Two distinct oracles are deployed to detect when test cases reveal implementation defects. Our empirical investigation shows that this approach is effective in revealing actual faults on 87 real-world REST APIs.

***Keywords*-Black-box testing, REST APIs, Automatic test case generation, Oracle.**

I. INTRODUCTION

REST APIs are the de-facto standard to implement and grant remote access to Web APIs. They are so largely accepted and adopted that the *Berlin Group Initiative*[1] elaborated a standard based on REST APIs for unifying the European Banking APIs. This initiative was meant to address the PSD2 European Union directive[2], that requested banks to open their customer data to authorized third-party service providers. Moreover, reference implementations of PSD2 compliant banking APIs are mostly available in the form of REST APIs[3].

REST APIs are often components of micro-services architecture [15], according to which each component should be small and assigned just one (or very few) responsibilities, resulting in a high number of simple components. Distinct components are usually deployed in different containers that can be dynamically allocated and deallocated, possibly across different hosts (for load balancing reasons).

Such a dynamic implementation of REST APIs, spread across many independent containers, poses peculiar challenges to automated analysis and testing with white-box approaches. A black-box approach, instead, gives up the possibility to exploit program information available in the source code. A black-box approach, in fact, only relies to a well-defined interface to access the REST API and does not need to cope with the very complex internal details of how components are deployed and run.

Moreover, the implementation of REST APIs might include commercial third-party libraries or frameworks, that come in the form of compiled code. When source code is not (or only partially) available, a black-box perspective in automated testing of REST APIs is a natural option.

In this paper we present RESTTESTGEN, a novel approach to automatically generate test cases for REST APIs. This approach is based on the definition of the interface to interact with a REST API, including the list of operations available, the format of the input/output data of their requests and responses. We propose the *Operation Dependency Graph* as a way to explicitly model data dependencies among operations that can be inferred from the REST API interface. This graph is updated when automatically generating the test cases, to dynamically decide when a new operation can be tested, because all its required input data could be guessed.

RESTTESTGEN aims at testing REST APIs according to two perspectives. Nominal execution scenarios are meant to test the system using input data as they are documented in the interface, while error execution scenarios exploit input data that violate the interface to expose implementation defects and unhandled exceptional flows.

An extensive empirical validation has been conducted, involving 87 real REST APIs for which only black-box access is available. RESTTESTGEN was able to test these case studies to a large extent, revealing a considerable number of implementation defects.

The rest of the paper is organized as follows. Section II covers the background on REST APIs. Section III proposes an overview of the RESTTESTGEN modules, that are presented in detail in the subsequent sections. Section IV defines the Operation Dependency Graph, while Section V and Section VI present, respectively, the *Nominal Tester* and the *Error Tester* modules. In Section VII, an empirical assessment of RESTTESTGEN is conducted and presented. Finally, after discussing related work in Section VIII, Section IX closes the paper.

[1] https://www.berlin-group.org/
[2] https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en
[3] https://www.openbankproject.com/

II. Background

A. REST APIs

A RESTful API (or REST API for short) is an API that respects the REST (REpresentational State Transfer) architectural style [8]. Nowadays, most of the APIs use a RESTful architecture over HTTP protocol to manage resources, allowing clients to access and manipulate them using a set of stateless operations.

REST APIs provide a uniform interface to create, read, update and delete (CRUD) a resource. A resource is generally identified by an HTTP URI, and CRUD operations are usually mapped to the HTTP methods POST, GET, PUT and DELETE to the resource URI.

For example, suppose there exists a REST API managing a collection of pets. A possible HTTP URI pointing to the resource could be `/pets`. In this case, the HTTP method `GET /pets` is used to retrieve the list of the pets and `POST /pets` could be used to add a new pet to the collection.

The resource URI and the HTTP methods may accept input parameters, to specify additional information for executing the API operations, such as Id of the object to retrieve (e.g., `/pets/<petId>`) or a structured object to be added to the collection using the POST method.

B. OpenAPI and Swagger

OpenAPI[4] defines a standard to document REST APIs. According to OpenAPI, an API service is described using a structured file (either in YAML or JSON) that specifies how to reach the API using a URI, what authentication schema is used and the details of all the operations available in the API, the input parameters (and their schema) to be used in requests and the schemas of responses. Previous versions of this specification (older than version 3.0.0) were called *Swagger*. Since older versions are still the most used, and anyway there is no major difference, in this paper we will use the term *Swagger* to mean an API specification either in the old and in the new format version.

Listing 1 contains an example of Swagger for *PetStore*, an API for managing *Pet* resources within a store. After an initial header that specifies versions and licenses, the field *servers* contains the base URL of the API, `http://petstore.swagger.io/v1` in the example.

The array *paths* contains the list of URL paths available in the API. In our example, there are two paths, i.e. `/pets` and `/pets/{petId}`.

Each path supports one or more HTTP method operations, which are usually specified by an *OperationID*. The method *GET* in `/pets` (*getPets*) is used to retrieve the list of all the pets. The method *GET* of the path `/pets/{petId}` refers to the operation *getPetById*, meant to retrieve the *Pet* object that matches a specific *petId*. Path parameters are specified directly in the path URL using curly braces, such as the *petId* input parameter in our example.

Modifiers can be used to attach constraints to data fields. For instance, the modifier *required* specifies that a parameter is mandatory and it can not be omitted. Moreover, each

[4] https://www.openapis.org/

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: getPets
      tags:
        - pets
      responses:
        '200':
          description: PetIds
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    petId:
                      type: integer
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                  $ref: "#/components/schemas/Error"
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: getPetById
      tags:
        - pets
      parameters:
        - name: petId
          in: path
          required: true
          description: The id of the pet to retrieve
          schema:
            type: string
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pet"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    # ...
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
```

Listing 1. OpenAPI specification example

request input and output is associated with a *schema* that specifies its type and, optionally, a set of constraints on its value (e.g., a *min* or *max* value for numeric parameters). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the parameter *petId* (line 41) is of type string (line 46), while the response should be in json (line 51) according to the schema *Pet*, which is composed of the fields *id*, *name* and *tag*, as specified at lines 69-75.

The swagger not only specifies the format of response in the nominal case (i.e., response code 200, line 17), but also the response of the API when an error occurs (line 33).

III. Approach Overview

RESTTESTGEN is a black-box tool, intended to automatically generate test cases for REST API. As a black-box approach, RESTTESTGEN does not assume access to source code nor to the compiled binary code of the API under test. We only assume to have input/output access to the API via the HTTP protocol. The API Swagger specification should be also available, to know which operations can be called and their input/output data format, to send well-formed HTTP requests.

A black-box approach is the only option when the source code is not available, or only partially available, e.g. when third-party components or commercial libraries are integrated, whose source code is not available. Additionally, a black-box approach is a valuable option when testing APIs with an architecture that is very complex for a white-box approach, e.g. because consisting of many (micro-) services, possibly developed with different languages and technologies. In fact, a black-box approach is independent from the programming language used to implement the API to test.

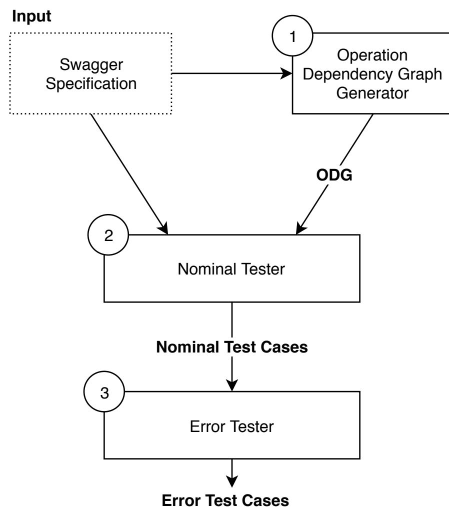

Figure 1. Automated test case generation: structure overview

RESTTESTGEN includes different modules, as shown in the overview of Figure 1. It takes as input the *Swagger* specification of the API service to test, to have information on the endpoints available in the REST API, the available operations and the input data format.

The first module analyzes the Swagger and computes the *Operation Dependency Graph*, a graph that models the data dependencies among the operations available in the service. This graph is meant to help our tool in sorting the operations to test depending on their data dependencies, i.e. first the operations are tested that output those data that are needed as inputs to test subsequent operations.

The next module, namely the *Nominal Tester*, reads the Operation Dependency Graph and the Swagger to automatically create test cases of the REST API. We called test cases generated by this module the *nominal* test cases, because they are created according to the specifications, trying as much as possible to follow the data constraints in the Swagger.

The nominal tests represent the input for the subsequent module, the *Error Handling Tester*. This last module applies a catalog of mutation operators to the nominal tests, with the aim of violating data constraints from the Swagger, to stress the data validation features of the REST API. For this reason, the tests generated by this module are called the *error* test cases.

IV. Operation Dependency Graph

This section describes the *Operation Dependency Graph* and how to build it, starting from the the information from the Swagger specification.

A. Graph Construction

The Operation Dependency Graph, or ODG for short, is a directed graph $G = (N, V)$. Nodes N are the operations in the REST API. The graph has an edge $v \in V$, with $v = n_2 \rightarrow n_1$, when there exists a data dependency between n_2 and n_1. We define a data dependency between two nodes n_2 and n_1, when there exists a *common field* in the output (response) of n_1 and in the input (request) of n_2. The intuitive meaning of this dependency is that n_1 should be tested before n_2, because the output of n_1 could be used to guess input values to test n_2.

We define two fields as *common* when:

- If they are of atomic type (i.e., string or numeric), they have the same name;
- If they are of non-atomic type (i.e., structured), they are associated to the same schema.

Edges are labeled with the name of the *common field(s)*, between the source and target nodes.

As an example, let's consider the segment of Swagger in Listing 1, it reports two operations. Operation *getPets* lists all the *petId* identifiers of all the pets available in the shop. Operation *getPetById* returns all the data related to a

particular pet object of type *Pet*, whose schema is defined in lines 63-75 in Listing 1.

These two operations have a data dependency on the common field *petId*. The field is present in the output of *getPets* (lines 26 in Listing 1) and in the input of *getPetById* (lines 41 to 46 in Listing 1). Thus, the corresponding OPD shown in Figure 2 will have two nodes, one per each of these two operations, and the edge labeled *petId*, from *getPetById* to *getPets*.

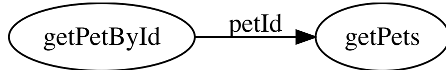

Figure 2. Sample of Operation Dependency Graph

The intuitive meaning of this graph is that to test the operation *getPetById* we require a valid value of *petId*, that would be difficult for a black-box testing framework to guess. So, the operation *getPets* should be tested earlier, to be able and fetch a valid value of *petId*.

The *Operation Dependency Graph* can grow quadratically with the number of edges, because, in the worst case, each pair of nodes is connected by a data dependency. In a complex REST API, with many dependent operations, the ODG will represent a valuable support to sort the operations to test and to plan for the acquisition of the needed data to be used during testing.

B. *Dependency Inference*

The example in Listing 1 represents the ideal case of a correctly defined Swagger. However, there is no syntax constraint that forces a developer to name data dependencies with exactly the same name. For instance, the field *petId* to be used in *getPetById*, could be simply called *id* and a developer would still understand what data to use for that field.

Alternatively, field names could contain typos (*petId* could become *pettID* or *pedID*), or characters in the wrong case (*petID* or *petid*). Thus, a perfect match among field names might fail. For this reason, the matching algorithm adopts a more solid algorithm that tolerates few typing mistakes.

In particular, these are the comparison operators used to match field names:

- **Case insensitive:** The comparison is case insensitive, to work around developer mistakes in using a consistent casing across operations;
- **Id completion:** When a field is just named *id*, we add a prefix to its name. In case this is a field of a structured object, the prefix is the name of the object. E.g., the field *id* of the object *pet* is renamed *petId*.
  In case this field is not part of a structured object, it is prefixed with the name of the operation in which it is involved, after removing get/set verbs from the operation name. For instance, the operation *getPet* becomes *Pet* after removing the verb "get", and it is used to change the field *id* into *petId*;
- **Stemming:** Instead of requiring two field names to be exactly the same, we tolerate some difference. We apply the Porter Stemming algorithm [19] to each parameter name to compare their *stem* instead of their exact names. For instance, two parameters named *pet* and *pets* are converted to the same word root *pet* and considered as the same parameter.

V. Testing of Nominal Cases

The aim of the *Nominal Tester* module is to automatically generate test cases meant to run nominal interactions, as they are documented in the Swagger. To achieve this objective, three sub-problems need to be solved, they are (i) deciding the order in which operations should be tested, (ii) generating input values, and (iii) deciding if input generation was successful or not.

A. *Operation Testing Order*

To elaborate the testing order among operations, we resort to two distinct dependencies, they are the CRUD semantic and the data dependencies from the OPD (see Section IV).

CRUD semantic. To successfully test an operation, a particular resource might be required to be in a certain state.

Considering the *CRUD* semantic, a successful *DELETE* operation requires that the target resource is present, so the resource should be first added using a *PUT* operation. A similar argument holds for a *POST* operation, meant to update an already existing resource. Conversely, a *PUT* operation that creates a resource requires the resource not to exist yet.

The dependencies related to the CRUD semantic are modeled based on the following priorities:

1) **HEAD** operation is the first in the priorities list, because its often used to check the validity of an API operation and to retrieve the header of a resource;
2) **POST** operation is commonly used to add a new resource, it has an higher priority since other operations may reference this novel operation and its parameter values;
3) **GET** operation retrieves information of an existing resource. Retrieved information can be used as input parameters for other operations;
4) **PUT** and **PATCH** are used to modify an existing resource with new parameter values;
5) **DELETE** is the operation with lowest priority because it deletes an existing resource that can not be referenced anymore by other operations.

Zhang et al. [21] also proposed a testing approach based on the CRUD semantic, using templates of pairs of related operations (e.g., POST+DELETE and POST+PUT), and this showed a positive impact on the test coverage.

ODP data dependencies. Data dependencies among operations are read from the OPG and used to sort the operations to test, in order to maximize the chances of reusing data already collected on previously tested operations, to test new operations.

The pseudo-code in Listing 2 shows the algorithm that we use to combine CRUD dependencies and ODG data dependencies, to decide the order to test operations.

```
1  while (length(graph.vertexes) > 0) {
2    # get graph leaves
3    to_test = graph.leaves
4
5    # random vertex if there is no leaf (only cycles)
6    if length(to_test == 0):
7        to_test = [random(graph.vertexes)]
8
9    # sort leaves according to CRUD
10   to_test_sorted = sort_for_crud(to_test)
11
12   # test each leaf
13   for (operation: to_test_sorted) {
14       is_tested = run_test_on(operation, n_fuzz)
15
16       # remove successfully tested leaves
17       if is_tested:
18         graph.remove(successful)
19   }
20
21   if (time > max_time)
22     break
23 }
```

Listing 2. Pseudo-code for ordering the operations to test.

The algorithm starts testing the graph's leaves, extracted with the query at line 3. Leaves are those operations with no outgoing edges, i.e. operations with no dependencies. No dependency means either no input fields (e.g., *getPets* in Listing 1) or input fields found in the output of no operations.

In case there is no leaf, it means that all the graph nodes have outgoing edges, and they are all involved in cycles. Cycles should be broken to start testing. In this case, at line 7, we randomly pick a node to start testing, i.e. we open the cycle at a random position.

If we only consider data dependencies, all these leaf nodes would have equivalent precedence, because they have no outgoing edges in the ODG. However, their order might still be optimized according to their CRUD semantic. Thus, at line 10, leaf nodes are then sorted according to their CRUD dependencies, if any, to increase the likelihood of testing an operation in the correct context, e.g. deleting or changing a resource after it has been created.

Then, at line 14, the algorithm attempts to test each leaf operation, in the sorted order. Each operation is fuzzed a fixed number of times n_{fuzz}, with different heuristics, to generate its input values (input generation heuristics are described in more detail in Section V-B). The value n_{fuzz} is a global constant set during the initialization of the tool. Multiple input generation attempts may be needed to guess valid input values. Our oracles (see Section V-C) are used to decide when a test can be considered successful.

We keep track whether the operation was successfully tested (in the local variable *is_tested*), because it might have provided useful data to test next operations. In case an operation is successfully tested, it is removed from the graph (at line 18). In this way, we remove from the graph those dependencies that are now satisfied, and new operations that only depended on the just tested operation will become leaf nodes in the edited graph. They will be tested in the next iterations of the algorithm.

The main loop (line 1) finishes when either the graph remains with no nodes, because all the operations are tested and testing is complete or when the time budget expires (line 21).

It is important to note that the operation order can not be precomputed off-line, but it should be updated at testing-time. In fact, we can mark a node as tested and remove it from the graph only after we could test it, and this is known only at testing time.

B. Input Value Generation

In order to test an operation, we need to guess appropriate input values. Input values are generated for each request using a probabilistic algorithm. With high probability (i.e., 80%) the algorithm applies a *response dictionary* based strategy, because reusing observed data is very likely to be effective in testing new operations. In the remaining cases, i.e. with 20% probability or when the former strategy fails, a new parameter value is generated starting from its schema.

Response dictionary. This heuristic is meant to reuse the knowledge of already tested operations to test new operations. Suppose for example that we need to test the operation *getPetsById* in the *PetStore API*. Our approach would need to have prior knowledge of which are the valid *pet ids* first.

For this reason, inspired by Ed-douibi et al. [7], we use the concept of *Response Dictionary*. The *Response Dictionary* is a map between field names and their observed values. For each operation that is successfully tested, the values of all the output fields that can be found in the response content are saved in the Response Dictionary, so that the values can be reused later when an input field with the same name is needed.

One of our extensions with respect by Ed-douibi et al. is represented by the matching algorithm that we use to look up into the *Response Dictionary*. While Ed-douibi et al. require an exact match between the name of the input value and the key in the dictionary, we tolerate some differences. In fact, our matching algorithm takes in input the field name for which a value is needed, and returns the key with the *closest* name among the keys in the dictionary. A non-perfect match is needed because of naming inconsistencies among input/output fields, due to implicit assumptions of

developers, or because of typing errors (as discussed in Section IV-B).

The name matching algorithm implements the following look-up strategy:

1) Look for a key with an exact match with the field name (e.g., *petId*);
2) Look for a key with an exact match with the concatenation of object name and field name (e.g., *petId* matches *pet+id*);
3) Look for a key with edit distance $< thr$ to the field name;
4) Look for a key with edit distance $< thr$ to the concatenation of object name and field name (e.g., *pets_Id* matches *pet+id*);
5) Look for a key that is a substring of the field name to find.

With lower probability and when the look-up strategy fails, the schema based field generation is applied.

Schema based field generation. A new parameter value is generated respecting the parameter schema type and its constraints.

Default and example values. Swagger supports an option to specify default values for input parameters and examples of values to be used. In case these values are specified, they can be attempted to test an operation.

Enum. When the type is enum, a value is randomly picked with uniform probability among the closed set of available values.

Random input generation. A random value is generated that matches the type of the input field. E.g., a random integer/decimal number or a random string. In particular, we give high probability (i.e., $p = 0.5$) of generating zero (on numeric input) or empty string (on string input). Otherwise, a numeric value if randomly picked with uniform probability from the allowed range. A random string is generated by concatenating a random number of alphanumeric characters respecting the parameter schema constraints *minLength and maxLength*.

C. Oracle

Our approach includes two oracles to assess if the automated test case generation is successful, based on the status code and on the match with the declared schema.

Status Code Oracle. The status code is a three digit integer value in the HTTP response, meant to describe the outcome of a request. A status code in the form 2xx stands for a correct execution, for instance 200 means OK, while 201 means that a resource has been successfully created.

A status code 4xx stands for an error that was correctly handled, e.g. 400 stands for *Bad Request* and 404 stands for *Not Found*. Conversely, a status code 5xx means that the server encountered an error that was not handled correctly, e.g. 500 means server crash.

We use the operation response status code as an oracle to assess if a test case was successfully created.

- *2xx.* When obtaining a status code 2xx in a response, we assume that our approach correctly guessed the input values to test an operation according to a nominal scenario. We conclude that this operation was successfully tested, so we mark this operation as tested and we can use the data in the response to populate the Response Dictionary;
- *4xx.* This status code means that testing was not successful. It could be due to two distinct reasons, either incorrect input values have been rejected by the server, or input values were correct and they exposed an implementation defect. However, from a black-box viewpoint, we can not tell which of the two cases applies. Conservatively, we assume that the correct input values have not been guessed. Thus, we discard this scenario and we continue test case generation;
- *5xx.* This status code means an internal server error, e.g. status code 500 means that server crash was not handled gracefully. A 5xx status code is probably due to a programming defect that should be fixed. So, this is an interesting scenario to document with a test case.

Our approach emits JUnit test cases for those interactions that cause status codes 2xx and 5xx, to let a developer replicate these scenarios.

Response Validation Oracle. The Swagger documents the operation responses and their schema, i.e. the intended status code and the fields in the responses, as in lines 47 to 59 of the example in Listing 1. Consistency between actual responses and their schema is important for remote programs that connect to a REST API. In fact, remote connection might fail in parsing inconsistent or malformed responses, and cause service discontinuity. Our second oracle reveals a mismatch between the intended response syntax (documented in the Swagger) and the actual response (observed at execution time), by using a schema validation library, namely *swagger-schema-validator*[5], on each response (either successful responses and error responses). In case of mismatch, a JUnit test case is emitted to document the defect.

For instance, considering a nominal execution of the operation *getPetById*, the operation response must contain an object of type *Pet*, as shown in the response example in Listing 3. This *Pet* object matches the *Pet* schema specified in lines 63 to 75 in Listing 1, i.e. three properties *id*, *name* and *tag* of the correct types.

```
1 {
2     "id": 1,
3     "name": "doggy",
4     "tag": "dog"
5 }
```

Listing 3. Example of *getPets* response compliant with the schema.

[5]https://github.com/bjansen/swagger-schema-validator

Listing 4 shows, instead, a case where the response content does not match the schema. Indeed, the object *Pet* in this example misses the parameter *name*. This particular execution output is classified as an implementation error by the oracle.

```
1 {
2     "id": 1
3     "tag": "dog"
4 }
```

Listing 4. Example of *getPets* response NOT compliant with the schema.

VI. Testing of Error Cases

The objective of the *Error Tester* module is to test the exceptional scenarios of the REST API, to assess if the REST API handles wrong requests in the appropriate way, i.e. they are discarded and errors end gracefully. To this aim, this module starts from nominal executions and mutate them to turn them malformed and inconsistent. In case a mutated request is not discarded or if it causes a fatal error, a defect is detected.

A. Mutation Operators

Nominal test cases are changed according to a catalog of mutation operators. Currently the subsequent mutation operators are available:

- *Missing required.* In Swagger, input parameters support the modifier *required*, which means that a field is mandatory. A request that misses a required field should be discarded by the REST API. This mutation consists in altering a request by removing a required input field;
- *Wrong input type.* Input parameters are strongly typed. This mutation alters an existing test case by changing the value of an input parameter such that its type becomes wrong. In case the declared type is string, the new value is a random number (integer or float). In case the declared type is a numeric, the new value is a random string. In case of type *enum*, a random value is picked that is different than the set of values in the enumeration.
- *Constraint violation.* The swagger can also specify additional constraints for strings (e.g., *minLength* and *maxLength*) and for numeric values (e.g., *min* and *max* values). This mutation operator edits the value in a request such that it violates a constraint. For instance, a string is trimmed or extended with a random suffix, or a numeric value is decreased/increased by a random delta until it exceeds the limits.

A nominal test is mutated many times, by applying each mutation operation once per each input field in the request. Mutated requests are then sent to the REST API and the response is analyzed by the oracles.

B. Oracle

Similarly to the Nominal Tester, also the Error Tester is supported by two oracles.

Status Code Oracle. The status code in the response is inspected to classify whether this test case reveals a programming defect:

- *2xx.* This status code stands for a correct execution. This means that incorrect inputs, that should have been rejected, are instead processed as valid. This scenario exposes a mismatch between the features declared in the Swagger and those actually implemented;
- *4xx.* This status code, instead, means that wrong inputs have been correctly detected and the request caused a graceful error. This is the expected execution on malformed requests;
- *5xx.* An internal server error exposed thanks to malformed input data. This is clearly a defect.

A JUnit test case is emitted for those error scenarios that cause status codes 2xx and 5xx.

VII. Experimental Validation

In this section, we provide an experimental validation of RESTTestGen, with respect to its effectiveness of revealing programming defects in real world REST APIs. The complete package to replicate our experiment is available online[6].

A. Research Questions

To define our empirical investigation, we formulate the following research questions:

- RQ_N: Is the *Nominal Tester* module effective in generating test cases?
- RQ_E: Is the *Error Tester* module effective in generating test cases?

The first research question RQ_N is intended to investigate if the *Nominal Tester* module is capable of testing the nominal scenarios of REST APIs, as they are documented in the Swagger file.

The second research question RQ_E instead focuses on the *Error Tester* module, and it is meant to investigate if is able expose failure in handling incorrect inputs, resulting from mutations of nominal test cases.

B. Case Studies

We applied RESTTestGen to an extensive group of REST APIs. We considered the REST APIs listed in the website *API.guru*[7] on 18 June 2019. For each REST API, this website also provides the corresponding Swagger specification. However, we had to apply some sanity checks

[6]RESTTestGen: Automated Black-Box Testing of RESTful APIs https://github.com/resttestgenicst2020/submission%5ficst2020

[7]https://apis.guru/browse-apis/

and filtering, to select case studies that are appropriate for assessing fully automated test case generation.

First of all we probed REST APIs to filter out those that were not responding, which were probably discontinued or just temporarily down.

Then, we also excluded all those that declared to require authentication, because they would have required a substantial manual effort to create a distinct account for each distinct REST API[8].

Eventually, we manually sent some probe requests to the remaining services to dynamically verify them, before finalizing the case studies of our experiment. We had to further exclude some REST APIs because, despite their Swagger did not mention authentication, their actual implementation did require it.

After filtering, the final case studies used in our empirical evaluation consists of 87 REST APIs, for a total of 2,612 operations, which means 30 operations per REST API on average.

C. Experimental Procedure

Based on the research questions formulated above, we defined these settings of our experiment.

RESTTESTGEN is run on all the Swagger files for the 87 case studies. For each Swagger the ODG is computed and then the Nominal Tester module is run, with a maximum time budget of 30 minutes per case study. In the result section, we will see that this time budget will prove to be appropriate, in fact our algorithm completes in less than 10 minutes in the majority of the case studies. During this time budget, the maximum number of sent requests is 13,944, with a mean of 162 requests per case study. However, the amount of time taken to test a service is strongly dependent on how long the tested service takes to respond.

The nominal test cases that obtained a response status code 2xx were then mutated by the Error Tester module, to try and automatically generate error test cases. This second module was also assigned a maximum time budget of 30 minutes per case study.

The number of attempts n_{fuzz} to fuzz each operation (see Section V-A) is set to 5. The edit distance threshold thr used in the Response Dictionary (see Section V-B) is set to 1.

All these experiments have been run on a 8 cores desktop PC, equipped with Intel(R) Core(TM) i7 870 running at 2.93GHz CPU, and with 8GB of RAM.

D. RQ$_N$: Nominal Tester

The results of Nominal Tester module on the set of case studies are shown in Table I. As we can see, all the 87 case studies (first line) have been subject to automated test case generation and for 62 of them (second line) at least one test for a nominal execution scenario (status code 2xx) could be automatically generated. For 20 case studies, the test case automatically generated by RESTTESTGEN exposed errors that were not properly handled by the REST API (status code 5xx). On 66 case studies, invalid response messages were observed, they are responses inconsistent with their schema defined in the Swagger.

[8]Some REST APIs support authentication with third party identity providers via O-Auth (e.g., login with Facebook or with Google accounts). This feature is supposed to simplify end-user authentication, but not the authentication of a program that is willing to access via API, that sill needs to (manually apply and) acquire an API-key to authenticate to the REST API.

Table I
CASE STUDIES TESTED BY THE NOMINAL TESTER.

Total APIs	87
APIs with status code 2xx	62
APIs with status code 5xx	20
APIs with validation error	66

Table II
OPERATIONS TESTED BY THE NOMINAL TESTER.

Total operations	2,612
Tested operations	2,560
Operations with status code 2xx	625
Operations with status code 5xx	151
Operations with validation errors	1,733

Table II shows a more detailed perspective, focusing on the case study operations. Among the total 2,612 operations, for 2,560 of them test cases could be generated by RESTTESTGEN. The untested operations are due to some failure of the tool, such as unsupported input parameter generation (e.g., files to be uploaded).

In particular, automatically generated test cases found a nominal execution for 625 of them (status code 2xx) and an ungraceful error for 151 (status code 5xx).

For 435 test cases the status code was 4xx, but they are not shown in the table, because they were hard to classify with a black-box access. In fact, they might be graceful errors due to programming defects, or just rejected requests due to failures by RESTTESTGEN in generating appropriate inputs.

Test cases with validation errors are still a majority: in 1,733 tests the response did not match the declared schema.

Considering these results, we can formulate the following answer to RQ$_N$:

The Nominal Tester module of RESTTESTGEN *is effective in automatically generating test cases with black-box access, because it was able to test 2,560 operations out of 2,612 operations on real world REST APIs. These tests exposed 151 faults in the form of not correctly handled internal errors and 1,733 inconsistent response messages.*

Table III
TEST CASES AUTOMATICALLY GENERATED BY THE ERROR TESTER.

Mutation operator	Mutants	Status code 2xx	Status code 5xx
Missing required	459	283	7
Wrong input type	707	511	16
Constraint violation	119	68	11
Total	**1,285**	**864**	**23**

E. RQ_E: Error Tester

Subsequently, the 625 nominal test cases with a status code 2xx have been subject to mutation by the Error Tester module, and executed on the case studies. Results of this second module are shown in Table III. For each mutation operator (first column) the table reports in the second column how many mutants (i.e., mutated test cases) could be generated.

The number of mutants is different across mutations, because different mutations impose different applicability precondition. For instance, mutation *Missing required* needs an input field with the *required* modifier. In case this modifier in not present, the mutation does not apply. The largest amount of mutants (i.e., 707 tests) could be generated by *Wrong input type*, because it is the mutation with the most simple preconditions, i.e. a field of type string, numeric or enum. Then, mutation *Missing required* generated 459 mutants and, eventually, mutation *Constraint violation* generated only 119 mutants, because only few case studies specify value constraints in their Swagger. In total 1,285 mutants have been generated.

These mutated test cases are then executed on the case studies and the response status codes are evaluated to assert the presence of programming defects. Table III reports how many mutants are still processed as valid input with status code 2xx (third column) and how many exposed an unhandled error with status code 5xx (fourth column). The majority of defects (864 cases) have been recorded for status code 2xx, they are wrong data that are still handled as correct. Only 23 cases could expose server errors as status codes 5xx. It should be noted that these 23 cases of status code 5xx are different and additional with respect to the 151 cases observed in the previous experiment. In fact, the test cases generated by the Nominal Tester and by the Error Tester are not overlapping, i.e. the former module relies on inputs whose type and value match the Swagger, while the latter module relies on inputs that violate the Swagger.

Given these data, we can answer to RQ_E as follows:

> *The Error Tester module of* RESTTESTGEN *is effective in automatically generating error test cases with black-box access of real world REST APIs, because they revealed 864 cases where wrong data are accepted as valid, and 23 cases with unhandled errors.*

F. Threats to Validity

There is a number of limitations that could potentially limit the validity of our empirical results.

Black-box access. With the aim of considering realistic case studies, we meant to involve real and existing REST APIs, hosted by their respective owners. As such, we could not inspect the case study source code to manually validate the results of automated testing, with respect to the correct classification reported by our oracles. Moreover, we could not measure the code coverage achieved by automatically generated tests. However, for the second oracle, the classification was quite objective, because it revealed a defect whenever the response was inconsistent with the documented schema.

Oracle based on status code. Using the status code to assess the result of automated testing might represent a threat. In fact, while the classification of tests with correct executions (status codes 2xx) and with *unhandled* errors (status code 5xx) is more objective, the classification of tests with *handled* errors (status codes 4xx) is more dubious. In fact, handled errors might occur either because of defects in the REST API implementation, or because of limitation of our tool that caused wrong input values to be used in test cases. Since we could not accurately classify execution with 4xx status codes, we conservatively assumed them to be due to failures by RESTTESTGEN.

Most of the defects detected by the *Error tester* are incorrect data accepted as valid (status code 2xx). This result highlights a potential limitation of the *Status Code Oracle* used in the *Nominal Tester*. This oracle might have incorrectly classified a test case run as a *pass* just because of its status code 2xx, that was also observed in case of invalid input values.

Missing authentication. To increase the number of REST APIs in our experiment, we decided to minimize the manual effort required to prepare each case study. Since account creation would have required a substantial effort and, sometimes, interaction with the REST API owner, e.g. to motivate why API access is required, we tested only those case studies that did not require account creation. This might represent a threat to the validity of our results, because sensitive operations might be forbidden to anonymous users (e.g., delete a resource) and the experiment might be limited to less crucial operations that, thus, are considered of lower importance by a case study owner. Nonetheless, the final case studies include a reasonably large number of REST APIs (consistent with related work [1], [7], [16]), that could be tested to a large extent.

Nondeterminism. RESTTESTGEN was executed only once per case study. However, since our algorithm contains non-deterministic components, e.g. in fuzzing input parameters, a more accurate experimental setting should have included multiple executions (e.g. 10-30 repetitions). We plan

to conduct a more extensive and complete experimentation, with more case studies and with more executions per each case study, as part of our future work.

VIII. Related Work

The specificity of REST APIs attracted the attention of the testing research community only recently. Novel approaches to test REST APIs have been proposed that can be divided in black-box approaches and white-box approaches.

A white-box perspective in automated testing relies on the availability of API source code to perform static analysis, or to instrument it to collect execution traces and metric values. A black-box approach, instead, does not require any source code, which is often the case when using closed source components and libraries. However, a black-box access to the REST API lacks much information potentially useful for the automatic test case generation.

The most related work is, probably, by Ed-douibi et al. [7]. They proposed a model-based approach for black-box automatic test case generation of REST APIs. A model is extracted from the Swagger/OpenAPI specification of a REST API, to generate both nominal test cases (with input values that match the model) and faulty test cases (with input values that violate the model). However, they do not explicitly model the dependencies among operations, while we define the *Operation Dependency Graph* to this aim. Moreover, we dynamically update this graph to decide the most appropriate operation for the next test. Additionally, we integrate the response dictionary in a series of heuristics to automate input data generation.

Another limitation of their approach is that it only applied to read-only operations, called *safe* operations by the authors, because they meant to avoid operations with side-effect on the API state. Conversely, our approach explicitly models side effect of operations (i.e. the CRUD semantics, see Section V-A) and exploits them to decide the order in which to test operations.

Similarly to our approach, also Atlidakis at al. [2] model the dependencies among the operations in a REST API to elaborate an appropriate ordering. However, while they use dependencies to pre-compute the order to test operations (e.g., using Breadth-first search or random walk), we propose to compute the next operations to test dynamically, based on the outcome of the operations that could be tested so far.

Segura et al. [16] proposed a complete different black-box approach, where the oracle is based on metamorphic relations among requests and responses. For instance, they send two queries to the same REST API, where the second query has stricter conditions than the first one (e.g., by adding an additional constraint). The result of the second query should be a proper subset of entries in the result of the first query. When the result is not a sub-set, the oracle revealed a defect. However, this approach only works for search-oriented APIs. Moreover, this technique is only partially automatic, because the user is supposed to manually identify the metamorphic relation to exploit and what input parameters to test.

White-box approaches are complementary to ours, because they assume to have access to the code of the API to test. Arcuri [1] proposed a fully automated white-box testing approach, to generate test cases with evolutionary algorithms. Similarly to ours, Arcuri's approach requires the API specification (i.e., the Swagger). Differently than us, his approach also requires access to the Java bytecode of the REST API to test. In fact, the objective is to achieve high code coverage. This approach has been implemented and available as a tool prototype called EvoMaster.

Many approaches have been proposed so far to test Web-services, based on their WSLD specification [3], [10]–[14], [17], [18], [20]. An extensive survey of techniques for automated testing of Web-services has been conducted and reported by Bozkurt et al. [4] and by Canfora et al. [5], [6].

Despite similar objectives, Web-services and REST APIs are conceived on top of different interaction models. Web-services are mostly based on SOAP [9], a message oriented model (mainly meant to overcome limitations of previous solutions, such as CORBA, Java/RMI, DCOM), while REST APIs rely on the concept of *web resources* accessible through stateless operations [8].

IX. Conclusion

This paper presents RESTTESTGEN, a novel approach for automatically generating black-box test cases for REST APIs. The Operation Dependency Graph is defined to model data dependencies among the operations in a REST API. This allows our approach to dynamically decide in which order to test operations, such that the input data required to test an operation are available from the output data of those already tested.

Two distinct testing modules are presented, the Nominal Tester and the Error Tester, to automatically generate test cases related to nominal execution scenarios and to error management scenarios. The empirical assessment showed that the proposed approach is effective in testing real worlds REST APIs, and in detecting a considerable amount of implementation defects.

As future work, we plan to extend the testing capability of RESTTESTGEN to try and assess the presence of security defects in the implementation of REST APIs. This would require to attempt black-box proof-of-concept attacks and to define a brand new oracle, capable to detecting successful attacks. Additionally, we plan to the extend our implementation to support authentication, to perform a wider empirical validation, possibly including more complex and critical REST APIs, such as those related to the FinTech domain.

REFERENCES

[1] A. Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):3, 2019.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.

[3] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. Wsdl-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*, pages 207–212. IEEE, 2005.

[4] M. Bozkurt, M. Harman, and Y. Hassoun. Testing web services : A survey. 2011.

[5] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *It Professional*, 8(2):10–17, 2006.

[6] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, pages 78–105. Springer, 2007.

[7] 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 (EDOC)*, pages 181–190. IEEE, 2018.

[8] R. T. Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[9] M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, and M. Gudgin. Soap version 1.2 part 1: Messaging framework. *W3C REC REC-soap12-part1-20030624, June*, pages 240–8491, 2003.

[10] S. Hanna and M. Munro. Fault-based web services testing. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 471–476. IEEE, 2008.

[11] Y. Li, Z.-a. Sun, and J.-Y. Fang. Generating an automated test suite by variable strength combinatorial testing for web services. *Journal of computing and information technology*, 24(3):271–282, 2016.

[12] C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai. Wsdl-based automated test data generation for web service. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 731–737. IEEE, 2008.

[13] E. Martin, S. Basu, and T. Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*, 2006.

[14] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[15] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*, pages 137–146, 2016.

[16] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Corts. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2017.

[17] H. M. Sneed and S. Huang. Wsdltest-a tool for testing web services. In *2006 Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*, pages 14–21. IEEE, 2006.

[18] W.-T. Tsai, R. Paul, W. Song, and Z. Cao. Coyote: An xml-based framework for web services testing. In *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, pages 173–174. IEEE, 2002.

[19] P. Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[20] W. Xu, J. Offutt, and J. Luo. Testing web services by xml perturbation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10–pp. IEEE, 2005.

[21] M. Zhang, B. Marculescu, and A. Arcuri. Resource-based test case generation for restful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 1426–1434, New York, NY, USA, 2019. ACM.