2018 IEEE 22nd International Enterprise Distributed Object Computing Conference

Automatic Generation of Test Cases for REST APIs: a Specification-Based Approach

Hamza Ed-douibi
Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya(UOC)
Barcelona, Spain
hed-douibi@uoc.edu

Javier Luis Cánovas Izquierdo
Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC)
Barcelona, Spain
jcanovasi@uoc.edu

Jordi Cabot
ICREA
Universitat Oberta de Catalunya (UOC)
Barcelona, Spain
jordi.cabot@icrea.cat

***Abstract*—The REpresentation State Transfer (REST) has gained momentum as the preferred technique to design Web APIs. REST allows building loosely coupled systems by relying on HTTP and the Web-friendly format JSON. However, REST is not backed by any standard or specification to describe how to create/consume REST APIs, thus creating new challenges for their integration, testing and verification. To face this situation, several specification formats have been proposed (e.g., OpenAPI, RAML, and API Blueprint), which can help automate tasks in REST API development (e.g., testing) and consumption (e.g., SDKs generation).**

In this paper we focus on automated REST API testing relying on API specifications, and particularly the OpenAPI one. We propose an approach to generate specification-based test cases for REST APIs to make sure that such APIs meet the requirements defined in their specifications. We provide a proof-of-concept tool implementing our approach, which we have validated with 91 OpenAPI definitions. Our experiments show that the generated test cases cover on average 76.5% of the elements included in the OpenAPI definitions. Furthermore, our experiments also reveal that 40% of the tested APIs fail.

***Index Terms*—API testing, REST APIs, OpenAPI**

I. Introduction

The REpresentation State Transfer (REST) has become the preferred technique when designing Web APIs. REST outlines the architectural principles, properties, and constraints to build Internet-scale distributed hypermedia systems [1]. However REST is a design paradigm and does not rely on any kind of standards to describe REST APIs, which makes REST API development, testing and integration a hard task.

The Web Application Description Language (WADL) [2] was proposed to describe REST APIs but failed to attract adoption due to its complexity and limited support to fully describe REST APIs, thus most of the REST APIs were described using informal text [3]. This situation triggered the creation of other formats, for instance, Swagger, API Blueprint, and RAML, just to cite few, which makes choosing a format or another subjective to the API providers. To face this situation, the OpenAPI Initiative[1] (OAI) was launched with the objective of creating a vendor neutral, portable, and open specification for describing REST APIs. OAI has succeeded in attracting major companies (e.g., Google, Microsoft or IBM) and the OpenAPI specification has become the choice of reference, thus increasing its adoption. For instance, *APIs.guru*[2], a repository of OpenAPI definitions, lists more than 800 APIs.

REST API specifications can also help automate different developments tasks such as generation of SDKs and API testing. This paper focuses on API testing. In fact, testing Web APIs and specially REST ones is a complex task [4]. Many approaches have proposed specification-based test cases generation but mostly for SOAP Web APIs by relying on their Web Services Description Language (WSDL) documents (e.g., [5], [6], [7], [8]). However, works targeting test cases generation for REST APIs are rather limited (e.g., [9]). On the other hand, we have witnessed an increasing number of open source and commercial tools offering automated API testing for different specification formats such as Swagger/OpenAPI (e.g., Ready API![3], Dredd[4], Runscope[5]) and API Blueprint (e.g., Dredd, Apiary[6]). Nevertheless, these tools only cover nominal test cases (i.e., using correct data) and neglect the fault-based ones (i.e., using incorrect data) which are important when selecting reliable Web APIs for composition [4]. Furthermore, they require extra configuration and the provision of input data.

In this paper we propose an approach to generate test cases for REST APIs relying on their specifications, in particular the OpenAPI one, with the goal to ensure a high coverage level for both nominal and fault-based test cases. We define a set of parameter inference rules to generate the input data required by the test cases. We follow a metamodelling approach, thus favoring reuse and automation of test case generation. Therefore, we define a TestSuite metamodel to represent test case definitions for REST APIs. Models conforming to this metamodel are created from REST API definitions, and later used to generate the executable code to test the API. We illustrate our approach using the OpenAPI specification but a similar approach could be followed for other specifications. We provide a proof-of-concept tool implementing our approach, which we have validated with 91 OpenAPI definitions

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

[2]https://apis.guru/openapi-directory/
[3]https://smartbear.com/product/ready-api/overview/
[4]http://dredd.readthedocs.io/
[5]https://www.runscope.com/
[6]https://apiary.io/

2325-6362/18/$31.00 ©2018 IEEE
DOI 10.1109/EDOC.2018.00031

from *APIs.guru*. Our experiments show that the generated test cases cover on average 76.5% of the elements included in the OpenAPI definitions (i.e., operations, parameters, endpoints, data definitions). Furthermore, our experiments also reveal that on average 40% of the tested APIs fail (either in the definition or implementation).

The rest of the paper is organized as follows. Sections II and III present the background and related work, respectively. Section IV describes our approach. Sections V, VI, VII, and VIII present the different steps of the test case generation process for the OpenAPI specification, namely: extracting OpenAPI models, inferring parameter values, generating test case definitions, and generating executable code, respectively. Section IX describes the tool support. Section X presents the validation process, and finally, Section XI concludes the paper and presents the future work.

II. Background

In this section we present some of the concepts and technologies used as the basis of this work, namely: REST APIs, the OpenAPI specification, and specification-based API testing.

A. REST APIs

REST defines a set of constraints which describe the Web's architectural style. It has been introduced by Roy Fielding in 2000 [1] and is commonly applied to the design of modern Web APIs. A Web API conforming to the REST architectural style is referred to as a REST API.

REST architectural style consists of a set constraints to address separation of concerns, visibility, reliability, scalability and performance. Particularly, REST describes six constraints, namely: *Client-Server*, *Stateless*, *Cache*, *Layered System*, *Code on Demand* and *Uniform Interface*. In general, Web APIs consider the constraints: *Client-Server*, *Stateless*, *Cache*, and *Uniform Interface*. *Layered System* is more important to the deployment of Web APIs than to their design. On the other hand, *Code on Demand* is not applied to Web APIs [10].

Uniform Interface is a fundamental constraint in the design of REST APIs. It describes four interface constraints, namely: (1) *identification of resources* (i.e., use Uniform Resource Identifier (URI) to identify resources); (2) *manipulation of resources through representations* (i.e., representations are transferred between REST components to manipulate resources); (3) *self-descriptive messages* (i.e., enable intermediate processing by constraining messages to be self-descriptive); and, (4) *hypermedia as the engine of application state* (i.e., Resources link to each other in their representations using hypermedia links and forms).

Although REST is protocol-agnostic, in practice most REST APIs use HTTP as a transport layer.Therefore, HTTP's uniform interface is generally applied to design REST APIs. Ideally, REST APIs map HTTP methods with CRUD operations (i.e., Create, Read, Update, Delete) as follows: (1) `GET` to retrieve a resource, (2) `PUT` to update a resource, (3) `POST` to create a resource, and (4) `DELETE` to remove a resource.

Listing 1. Excerpt of the OpenAPI definition of Petstore.

```
{"swagger"  : "2.0",
 "host"     : "petstore.swagger.io",
 "basePath" : "/v2",
 "paths"    : {
   "/pet/findByStatus" : {
    "get" : {
      "operationId" : "findPetsByStatus",
      "produces"    : ["application/xml","application/
          json"],
      "parameters"  : [
        { "name"      : "status",
          "in"   : "query",
          "required"  : true,
          "type" : "array",
          "items"     : {
            "type"    : "string",
            "enum"    : ["available", "pending", "sold"],
            "default" : "available" },
          "collectionFormat" : "multi" }
      ],
      "responses":{
         "200": { "schema" :
                  { "type"  : "array",
                    "items" : {"$ref" : "#/definitions/
                        Pet"}}}
      }}},
   "/pet/{petId}" : {
    "get" : {
      "operationId" : "getPetById",
      "parameters"  : [
        { "name"      : "petId",
          "in"   : "path",
          "required"  : true,
          "type" : "integer",
          ...}
      ],...}}},
  ...
  "definitions":{
    "Pet":{
      "type" : "object",
      "required" : ["name", "photoUrls"],
      "properties" : {
        "id" : { "type" : "integer",
                 "format":"int64" }, ...
                 }, ...}}
    }
```

In practice, most APIs do not fully conform to Fielding's definition of REST [11]. Assessing whether an API is *RESTful* (i.e., compliant with REST principles) is out of the scope of this paper. However, we rely on REST principles and the semantics of HTTP/1.1[7] to test REST APIs. In the following we call a REST API any Web API that uses HTTP methods to facilitate communication between a client and server.

B. The OpenAPI specification

The OpenAPI specification (formerly known as the Swagger Specification) allows defining the resources and operations of a REST API, either in JSON or YAML. We will illustrate our approach using the OpenAPI specification v2.0, which is still the mostly used to describe REST APIs even if OpenAPI v3 has recently been released. Listing 1 shows an overview of the OpenAPI definition of Petstore, an illustrative REST API released by the OAI, in JSON format. Petstore allows managing pets (e.g., add/find/delete pets), orders (e.g., place/delete orders), and users (e.g., create/delete users). The `paths` object includes relative paths to individual endpoints. Each

[7]http://www.rfc-editor.org/rfc/rfc2616.txt

path item (e.g., `/pet/findByStatus`) includes operations using HTTP methods (e.g `get` for `/pet/findByStatus`), which in turn, include properties such as their parameters (e.g., `status`), response status codes(e.g, `200`), and the media types they consume/produce (e.g, `application/json`). The definition includes also JSON schemas defining the data types used in the parameters/responses of the operations (e.g., `Pet`). When needed, we will refer to this example to explain our approach.

C. Specification-based API Testing

API testing is a type of software testing that aims to validate the expectations of an API in terms of functionality, reliability, performance, and security. *Specification-based API testing* is the verification of the API using the available functionalities defined in a specification document [4]. In specification-based REST API testing, test cases consist of sending requests over HTTP/S and validating that the server responses conform to the specification, such as the OpenAPI one. For instance, a test case could be created for the operation *findPetsByStatus* of the Petstore example (see Listing 1) by sending a GET request to *http://petstore.swagger.io/v2/pet/findByStatus?status=available* and expecting a response having the HTTP status code `200` and a JSON array conforming the schema defined in the specification. Automated specification-based API testing involves generating such test cases.

Fault-based testing is a special type of testing which aims to demonstrate the absence of *prespecified* faults [12]. Instead of detecting unknown faults, the aim of fault-based test cases is to prove that known faults do not exist, thus promoting reliability. For instance, the parameter *status* of the operation *findpetsByStatus* is of type enumeration and restricted to the values: `available`, `pending`, and `sold`. Therefore, a test case could be created for the operation *findPetsByStatus* using an incorrect value for the parameter *status* by sending a GET request to *http://petstore.swagger.io/v2/pet/findByStatus?status=test* and expecting a response having the status code `400 BAD REQUEST`.

III. Related Work

Many approaches proposed both nominal (e.g., [5], [6], [7], [8]) and faulty (e.g., [13], [14]) specification-based test cases generation for the classical SOAP Web APIs by relying on their WSDL definitions. Research works targeting test cases generation for REST APIs, on the other hand, are relatively limited. For instance, Fertig et al. [15], Chakrabarti and Kumar [16], and Benac Earle et al. [17] propose different approaches for automated test case generation, but relying on manual definition of a model, a test specification DSL, and a JSON schema, respectively, thus making their approaches bound to their ad-hoc specifications. Arcuri [9] proposes an approach to generate white-box integration test cases for REST APIs using search algorithms and relying on the OpenAPI specification. However, in such scenario, the developer must have full access to the code of the API to generate the test cases. On the other hand, our approach does not require providing any model or definition other than OpenAPI.

Beyond the research context, several commercial and open source tools provide testing facilities for REST APIs. Some of these tools propose automated testing for REST API specifications, such as the OpenAPI specification. These tools can be divided into three categories, namely: (1) integrated environments, (2) command line tools, and (3) development frameworks. The integrated environment testing tools provide a GUI which can be used to create and run test cases (e.g., SoapUI/ReadyAPI![8], Runscope, or APIFrotress[9]). Command line tools propose a command line interface for configuring and running the test cases (e.g., Dredd or got-swag[10]). Finally, development frameworks provide testing facilities for different programming languages (e.g., Hippie-Swagger[11], Swagger Request Validator[12], or Swagger Tester[13]). Certainly, these tools prove the importance of specification-based testing for REST APIs, but they all require a considerable effort to configure them and to provide input data. Furthermore, these tools do not provide fault-based testing out the box.

IV. Our Approach

We define an approach to automate specification-based REST API testing, which we illustrate using the OpenAPI specification, as shown in Figure 1. Our approach relies on model-based techniques to promote the reuse and facilitate the automation of the generation process. OpenAPI definitions are represented as models conforming to the OpenAPI metamodel, while test case definitions are stored as models conforming to the TestSuite metamodel. The OpenAPI metamodel allows creating models conforming to the OpenAPI specification. This metamodel was previously introduced in the work by Ed-douibi et al. [18], where an approach to discover OpenAPI models was also presented. On the other hand, the TestSuite metamodel allows defining test case definitions for REST API specifications. This metamodel is inspired by the best practices in testing REST APIs.

Our approach has four steps. The first step extracts an OpenAPI model from the definition document of a REST API, by parsing and processing the JSON (or YAML) file. The second step extends the created OpenAPI model to add parameter examples, which will be used as input data in the test cases. The third step generates a TestSuite model from the OpenAPI model by inferring test case definitions for the API operations. Finally the last step transforms the TestSuite model into executable code (JUnit in our case). In the following sections, we explain each step in detail.

[8] https://www.soapui.org/
[9] http://apifortress.com/
[10] https://github.com/mobilcom-debitel/got-swag
[11] https://github.com/CacheControl/hippie-swagger
[12] https://bitbucket.org/atlassian/swagger-request-validator
[13] https://swagger-tester.readthedocs.io/en/latest/

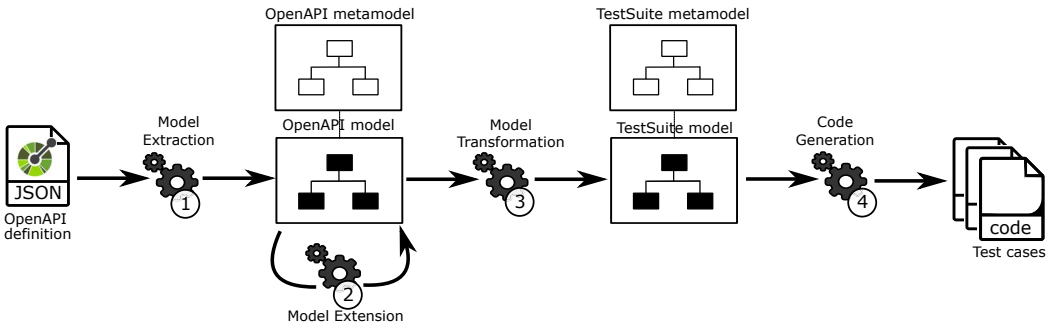

Fig. 1. Our approach for OpenAPI.

V. Extracting OpenAPI Models

In this section we describe the first step of our approach. We first briefly introduce the OpenAPI metamodel and then describe the extraction process.

A. The OpenAPI Metamodel

Figure 2 shows an excerpt of the OpenAPI metamodel, which was introducted in a previous work by Ed-douibi et al. [18]. For the sake of simplicity, we only describe the details needed in the scope of this paper but more details can be found at the paper cited above.

An OpenAPI definition is represented by the `API` element which includes, among others, a host, a base path, a set of paths and data type definitions. A data type definition is represented by the `Schema` element, which includes the details of a data type as described in the OpenAPI specification. An API endpoint is represented by the `Path` element. It includes a relative path (i.e., `pattern` attribute) and a set of operations (i.e., the references `get`, `post`, `put`, `delete`). An API operation is represented by the `Operation` element which includes, among others, an identifier, a set of parameters (i.e., `parameters` reference), and a set of responses. An operation parameter is represented by the `Parameter` element which includes a name, the location in the REST request (i.e., `parameterLocation` attribute, e.g., `query`, `body`), a type (e.g., `integer`, `string`), the data structure definition of the parameter (i.e., `schema` reference) if such parameter is located in the body of the request, a set of constraints inherited from `JSONSchemaSubset` (e.g., maximum, enumeration), and an example[14]. The `Parameter` element includes also constraints defined as attributes (e.g., the `required` attribute) or inherited from the `JSONSchemaSubset` element (e.g, `maximum` attribute). An operation response is represented by the `Response` element which includes the HTTP status code, a data type definition (i.e., `schema` reference), and a set of examples and headers.

B. Extraction Process

Creating OpenAPI models from JSON (or YAML) OpenAPI definitions is rather straightforward, as our metamodel mirrors the structure of the OpenAPI specification. Only special attention had to be paid to resolve JSON references (e.g. `#/definitions/Pet`). Thus, the root object of the JSON file is transformed to an instance of the `API` model element, then each JSON field is transformed to its corresponding model element. Figure 3 shows an excerpt of the generated OpenAPI model of the Petstore API including the *findPetsByStatus* operation and a partial representation of the `Pet` schema. Note that this model also includes an inferred value for the parameter *status* (i.e., `example = available`). We present the inference rules in the next section.

VI. Inferring Parameter Values

The goal of this step is to enrich OpenAPI models with the parameter values needed to generate test cases for an operation. Definition 1 describes a testable operation and PR 1, PR 2, and PR 3 are the applied rules to infer parameter values ordered by preference.

Definition 1 (Testable Operation)**.** An API operation can be tested if all the values of its required parameters can be inferred.

PR 1 (Simple parameter value inference)**.** A value of a parameter `p` could be inferred from: (1) examples (i.e., `p.example` and `p.schema.example`), (2) default values (i.e., `p.default` or `p.items.default` if `p` of type array) or (3) enums (i.e., the first value of `p.enum` or `p.items.enum` if `p` is of type array).

PR 2 (Dummy parameter value inference)**.** A value of a parameter `p` could be set to a dummy value (respecting the type) if a request to the operation of `p` including that value returns a successful response (i.e., a `2xx` response code class).

PR 3 (Complex parameter value inference)**.** A value of a parameter `p` could be inferred from the response of an operation `o` if: (1) `o` is testable; (2) `o` returns a successful response `r`; and (3) `r.schema` contains a property matching `p`[15].

The previous rules are applied in sequence to infer a parameter value. For instance, PR 1 was applied to the

[14]As the OpenAPI specification does not define parameter examples, the `x-example` vendor extension is commonly used to provide examples.

[15]Some basic heuristics are applied in the parameter matching process (e.g., *petId* parameter is inferred from the *id* property of the schema *Pet*.)

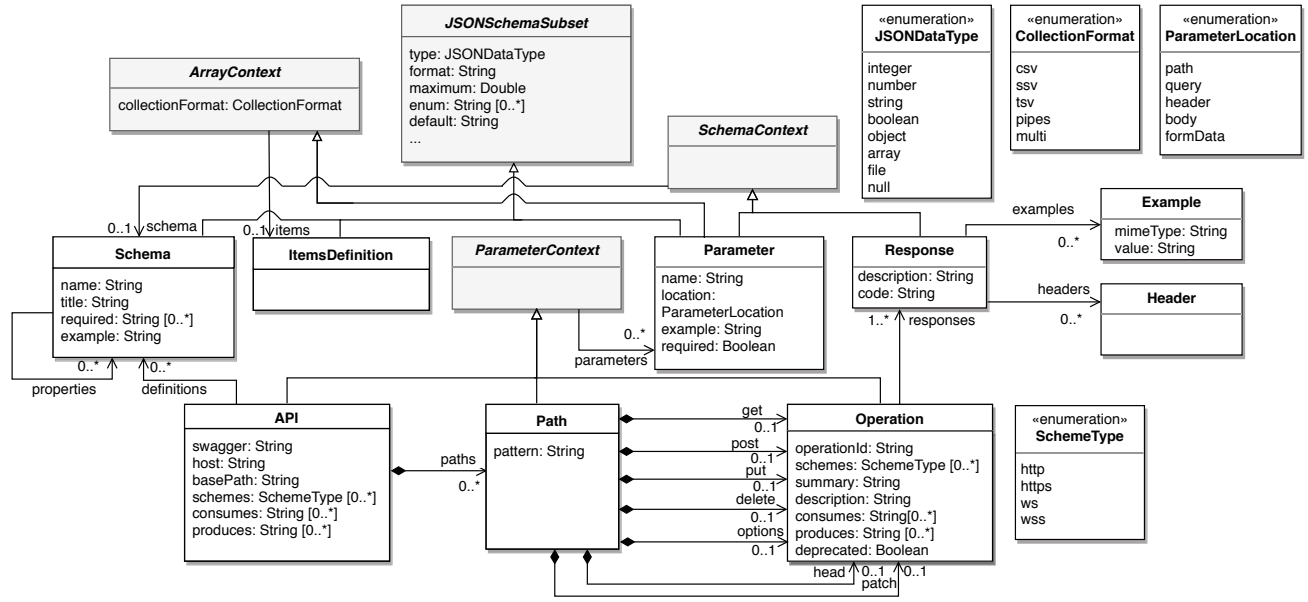

Fig. 2. Excerpt of the OpenAPI metamodel (Ed-douibi et al. [18]).

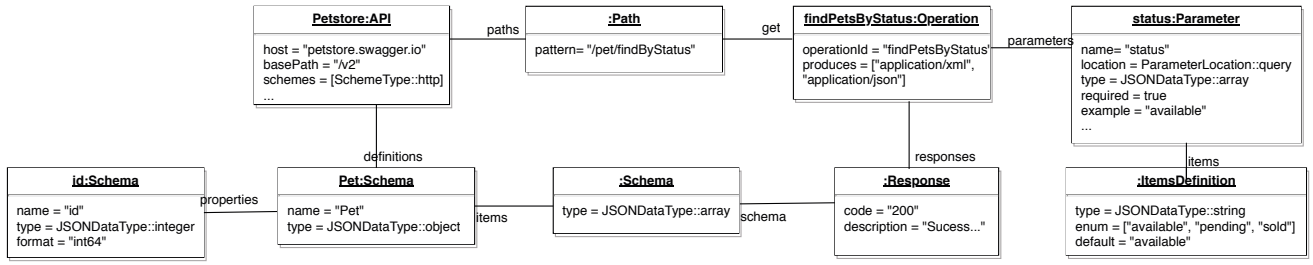

Fig. 3. Excerpt of the OpenAPI model corresponding to the Petstore API.

parameter *status* of the operation *findPetsByStatus* of the Petstore API showed in Figure 3 since `items.default = available`. PR 3 was applied to the parameter *petId* of the operation *getPetById* since a *petId* example can be inferred from the operation *findPetsByStatus*. Note that PR 2 and PR 3 stress the API (i.e., they involve sending several requests to the API) and may lead to biased results as the very own API under test is used to infer the values. Thus, the three rules are applied to infer the required parameters while only the first rule is used for the optional ones.

VII. Extracting Test Case Definitions

In the following we explain the generation process of test case definitions from OpenAPI models. We start by introducing the TestSuite metamodel, used to represent test case definitions; and then we present the transformation rules to create test cases.

A. The TestSuite Metamodel

The TestSuite metamodel allows creating test case definitions for REST APIs. Figure 4 shows an excerpt of the TestSuite metamodel. The `TestSuite` element represents a test suite and is the root element of our metamodel. This element includes a name, the URL of the REST API definition (i.e., `api` attribute), and a set of test case definitions (i.e., `testCases` reference).

The `TestCase` element represents a test case definition and includes a name, a description, and a set of test steps (i.e., `testSteps` references). The `APIRequest` element is a specialization of the `TestStep` element which represents the details of an API request to be sent and the logic to validate the returned response. It includes the target operation (i.e., `operationId` attribute), the content type (e.g., `application/json`), the accepted MIME type (i.e., `accept` attribute, e.g., `application/json`), and the transfer protocol of the request (i.e., `scheme` attribute, e.g., `http`). The values for the content type and accepted MIME type should adhere to RFC 6838[16]. It also includes the parameters of the request (i.e., `parameters` reference), the request authorization method (i.e., `authorizations`

[16]https://tools.ietf.org/html/rfc6838

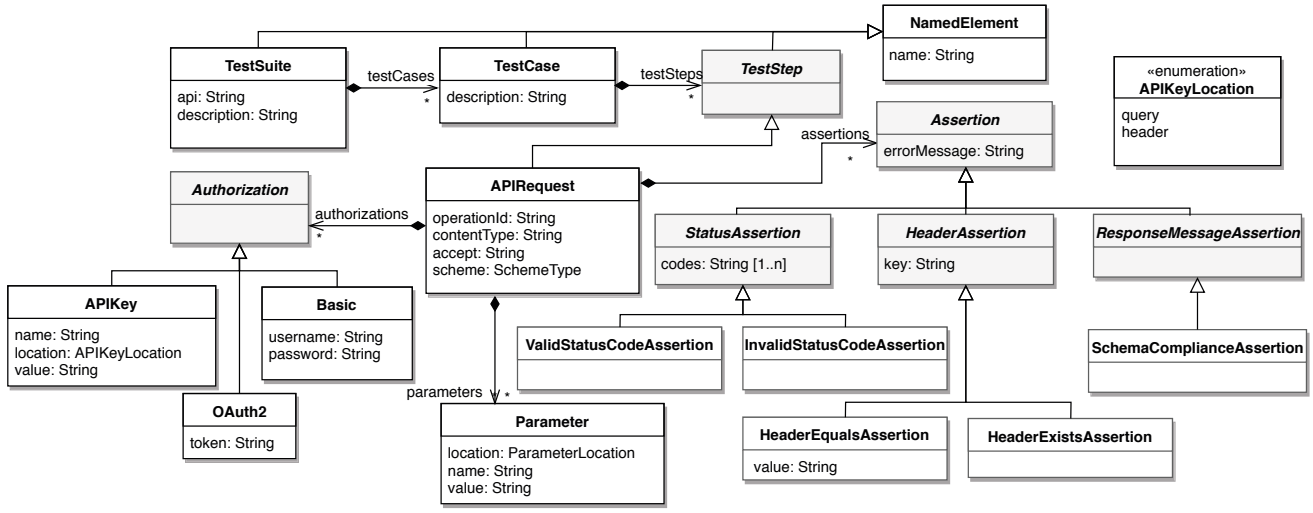

Fig. 4. Excerpt of the TestSuite metamodel

reference), and the assertions to be validated for this API request (i.e., `assertions` reference).

A parameter of an API request is represented by the `Parameter` element and includes its location (i.e., `location` attribute), name, and value. `Parameter` elements are mapped to parameters of the REST API definition by their name and location.

The `Authorization` element represents an abstract authorization method in a REST API. It has three specializations, namely: `APIKey` for API key authorizations, `Oauth2` for OAuth 2.0 authorization using a client token, and `Basic` for basic authentication using a username and a password.

The `Assertion` element represents the root element of the hierarchy of assertions supported by the TestSuite metamodel. As can be seen assertions are organized in three categories which define validations regarding: (1) the HTTP status code of the received HTTP response (i.e. the abstract element `StatusCodeAssertion`); (2) the header of the received HTTP response (i.e., the abstract element `HeaderAssertion`); and (3) the message of the received HTTP response (i.e., the abstract element `ResponseMessageAssertion`).

The `ValidStatusCodeAssertion` and `InvalidStatusCodeAssertion` elements are specializations of the `StatusCodeAssertion` element and allow checking that the HTTP status of the received response is within the defined list of codes, or not, respectively. The `HeaderEqualsAssertion` and `HeaderExistsElement` elements are specializations of the element `HeaderAssertion` and allow checking that the value of an HTTP header in the response is equals to the expected value, and whether an HTTP header exists in the response, respectively.

The `SchemaComplianceAssertion` element is a specialization of the `ResponseMessageAssertion` element which allows checking whether the the response message is compliant with the schema defined by the definition (e.g., check that the returned *Pet* instance is compliant with the `Pet` definition).

B. OpenAPI to TestSuite Transformation

We present now the generation rules we have defined to test that REST APIs respect their OpenAPI definitions. We define two rules (i.e., GR 1 and GR 2) to generate test case definitions in order to assess that the REST APIs behave correctly using both correct and incorrect data inputs. GR 1 generates nominal test case definitions which assess that given correct input data, the API operations return a successful response code (i.e., `2xx` family of codes) and respect their specification. GR 2 generates faulty test case definitions which assess that given incorrect input data, the API operations return a client error response code (i.e., `4xx` family of codes).

GR 1 (Nominal test case definition)**.** If an operation `o` is testable then one `TestCase` testing such operation is generated such as `APIRequest` includes the inferred required parameter values (if any). Additionally, if `o` contains inferable optional parameters then another `TestCase` testing such operation is generated such `APIRequest` includes the inferred required and optional parameter values. In both cases `APIRequest` includes the following assertions:

- `ValidStatusCodeAssertion` having a successful status code (i.e., `2xx` family of codes)
- `SchemaComplianceAssertion`, if `o.responses` contains a response `r` such as `r.code` is a successful status code (i.e., `2xx` family of codes) and `r.schema = s`
- `HeaderExistsAssertion` having `key = h`, if `o.responses` contains a response `r` such as `r.code` is a successful code (i.e., `2xx` family of codes) and `r.headers` contains `h`

GR 2 (Faulty test case definition)**.** For each parameter `p` in an operation `o`, a `TestCase` testing such operation is generated for the following cases:

TABLE I
WRONG DATA TYPES GENERATION RULES.

PARAMETER TYPE	GENERATION RULE
`object`	an object violating the object schema
`integer/int32`	a random string or a number higher than $2^{31}-1$
`integer/int64`	a random string or a number higher than $2^{63}-1$

`number/float`
`number/double`
`string/byte`
`string/datetime`
`string/date`
	a random string

`boolean`	a random string different from `true` and `false`

TABLE II
VIOLATED CONSTRAINTS GENERATION RULES.

CONSTRAINT	GENERATION RULE
`enum`	a string or a number outside of the scope of the enumeration
`pattern`	a string violating the regEx
`maximum/exclusiveMaximum`	a number higher than maximum
`minimum/exclusiveMinimum`	a number lower than minimum
`minLength`	a string with the length lower than `minLength`
`maxLength`	a string with the length higher than `maxLength`
`maxItems`	an array having more items than `maxItems`
`minItems`	an array having less items than `minItems`
`uniqueItems`	an array with duplicated values
`multipleOf`	a number not divided by `multipleOf`

- **Required missing:** If `p` is required and not located in the path then `APIRequest` will not include a value of `p`.
- **Wrong data types:** If `o` is testable and `p` is not of type string then `APIRequest` will include the inferred required parameter values (if any) and a wrong value of `p` as described in Table I.
- **Violated constraints:** If `o` is testable and `p` includes a constraint then `APIRequest` will include the inferred required parameter values (if any) and a value of `p` violating such constraint as described in Table II.

In all cases `APIRequest` includes the following assertion:

- `ValidStatusCodeAssertion` having a client-error code (i.e., `4xx` family of codes).

Figure 5 shows an example of nominal and faulty test cases for the operation *findPetsByStatus* of Petstore represented as a model conforming to our TestSuite metamodel. Note that our approach does not consider dependencies between operations since such information is not provided in OpenAPI v2.0. With no knowledge about dependencies it is not possible to *undo* the changes and therefore each operation is treated in an isolated mode (i.e., test case side-effects are ignored, e.g, creating resource, deleting resources). Heuristics could be defined to detect such dependencies, however, they would assume that API providers follow the best practices in the design of their REST APIs and respect the semantics of HTTP protocol, which in practice is generally not the case [11]. This situation does not affect *read-only* APIs (e.g., Open Data APIs) but could affect *updateable* APIs. Thus, we created two generation modes: *safe* (i.e., only GET operations are considered to generate test cases), and *unsafe* (i.e., all operations are considered to generate test cases).

VIII. CODE GENERATION

The final step of the process consists on generating test cases for a target platform. Since the test case definitions are platform-independent, any programing language or testing tool could be considered. In Section IX we will illustrate our approach for Java programming language and JUnit templates.

IX. TOOL SUPPORT

We created a proof-of-concept plugin implementing our approach. The plugin extends the Eclipse platform to generate: (1) OpenAPI models from OpenAPI definition files; (2) TestSuite models from the generated OpenAPI models; and (3) Maven projects including the JUnit templates implementations for the test case definitions in the TestSuite models. The plugin has been made available in our GitHub repository [19].

The OpenAPI metamodel and the TestSuite metamodel are implemented using the Eclipse Modeling Framework (EMF). We created a set of Java classes to infer the parameter values as described in Section VI. We defined an ATL transformation to derive a TestSuite model from an input OpenAPI model following the generation rules described in Section VII. Finally, we used Acceleo[17] to generate the JUnit test cases.

Figure 6 shows a screenshot of the generated Maven project for the Petstore API including the corresponding tests for test cases showed in Figure 5. The generated classes rely on JUnit[18] to validate the tests and Unirest framework[19] to call the REST API. To test the schema compliance we infer the JSON schema from the API description and use the framework JSON Schema Validator[20] to validate the *entity-body* of the response against the inferred schema. The actual implementation of the tool supports the authentication methods Basic and API Key. The complete implementation is available in our Github repository [19].

X. VALIDATION

In order to validate our approach and the companion tool implementation, we address the following research questions:

RQ1 What is the coverage level of the generated test cases? Test cases cover a set of endpoints, operations, parameters and data definitions of the OpenAPI definition. Our goal is to provide high coverage degrees.

RQ2 What are the main failing points in the definitions and implementation of real world REST APIs? We use our approach to study how current REST APIs perform and when they usually fail.

[17]https://www.eclipse.org/acceleo/
[18]https://junit.org/
[19]http://unirest.io/
[20]https://github.com/java-json-tools/json-schema-validator

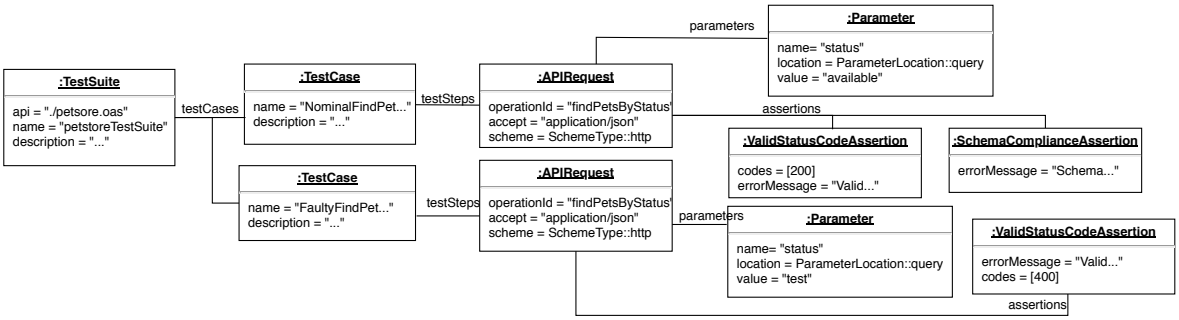

Fig. 5. TestSuite model representing nominal and faulty test cases for the Petstore API.

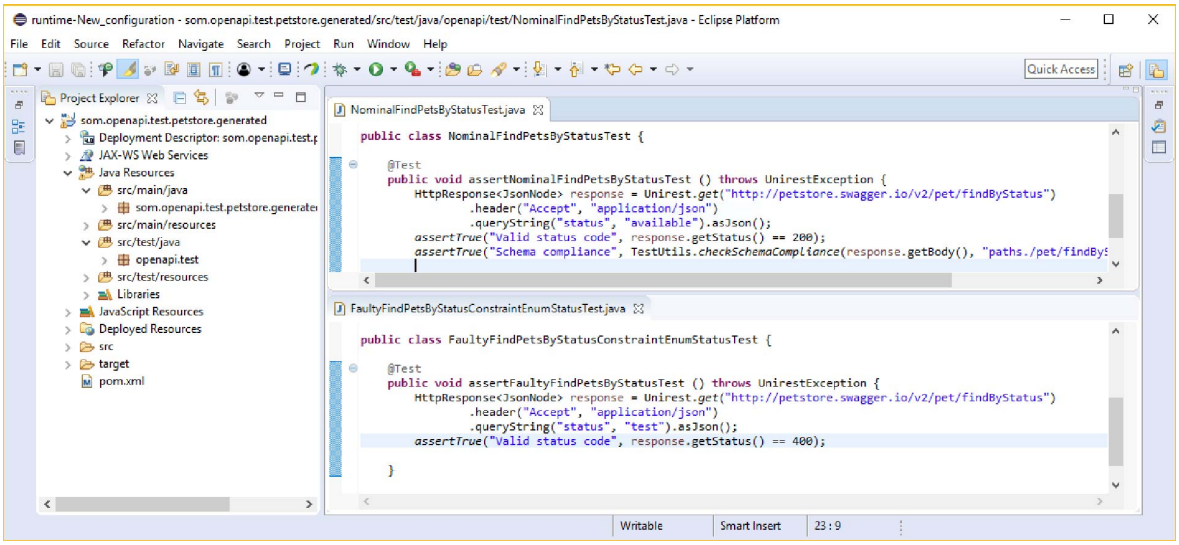

Fig. 6. A screenshot of the generated Maven project of the Petstore API showing a nominal and a faulty test case.

To answer these questions we ran our tool on a collection of OpenAPI definitions. We describe next how we created this collection and addressed the research questions.

A. REST APIs Collection and Selection

We created a collection of REST APIs by querying *APIs.guru*, a site including 837 REST APIs described by OpenAPI definitions. To build our collection, we performed a two-phased filtering process to (1) select free, open and available REST APIs which are not services (i.e., providing access to data models) and (2) remove those REST APIs with incorrect or invalid OpenAPI definitions. Next we detail these phases.

The first phase performs the selection process relying on the metadata of each API provided in the OpenAPI definition or in its external documentation. The selection criteria in this phase removes: (1) APIs providing access to functionalities and services (mostly applying fees) to manage specific environments such as IoT, Cloud, and messaging which are normally tested in controlled environments (654 APIs); (2) deprecated and/or unreachable APIs (15 APIs); and (3) APIs with restricted access (e.g., available for partners only, rate limited, etc.) (21 APIs). At the end of this phase our collection included 147 APIs.

The second phase analyzes each OpenAPI definition of the previously selected REST APIs and removes (1) APIs which showed semantic errors (i.e., validation failed against the OpenAPI specification) (21 APIs); (2) APIs relying on OAuth as authentication mechanism (15 APIs); (3) big APIs including more than 100 parameters to avoid biased results (14 APIs); and (4) not REST-friendly APIs (e.g., using formData, appending body to GET methods) (6 APIs). At the end of this phase, our collection included 91 OpenAPI definitions[21] belonging to 32 API providers. Most of the APIs in the collection provide support for *text processing* (25 APIs) and *Open Data* (21 APIs). Other APIs target other domains such as *transport* (6 APIs), *media* (5 APIs), and *entertainment* (4 APIs). The REST APIs described by these definitions have on average 4 operations and 10 parameters. The collection included 71 REST APIs that require authentication, thus API keys were

[21]The full list of collected APIs, including the application of the selection process, and the results for answering the research questions are available at http://hdl.handle.net/20.500.12004/1/C/EDOC/2018/001.

TABLE III
COVERAGE OF THE TEST CASES IN TERMS OF OPERATIONS, PARAMETERS, ENDPOINTS AND DEFINITIONS.

ELEMENTS	COUNT	COVERAGE			COVERAGE (%)		
		ALL	NOMINAL	FAULTY	ALL	NOMINAL	FAULTY
OPERATIONS	367	320	303	233	87%	82%	63%
PARAMETERS	949	595	485	476	62%	51%	50%
ENDPOINTS	356	289			81%		
DEFINITIONS	313	239			76%		

TABLE IV
ERRORS FOUND IN THE TEST CASES.

	TOTAL	NOMINAL TEST CASES		FAULTY TEST CASES	
		4XX/500	SCHEMA	500	2XX
NUMBER OF APIS	37	9	11	11	20
%	40%	25%	30%	30%	55%

requested to use them. Furthermore, the OpenAPI definitions of 51 of the selected APIs were polished to follow the security recommendations (i.e., using the *SecurityDefinition* element provided by the specification instead of parameters).

To minimize the chances of subjectivity when applying this process, we applied a code scheme as follows. The process was performed by one coder, who is the first author of this paper. Then a second coder, who is the second author of this paper, randomly selected a sample of size 30% (251 Web APIs) and performed the process himself. The intercoder agreement reached was higher than 92%. All disagreement cases were discussed between the coders to reach consensus.

B. Results

Once we built our collection, we ran our tool for each REST API to generate and execute the test cases.

1) RQ1. Coverage of the generated test cases: For the 91 APIs, we generated 958 test cases (445 nominal / 513 faulty). We analyzed these test cases to determine their coverage in terms of operations, parameters, endpoints and definitions. We defined the coverage as follows: an operation is covered when at least one test case uses such operation; a parameter is covered when at least one test case includes an operation targeting such parameter; an endpoint is covered when all its operations are covered; and a definition is covered when it is used in at least one test case. We report the coverage for both the total test cases and the nominal/faulty ones.

Table III summarizes our results. As can be seen, we obtain high coverage levels except for parameters. The value for nominal test cases for parameters is low since this kind of test cases rely on required parameters (i.e., testable operations) and only 30% of the parameters were required. On the other hand, the nature of the parameters and poor OpenAPI definitions affected the coverage of faulty test cases. Most of the parameters were of type *String* and do not declare constraints to validate their values, thus hampering the generation of faulty test cases. Richer OpenAPI definitions would have helped us to tune our inference technique and therefore increase the coverage for operations and parameters. Further analysis shown that many APIs provide constraints regarding their parameters in natural language instead of the method provided by the OpenAPI specification. For instance, the *data at work API*[22] includes a parameter named *limit* and described as "Maximum number of items per page. Default is 20 and cannot exceed 500". Instead, using the tags *default* and *maximum* defined by the OpenAPI specification would have helped us generate more test cases for this parameter.

2) RQ2. Failing points in the definitions and implementation in real world REST APIs: To answer this question we analyzed the errors found when executing the generated test cases. Table IV shows the results. As can be seen, 37 of the 91 selected REST APIs (40% of the APIs) raised some kind of error, which we classified into two categories (nominal/faulty). The nominal test case errors included: those regarding the status code (i.e., obtaining an error status code when expecting a successful response, see column 3) and those regarding schema compliance (i.e., the object returned in the body of the response is not compliant with the JSON schema defined in the specification, see column 4). The faulty test case errors included two kind of errors regarding the status code where a client error response (i.e., 4xx code) was expected but either a server error response code (i.e., 500 code, see column 5) or a successful response code (i.e., 2xx code, see column 6) was obtained. The errors regarding nominal test cases are mainly related to mistakes in the definitions, while the faulty ones are related to bad implementations of the APIs.

The errors in the definitions vary from simple mistakes such a missing *required* field for a parameter (e.g., the parameter *q* in the operation *GET action/organization_autocomplete* of the *BC Data Catalogue API*[23]) to complex ones such as having a wrong JSON schema (e.g., *Data at Work API*).

The errors in the implementation of the APIs are characterized by sending `500` (i.e., internal server error) or `2xx` (i.e., successful) status codes on an error condition instead of a `4xx` status code. On the one hand, the `500` internal server error tells the client that there is a problem on the server side. This is probably a result of an unhandled exception while dealing with the request. Instead, the server should include a validation

[22]https://api.apis.guru/v2/specs/dataatwork.org/1.0/swagger.json
[23]https://api.apis.guru/v2/specs/gov.bc.ca/bcdc/3.0.1/swagger.json

step which checks the client's input and sends a client side error code (e.g., `400` wrong input) on violated constraints with a text explaining the reasons. On the other hand, sending `200` status code on an error condition is a bad practice [10]. Furthermore, four errors were linked to the limitation of OpenAPI to define mutually exclusive required parameters (e.g., The *Books API*[24]), resulting in inconsistent definitions. Such limitation is confirmed by Oostvogels et al. [20].

Even though we do not consider links between operations and side effects, we believe our approach may help developers to identify the main error-prone points in REST API development. Thus, our experiments showed that extra attention should be paid when dealing with data structures and returning the proper HTTP error code to allow the client to recover and try again.

C. Threats to Validity

Our work is subjected to a number of threats to validity, namely: (1) internal validity, which is related to the inferences we made; and (2) external validity, which discusses the generalization of our findings. Regarding the internal validity, many definitions in *APIs.guru* have been created or generated by third-parties which may have consequences on the quality of such definitions. Therefore, different results can be expected if all definitions are provided by the API owners. Furthermore, some providers are over-represented in *APIs.guru* which may lead to biased results. As for the external validity, note that the nature and size of the sample may not be representative enough to generalize the findings of the experiments.

XI. Conclusion

In this paper we presented a model-driven approach to automate specification-based REST API testing by relying on OpenAPI. We used metamodels to represent OpenAPI and test case definitions (i.e., test suites) and models conforming to these metamodels are used to generate the test cases. The TestSuite metamodel is reusable, thus allowing defining test suites for any REST API. As a proof-of-concept, we created a plugin implementing our approach for the Eclipse platform [19]. Our experiments show that the generated test cases cover on average 76.5% of the elements included in the definitions and that 40% of the tested APIs contain bugs either in their specification or server implementation.

As future work, we are interested in increasing the coverage levels by improving our parameter inference technique, for instance, by using natural language processing techniques to infer parameter values and constraints from the parameter descriptions. The usage of search-based techniques to generate test cases could also be useful here. We plan to extend our approach in order to support the newly released version of OpenAPI (i.e., OpenAPI v3.0) as it is getting more attraction and adoption. OpenAPI v3.0 includes some interesting new features (e.g., explicit links between operations) which will allow us to deal with dependencies between operations and side-effects caused by executing test cases. Thus, we would like to extend our TestSuite metamodel in order support: (1) links between test steps which will require adding a property transfer mechanism and (2) *pre-* and *post-* requests. This will therefore improve our approach by including: (1) integrity test cases to evaluate the life-cycle of a particular resource in a REST API; and (2) *pre-* and *post-* operations to prepare data for a test case and manage side-effects, respectively.

[24]https://api.apis.guru/v2/specs/nytimes.com/books_api/3.0.0/swagger.json

Acknowledgment

This work has been supported by the Spanish government (TIN2016-75944-R project).

References

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.
[2] M. J. Hadley, "Web Application Description Language (WADL)," Tech. Rep., 2006.
[3] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. "Big"' Web Services," in *Int. Conf. on World Wide Web*, 2008, pp. 805–814.
[4] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and Verification in Service-Oriented Architecture: A Survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.
[5] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-based Automatic Test Case Generation for Web Services Testing," in *Int. Workshop on Service-Oriented System Engineering*, 2005, pp. 207–212.
[6] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "WS-TAXI: A WSDL-based Testing Tool for Web Services," in *Int. Conf. on Software Testing Verification and Validation*, 2009, pp. 326–335.
[7] S. Hanna and M. Munro, "Fault-Based Web Services Testing," in *Int. Conf. on Information Technology: New Generations*, 2008, pp. 471–476.
[8] J. Offutt and W. Xu, "Generating Test Cases for Web Services Using Data Perturbation," *Software Engineering Notes*, vol. 29, no. 5, pp. 1–10, 2004.
[9] A. Arcuri, "RESTful API Automated Test Case Generation," in *Int. Conf. on Software Quality, Reliability and Security*, 2017, pp. 9–20.
[10] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, Inc., 2013.
[11] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: a large-scale analysis of compliance with principles and best practices," in *Int. Conf. on Web Engineering*. Springer, 2016, pp. 21–39.
[12] L. J. Morell, "A Theory of Fault-based Testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, 1990.
[13] W. Xu, J. Offutt, and J. Luo, "Testing Web Services by XML Perturbation," in *Int. Symp. on Software Reliability Engineering*, 2005, pp. 257–266.
[14] J. Zhang and L.-J. Zhang, "Criteria Analysis and Validation of the Reliability of Web Services-Oriented Systems," in *Int. Conf. on Web Services*, 2005.
[15] T. Fertig and P. Braun, "Model-driven Testing of RESTful APIs," in *Int. Conf. on World Wide Web*, 2015, pp. 1497–1502.
[16] S. K. Chakrabarti and P. Kumar, "Test-the-REST: An Approach to Testing RESTful Web-Services," in *Int. Conf. on Advanced Service Computing*, 2009, pp. 302–308.
[17] C. Benac Earle, L.-Å. Fredlund, Á. Herranz, and J. Mariño, "Jsongen: A QuickCheck Based Library for Testing JSON Web Services," in *Workshop on Erlang*, 2014, pp. 33–41.
[18] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot, "Example-driven Web API Specification Discovery," in *Eur. Conf. on Modelling Foundations and Applications*, 2017.
[19] "OpenAPI Tests Generator. http://hdl.handle.net/20.500.12004/1/A/TESTGEN/001."
[20] N. Oostvogels, J. De Koster, and W. De Meuter, "Inter-parameter Constraints in Contemporary Web APIs," in *Int. Conf. on Web Engineering*, 2017, pp. 323–335.