Test Coverage Criteria for RESTful Web APIs

Alberto Martin-Lopez
amarlop@us.es
University of Seville
Seville, Spain

Sergio Segura
sergiosegura@us.es
University of Seville
Seville, Spain

Antonio Ruiz-Cortés
aruiz@us.es
University of Seville
Seville, Spain

ABSTRACT

Web APIs following the REST architectural style (so-called RESTful web APIs) have become the de-facto standard for software integration. As RESTful APIs gain momentum, so does the testing of them. However, there is a lack of mechanisms to assess the adequacy of testing approaches in this context, which makes it difficult to automatically measure and compare their effectiveness. In this paper, we first present a set of ten coverage criteria that allow to determine the degree to which a test suite exercises the different inputs (i.e. requests) and outputs (i.e. responses) of a RESTful API. We then arrange the proposed criteria into eight Test Coverage Levels (TCLs), where TCL0 represents the weakest coverage level and TCL7 represents the strongest one. This enables the automated assessment and comparison of testing techniques according to the overall coverage and TCL achieved by their generated test suites. Our evaluation results on two open-source APIs with real bugs show that the proposed coverage levels nicely correlate with code coverage and fault detection measurements.

KEYWORDS:

REST,

testing,

web services,

coverage criteria

1 INTRODUCTION

Web Application Programming Interfaces (APIs) are key in the development of distributed architectures, as they enable the seamless integration of heterogeneous systems. This has, in turn, fostered the emergence of new consumption models such as mobile, social and cloud applications. The increasing use of web APIs is reflected in the size of popular API directories such as ProgrammableWeb [11] and RapidAPI [12], which currently index over 21K and 8K web APIs, respectively. Contemporary web APIs usually follow the REpresentational State Transfer (REST) architectural style [7], being referred to as RESTful web APIs. *RESTful web APIs* provide uniform interfaces to interact with resources (e.g. a song in the Spotify API) via create, read, update and delete (CRUD) operations, generally through HTTP interactions. RESTful APIs are commonly described using languages such as the OpenAPI Specification (OAS), which provides a structured way to describe a RESTful API in a both human and machine-readable way, making it possible to automatically generate, for example, documentation, source code (clients and servers) and tests. In what follows, we will use the terms RESTful web API, web API, or simply API interchangeably.

RESTful APIs can be tested using black-box [2, 6, 14] and white-box [1] approaches. The former are usually based on the API specification and try to cover all elements and features defined in it, while the latter typically focus on source code or mutation coverage measures. While white-box approaches can be easily compared in terms of source code coverage, no standardized coverage criteria exist for black-box. This lack of criteria impedes the comparison of testing techniques and hinders the development of new ones, since there is no automated nor easy way to evaluate their effectiveness.

In this paper, we present a catalogue of ten test coverage criteria for RESTful web APIs. Each coverage criterion measures how many elements of an API are covered by a test suite, both in terms of test inputs (i.e. API requests) and outputs (i.e. API responses). To this end, we took inspiration on the OAS language, which allows to describe the functionality of an API in a straightforward manner. We propose to arrange the coverage criteria into eight Test Coverage Levels (TCLs), where TCL0 represents the weakest coverage level and TCL7 represents the strongest one. These levels constitute a common framework for the assessment and comparison of testing techniques for RESTful APIs, called the Test Coverage Model (TCM). This framework aims at fully automating the evaluation of testing approaches in this context, based on the overall coverage and TCL achieved by their generated test suites. For the evaluation of our approach, we performed experiments on two open-source APIs with real bugs. The results show that the proposed coverage levels nicely correlate with code coverage and fault detection measurements, i.e. the higher the TCL that a test suite complies with, the more chances to find faults and the more code will be covered.

The remaining of the paper is organized as follows: Section 2 introduces the basic concepts regarding RESTful web APIs. Section 3 presents our proposal of test coverage criteria in the context of REST. In Section 4, a set of coverage levels is laid out. Section 5 exposes the results on the case study performed to validate our proposal. The related work is discussed in Section 6. Finally, Section 7 draws the conclusions and presents future lines of research.

2 RESTFUL WEB APIS

RESTful web APIs are usually decomposed into multiple RESTful web services [9, 13], each of which implements one or more create, read, update or delete (CRUD) operations over a specific resource. These operations are usually mapped to the HTTP methods POST, GET, PUT and DELETE, respectively. Figure 1 depicts an excerpt of an OpenAPI specification from a sample API called MyMusic, containing seven operations. As illustrated, an OpenAPI Specification document describes an API in terms of paths, operations, resources, request parameters and responses. A *resource* is any type of information that can be exposed to the Web (e.g. a photo, a HTML document, information about a book) and it is addressable by a unique Uniform Resource Identifier (URI). The API described in Figure 1 provides operations for handling two resources: *Songs* and *Playlists*. A *path* (also called *route* or *endpoint*) represents a resource over which operations can be performed (e.g. */songs*). The term *operation* refers to the use of a HTTP method over a specific path. Below there are some examples of operations on the MyMusic API.

`GET /songs?q=rhapsody&type=cover` Retrieve all covers that contain the keyword `'rhapsody'` in its name.

`POST /playlists` Create a new playlist.

`PUT /playlists/{playlistId}` Update an existing playlist, identified by the `playlistId` field.

`DELETE /playlists/{playlistId}` Remove a playlist.

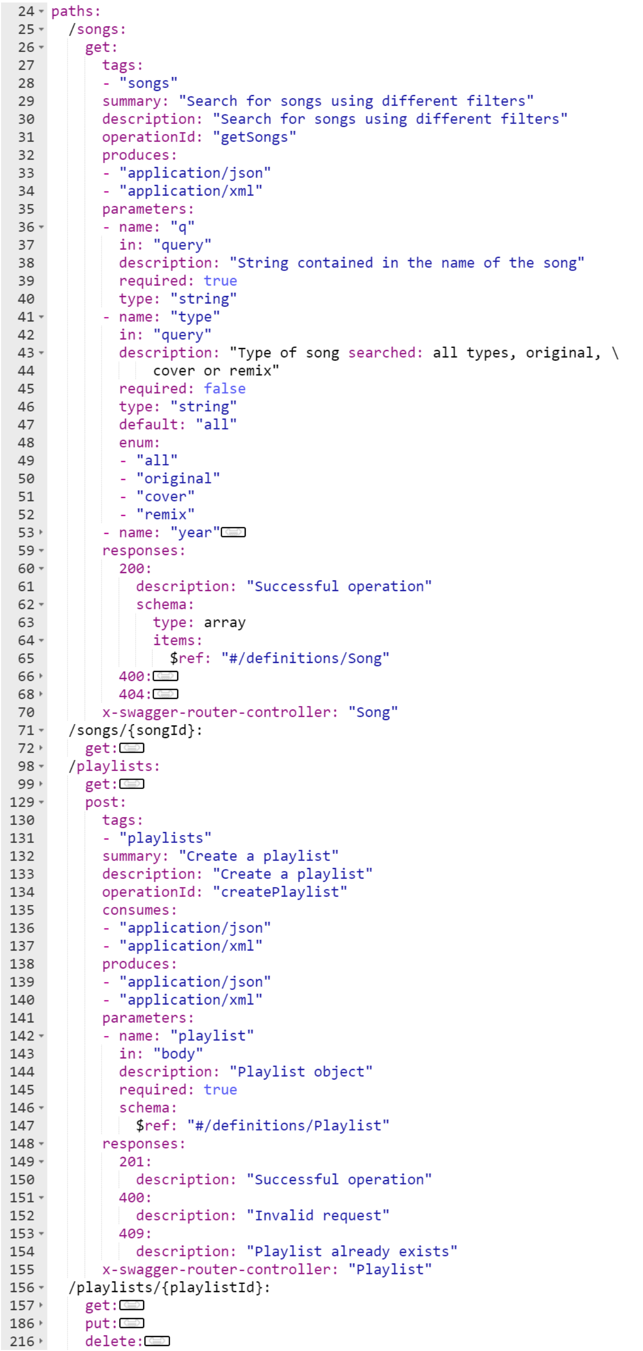
Operations accept parameters. A *parameter* is a piece of information that can be passed together with the request for several purposes, such as filtering and sorting results (e.g. `q` and `type` in the GET operation shown above). For every operation, several expected responses can be specified. A *response* is identified by the returned status code and can optionally include a body. The *status code* determines the result of the operation (e.g. successful or not) and the *response body* includes additional information (e.g. a requested resource). For instance, *GET /songs* in the previous example could return a 400 status code if the required parameter `q` was not included in the request, or a 200 together with a set of results in the response body if there were no errors in the API call.

3 TEST COVERAGE CRITERIA

In this section, we present a catalogue of coverage criteria for RESTful web APIs. These criteria are divided into two types: input criteria (those related to the API requests) and output criteria (those related to the API responses). It is worth noting that the proposed criteria can be applied at different levels. Hence, for example, we could measure the coverage achieved by a test suite in a whole API, a certain path, or a specific operation. In what follows, we present each coverage criterion. For the sake of understandability, we will make reference to the test suite shown in Table 1. It is composed of seven test cases (TCs) for the MyMusic API (Figure 1). Each TC is composed of some inputs (i.e. one or more API calls) and one expected output (i.e. the last API response obtained).

3.1 Input Coverage Criteria

This type of criteria measure the degree to which test cases cover elements related to API requests.



```
paths:
  /songs:
    get:
      tags:
      - "songs"
      summary: "Search for songs using different filters"
      description: "Search for songs using different filters"
      operationId: "getSongs"
      produces:
      - "application/json"
      - "application/xml"
      parameters:
      - name: "q"
        in: "query"
        description: "String contained in the name of the song"
        required: true
        type: "string"
      - name: "type"
        in: "query"
        description: "Type of song searched: all types, original, \
            cover or remix"
        required: false
        type: "string"
        default: "all"
        enum:
        - "all"
        - "original"
        - "cover"
        - "remix"
      - name: "year"
      responses:
        200:
          description: "Successful operation"
          schema:
            type: array
            items:
              $ref: "#/definitions/Song"
        400:
        404:
      x-swagger-router-controller: "Song"
  /songs/{songId}:
    get:
  /playlists:
    get:
    post:
      tags:
      - "playlists"
      summary: "Create a playlist"
      description: "Create a playlist"
      operationId: "createPlaylist"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/json"
      - "application/xml"
      parameters:
      - name: "playlist"
        in: "body"
        description: "Playlist object"
        required: true
        schema:
          $ref: "#/definitions/Playlist"
      responses:
        201:
          description: "Successful operation"
        400:
          description: "Invalid request"
        409:
          description: "Playlist already exists"
      x-swagger-router-controller: "Playlist"
  /playlists/{playlistId}:
    get:
    put:
    delete:
```

Figure 1: Excerpt of an OpenAPI specification in YAML.

Path coverage. This criterion measures the coverage of a test suite according to the paths it exercises. The coverage is computed as the number of paths executed divided by the total number of paths of the API. To achieve 100% path coverage, at least one request must address each path of the API. For instance, the MyMusic API exposes four paths (*/songs*, */songs/{songId}*, */playlists* and */playlists/{playlistId}*), so four HTTP requests are needed, one per path, to reach 100% path coverage. There are multiple ways to meet

Table 1: Test suite for the API MyMusic.

TC	Request	Expected response

#1	`GET /songs?q=rhapsody&type=original&year=1975`
	`Accept: application/json`
		Status code: 200
		Response body: array of Song objects
		Content-type: JSON

#2	`GET /songs?q=happier&type=all`
	`Accept: application/xml`
		Status code: 200
		Response body: array of Song objects
		Content-type: XML

#3	`GET /songs?q=pwbglauypw&type=cover&year=2020`
	`Accept: application/json`
		Status code: 404
		Response body: Error object
		Content-type: JSON

#4	`GET /songs?type=remix`
	`Accept: application/json`
		Status code: 400
		Response body: Error object
		Content-type: JSON

#5	`GET /songs/1`
	`Accept: application/json`
		Status code: 200
		Response body: Song object
		Content-type: JSON

#6	`GET /songs/99999999`
	`Accept: application/xml`
		Status code: 404
		Response body: Error object
		Content-type: XML

#7	1. `POST /playlists` (body containing JSON object)
	`Content-type: application/json`
	2. `GET /playlists/1`
	`Accept: application/json`
		Status code: 200
		Response body: Playlist object
		Content-type: JSON

this criterion in the MyMusic API, for instance, with TCs #1, #5 and #7. Likewise, TCs #4, #6 and #7 cover all the paths as well.

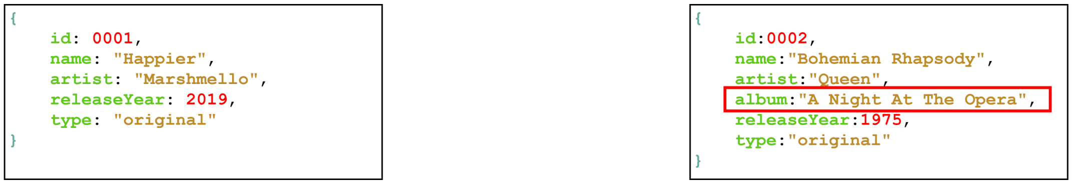
Operation coverage. This criterion measures the coverage of a test suite according to the operations executed. The coverage is computed as the number of operations executed divided by the total number of operations of the API. To achieve 100% operation coverage, every path must be sent one request per allowed HTTP verb (GET, POST, PUT or DELETE). Notice that this criterion can be applied to the entire API or to a specific path. For example, TCs #1, #5 and #7 reach 57% operation coverage in the MyMusic API, since they execute 4 out of the 7 operations of the API. At the same time, TC #1 on its own achieves 100% operation coverage on the */songs* path, since only one operation can be performed on it.

Parameter coverage. This criterion measures the coverage of a test suite according to the operation parameters it uses. The coverage is computed as the number of parameters used divided by the total number of parameters in the API. To achieve 100% parameter coverage, all input parameters of every operation must be used at least once. Exercising different combinations of parameters is desirable, but not strictly necessary to achieve 100% of coverage under this criterion. The reason for excluding combinatorial coverage criteria (e.g. t-wise [4]) is to ease the development of coverage analysis tools. This criterion can also be considered for specific subsets of the API. As an example, TC #1 achieves 100% parameter coverage for the */songs* path, since all parameters are used once. Overall, however, the test suite reaches 60% parameter coverage on the entire API, given that 4 out of 10 parameters are not used, because these parameters belong to the three operations not executed by the test suite (the query parameter in the operation *GET /playlists*, the body and path parameter in *PUT /playlists/{playlistId}* and the path parameter in *DELETE /playlists/{playlistId}*).

Parameter value coverage. This criterion measures the coverage of a test suite according to the parameter values exercised. This criterion applies only to parameters with a finite number of possible values, namely, *booleans* and *enums*. The coverage is computed as the number of different values that parameters are given divided by the total number of possible values that all parameters can take. To achieve 100% parameter value coverage, every *boolean* and *enum* parameter must take all possible values. Nevertheless, it is suggested to test multiple values with other types of parameters such as *strings* and *integers*. There is only one *enum* parameter in the MyMusic API: the parameter `type` in the *GET /songs* operation. TCs #1, #2, #3 and #4 cover the four values that it accepts (`'original'`, `'all'`, `'cover'` and `'remix'`), and therefore they achieve 100% coverage under this criterion in the whole API.

Content-type coverage. This criterion measures the coverage of a test suite according to the input content-types used in API requests. This criterion applies only to operations that accept data in the request body (i.e. POST, PUT and DELETE). The coverage is computed as the number of input content-types used divided by the total number of input content-types across all API operations. To achieve 100% input content-type coverage, for every operation that accepts a request body, all data formats (e.g. JSON and XHTML) must be tested. This criterion can be applied to each operation individually. As an example, the MyMusic API has only two operations that accept a body, namely *POST /playlists* and *PUT /playlists/{playlistId}*; each of these can process JSON and XML, therefore at least four requests are needed to achieve 100% input content-type coverage on the entire API. TC #7 reaches 50% coverage of this criterion for the *POST /playlists* operation, since the JSON content-type is covered but XML is not.

Operation flow coverage. This criterion measures the coverage of a test suite according to the sequences of operations it executes.



```
{
    id: 0001,
    name: "Happier",
    artist: "Marshmello",
    releaseYear: 2019,
    type: "original"
}
```

(a) GET /songs/0001

```
{
    id:0002,
    name:"Bohemian Rhapsody",
    artist:"Queen",
    album:"A Night At The Opera",
    releaseYear:1975,
    type:"original"
}
```

(b) GET /songs/0002

Figure 2: API responses including optional and required properties.

The definition of full coverage of this criterion highly depends on the API under test. Several proposals exist in the literature about the operation flows that should be tested [1, 2, 8, 16], but none of them is widely accepted and used in industry. For this reason, and for the sake of simplicity, we propose to use a simplified version of the flows defined by Arcuri in [1]: for every resource that can be created, at most four operation flows must be tested, namely those related to its reading (one or several), updating and deletion after its creation. If the resource is a sub-resource of another one, the creation of the parent resource must be included in the operation flow. In the MyMusic API, four operation flows can be executed: i) create playlist ⟶ read one playlist; ii) create playlist ⟶ read several playlists; iii) create playlist ⟶ update playlist; iv) create playlist ⟶ delete playlist. TC #7 executes the first flow (individual read), so this criterion is 25% covered.

3.2 Output Coverage Criteria

This type of criteria measure the degree to which test cases cover elements related to API responses.

Status code class coverage. This criterion measures the coverage of a test suite according to its ability to produce both correct and erroneous responses in the API under test. These responses are typically identified by 2XX and 4XX status codes respectively, however, this may vary depending on the API, therefore the tester must define the meaning of *correct* and *erroneous* for their particular case. It is assumed that every operation should at least return one successful response, therefore, to achieve 100% coverage of this criterion, at least one test case per API operation is needed; if every operation can return both correct and erroneous status codes, two test cases per operation are needed to reach 100% status code class coverage. The coverage is computed as the number of classes of status codes obtained in API responses (maximum two per operation) divided by the total number of classes of status codes in the whole API. In the MyMusic API, the seven operations can return correct and erroneous responses, therefore fourteen test cases are needed to fully cover this criterion. Overall, the test suite achieves 36% status code class coverage (5 out of 14 classes covered). At the same time, TCs #1 and #3 suffice to fulfill this criterion for the *GET /songs* operation.

Status code coverage. This criterion extends the previous one by considering status codes instead of simply classes of status codes. Therefore, to achieve 100% coverage, all status codes of all operations must be obtained. TCs #1, #3 and #4 achieve 100% coverage for the *GET /songs* operation of the MyMusic API, and also for the */songs* path, since it has no more operations.

Response body properties coverage. This criterion measures the coverage of a test suite according to its ability to produce responses containing all properties of resources. Figure 2 shows two API responses containing a *Song* resource, which is composed of multiple properties such as the name of the song. The coverage of this criterion is computed as the number of properties obtained divided by the total number of all properties from all objects that can be obtained in API responses. To achieve 100% coverage of this criterion, all properties from all response objects must be obtained. As an example, take the two responses to the request *GET /songs/{songId}* shown in Figure 2. Retrieving a *single*[1] (left-hand side) does not achieve full coverage, since the property *album* is not present in the response. Retrieving a song that is part of an album (right-hand side), by contrast, meets this criterion for this specific response body, since it includes all properties of the *Song* object. Intuitively, this criterion can be applied to specific responses individually, like in the MyMusic API, where TC #1 covers this criterion for the successful response to the *GET /songs* operation, since the response will surely include the same object depicted in the right-hand side of the previous figure.

Content-type coverage. This criterion has the same meaning as the input content-type criterion, but in this case the coverage is measured on the output data formats obtained in API responses. TCs #5 and #6 achieve 100% coverage of this criterion for the *GET /songs/{songId}* operation, since all content-types are obtained in the responses, i.e. JSON and XML.

4 TEST COVERAGE MODEL

Inspired by the REST Maturity Model of Richardson [13], we propose to arrange the previous coverage criteria into eight different Test Coverage Levels (TCLs), constituting a common framework for the assessment and comparison of test suites addressing RESTful APIs, called the *Test Coverage Model* (TCM). The goal is to rank test suites based on the TCL they can reach, where TCL0 represents the weakest coverage level and TCL7 the strongest one. In order to reach a specific TCL, all criteria belonging to previous levels must have been met. This does not mean that criteria from higher levels subsume those from lower levels. Figure 3 illustrates the aforementioned levels and the test coverage criteria they are comprised of (as described in the previous section). Note that a distinction is made between input and output coverage criteria.

Level 0 represents a test suite where no coverage criterion is fully met. Any test suite, therefore, has TCL0 by default. **Level 1** is the easiest to achieve and the most naive, as it only requires paths

[1] A *single* refers to a song that is released on its own, not as part of an album.

to be covered. A test suite reaching only level 1 can be thought of as very weak. In **Level 2**, all operations must be covered. **Level 3** requires that all content-types for every operation are tested, both input and output. The next three levels mainly focus on parameters and output criteria. **Level 4** requires parameters and status code classes to be covered. In **Level 5**, the criteria that must be fulfilled are parameter values and status codes. In **Level 6**, it is required that the response body properties criterion be covered. It is suggested to cover different combinations of parameters in levels 4, 5 and 6, as well as new parameter values in levels 5 and 6. **Level 7** constitutes the last stage of the TCM and the hardest to reach. It focuses on the coverage of operation flows.

As an example, consider the test suite from Table 1 and the TCL it achieves depending on the elements of the API considered:

Entire API: TCL1. All the four paths are exercised by the test cases but three out of the seven operations are not executed (*GET /playlists*, *PUT /playlists/{playlistId}* and *DELETE /playlists/{playlistId}*).

Path ***/songs***: TCL6. All operations are covered by the test cases and, for every operation, all their sub-elements are covered as well (input and output content-types, parameters, parameter values, status codes and response body properties).

Operation ***GET /songs***: TCL6. All elements of the operation are covered: parameters (`q`, `type` and `year`), parameter values (`'all'`, `'original'`, `'cover'` and `'remix'` for the `type` parameter), input and output content-types (JSON and XML), status codes (200, 400 and 404) and response body properties (`id`, `name`, `artist`, `album`, `type` and `releaseYear`).

Operation ***GET /playlists***: TCL1. This operation is not executed by the test cases, so the test suite reaches only level 1, since the path is actually covered by TC #7.

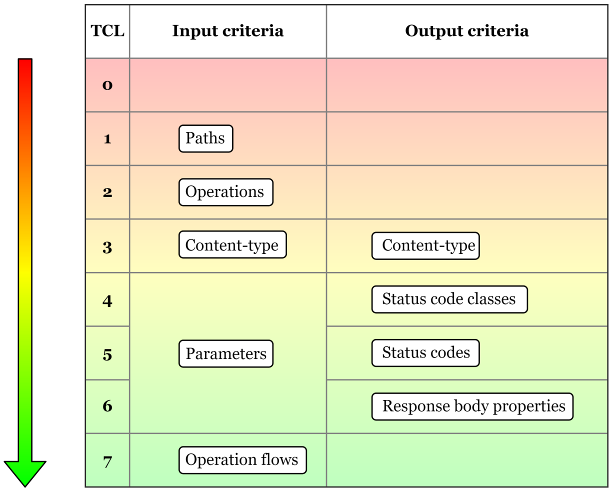
Operation ***GET /songs/{songId}***: TCL4. Both content-types (JSON and XML) are covered, all parameters are used at least once (`songId`) and both classes of status codes are covered (one test case gets a 200 status code in the response and the other obtains a 404).

5 CASE STUDY

We performed a case study with two open-source APIs to validate our proposal of test coverage criteria for RESTful APIs. To this end, we designed seven test suites for each API, each complying with every TCL defined in the TCM, i.e. fully covering the criteria included in every TCL. In doing so, we aim to validate the following hypothesis: *The TCL of a test suite is correlated to the number of bugs it is able to detect and the code coverage it achieves.*

5.1 Subject APIs

We decided to use the same APIs of Arcuri [1] for the empirical study, so that we could compare our work with a recent, novel technique of RESTful APIs testing. We selected two of the five APIs analysed in his work, named *scout-api*[2] and *features-service*[3]. The *scout-api* API allows to search for suitable activities for boys and girls scouts, and the *features-service* API allows to manage product feature models. We selected these APIs because they have medium sizes and they differ sufficiently from each other, e.g. the first uses a wide range of path, query and JSON object parameters whereas the second mostly uses a small range of path parameters.



TCL	Input criteria	Output criteria
0		
1	Paths	
2	Operations	
3	Content-type	Content-type
4	Parameters	Status code classes
5		Status codes
6		Response body properties
7	Operation flows	

Figure 3: Test Coverage Model: Levels & Criteria.

The information about the APIs is summarized in Table 2. The last column shows the number of real errors identified by Arcuri [1]. An error is considered as an operation returning at least one 5XX status code as a result of a test case.

Table 2: Subject APIs.

Name	Classes	LoC	Operations	Errors
scout-api	75	7479	49	33
features-service	23	1247	18	14

5.2 Setup

Before going in depth into the results obtained, it is necessary to bear in mind a number of considerations. For the evaluation of TCL4, and as recommended in most guidelines for the design of RESTful APIs [9, 13], we defined *correct* responses as those returning 2XX status codes, and *erroneous* as those returning 4XX or 5XX status codes. On the other hand, we designed the test suites such that they contained the minimum number of test cases possible to meet the criteria of the corresponding TCL. To this end, we did not test special combinations of parameters in TCLs 4, 5 and 6, and we did not add extra test cases to test more than one value per parameter in TCLs 5 and 6, even though the guidelines of the TCM encourage to do the opposite. By doing this, we intentionally assessed the validity of the TCM in a worst case scenario.

Regarding the experiments performed, the test cases were manually written in Java, using the JUnit framework and the REST Assured library. The tests were run in the IDE IntelliJ IDEA, which allowed us to automatically measure the failures detected and the code coverage achieved by the test suites.

5.3 Results

Table 3 shows the results of our evaluation and a comparison with Arcuri's and the test suite provided with each API, for which we

[2] https://github.com/mikaelsvensson/scout-api
[3] https://github.com/JavierMF/features-service

Table 3: Evaluation results and comparison between test suites of the TCM, Arcuri and the API developers.

API	Test Coverage Model (TCM)				Arcuri			Existing test suite		
	TCL	Test cases	Code coverage	Errors	Test cases	Code coverage	Errors	Test cases	Code coverage	Errors
scout-api	1	21	18%	3	177	38%	**33**	19	**41%**	0
	2	49	20%	21						
	3	49	20%	21						
	4	98	34%	23						
	5	224	35%	29						
	6	232	40%	33						
	7	232	40%	**33**						
features-service	1	11	35%	7	50	64%	**14**	22	**78%**	2
	2	18	39%	13						
	3	18	39%	13						
	4	36	75%	13						
	5	37	76%	13						
	6	37	76%	13						
	7	37	74%	13						

only considered test cases making HTTP calls to the API, as Arcuri did [1]. Values in bold represent the highest values achieved among our approach, Arcuri's and the API's existing test suites.

It can be clearly appreciated how the TCL of a test suite *does* have an effect on the bugs found and the code covered. Whereas fault finding gradually increases in one of the APIs, code coverage increases in both of them for most levels. There is one exception: the decrement in the code covered by TCL7 in the *features-service* API, although it is only 2% less. This is because the operation flows defined in Section 3.1 do not suffice to put the system in the same state as when populating the database as desired, which proves that the operation flows that should be tested are API-dependant. We tested more complex sequences of operations and were able to cover the same code as with the previous level, but these were not considered for the case study, in order to conform to the minimum requirements to fulfill the operation flows criterion.

In terms of code coverage, TCL7 achieves better results than those of Arcuri and similar results to the existing test suites in both APIs. TCL4 is clearly a turning point, since it achieves the greatest improvement in comparison with the previous level. This highlights the importance of testing all operation parameters and obtaining both classes of status codes, correct and erroneous.

In terms of fault finding, TCL7 reveals many more bugs than the existing test suites of the APIs (33 vs 0 in *scout-api*, and 13 vs 2 in *features-service*), and almost the same as Arcuri (33 vs 33, and 13 vs 14). There is only one bug that our approach was not able to find, which may be explained by the fact that we did not test many parameter values, aiming to assess the validity of the TCM with test suites complying with the minimum requirements for each level. Especially significant is the case of the TCL2-compliant test suite of the *features-service* API, which detects the same number of faults as one of TCL7. This highlights the importance of testing all API operations, despite not using all operation parameters.

Overall, TCL7 achieves more than twice code coverage than TCL1 in both APIs, and uncovers between 2 and 11 times more bugs. This supports the idea that, by systematically covering all criteria defined in our catalogue, it is possible to obtain sound coverage and fault detection results.

5.4 Discussion

The results of this case study reflect the potential of the TCM using the minimum number of test cases needed to comply with every TCL. We also conducted an exploratory experiment to evaluate the TCM following a stronger approach regarding the use of parameter values, namely, testing between 1 and 3 values for each parameter in TCL6, and obtained higher code coverage for both APIs (41% in *scout-api*, and 79% in *features-service*), which suggests that parameter values are key in the efficacy of the TCM.

In order to obtain significant results when following the TCM approach, it is required that the system database is properly populated, containing all kinds of resources. We noticed that the results obtained (especially in terms of code coverage) were poorer when it was empty, since some operations would not work as expected (e.g. retrieving a resource that does not exist). To this end, it may be necessary to manually populate the database accordingly to the needs of each test case (as we did). However, there may also be cases where this is not possible, for example when performing black-box testing on an industrial web API. In this scenario, operation flows play a key role in the design of valid test cases, since they allow to put the system in the desired state prior to a specific API call.

It is worth mentioning that the tests from the API's existing test suite are more complex than those of our work and Arcuri's. For example, we only check the status codes and test a specific CRUD operation per test case, while the API's existing test cases may check multiple aspects such as response body properties and perform multiple CRUD operations on a single test case. This is why the existing test suites contain notably less test cases than ours and Arcuri's.

As a final remark, the TCM approach does not guarantee good fault finding statistics on its own (similarly to what happens with other traditional coverage criteria such as statement coverage), since this depends on the assertions (test oracles) used in the test cases. High TCLs do guarantee high code coverage, but these need to be accompanied by meaningful assertions and oracles in order to increase the chances of finding bugs. For our experiments, we only checked the response status codes and yet were able to find a significant amount of bugs. Had we used smarter approaches like metamorphic testing [14], we might have been able to discover more and more complex faults.

6 RELATED WORK

Our work is related to black-box testing approaches for web services. Some of these works make an effort to measure, up to a certain degree, the elements of the web services under test covered by the test cases generated by their techniques. Ed-douibi et al. [6] automatically generated 951 test cases for 91 RESTful APIs based on their OAS specifications, and measured the coverage achieved in terms of inputs, namely: paths, operations, parameters and OAS object definitions. Other authors have proposed test coverage criteria for web services based on their specifications such as the Web Services Description Language (WSDL) [5]. In this regard, Bartolini et al. [3] enunciated three coverage criteria, namely operation coverage (as defined in the WSDL file), message coverage (input messages declared in the WSDL specification) and schema coverage (the parts composing each message), used for measuring the thoroughness with which their tool could test a service. Bai et al. [16] also used WSDL to analyse the test coverage achieved by test cases according to four types of elements: parameters, messages (input and output), operations and operation flows. Lastly, Jokhio et al. [10] used the Web Service Modelling Ontology (WSMO) framework to generate test cases and measure the coverage with two specific criteria: boundary coverage, referred to boundary conditions such as minimum and maximum values for parameters, and transition rules path coverage, referred to the different execution paths that the program may follow when receiving a given request. In comparison with these papers, our work constitutes the first and most complete framework to measure black-box coverage in RESTful web services. Furthermore, the validity of this framework has been demonstrated with two real RESTful web APIs and several testing techniques.

Regarding the tools available in the market, ReadyAPI [15] is the only one that provides meaningful information about the coverage achieved by a test suite in terms of the functionality covered. Given a Web Application Description Language (WADL) or OAS document and a test suite created in ReadyAPI, the program is able to run the test suite and compute coverage statistics on the elements covered in the API. However, it supports only 6 out of the 10 criteria proposed in this paper: on the one hand, the coverage is computed in terms of the sub-elements of each operation, and therefore does not provide information regarding path or operation coverage; on the other hand, it lacks support for the criteria of parameter values and operation flows. Lastly, the tool does not offer any general overview of the thoroughness of the test suite, as opposed to the TCM approach presented in this paper.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a catalogue of test coverage criteria for RESTful APIs and proposed a framework for the evaluation of testing approaches in this context. To the best of our knowledge, this is the first attempt to establish a common framework for the comparison of testing techniques, by providing an easy and automatic way of measuring the thoroughness of test suites addressing RESTful web APIs, based on the criteria they cover and the coverage level they reach. Furthermore, we evaluated the validity of our proposal with two real-world APIs and found that the TCLs nicely represent the potential of test suites, where the lowest TCLs usually get low code coverage and find few faults and the highest TCLs cover as much code and find as many faults as traditional and modern testing techniques. We trust that the results of our work pave the way for the automated assessment and comparison of testing approaches for RESTful APIs.

Several challenges remain for future work. It is desirable to perform further evaluations of the proposed criteria and the TCM with other APIs and test generation techniques, so as to ascertain the validity of the results obtained. We aim to provide tool support that allows to analyse test suites and find out their maximum TCL. We also plan to elaborate on the definition of the operation flows criterion, since it is not fully formalised and there is no agreement in the literature about the operation flows that should be tested [1, 2, 8, 16]. Lastly, our approach opens new promising research opportunities in terms of test automation. For example, search-based techniques could be used to generate test suites that maximise API coverage.

ACKNOWLEDGEMENTS

This work has been partially supported by the European Commission (FEDER) and Spanish Government under projects BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21), and the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077).

REFERENCES

[1] A. Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. on Software Engineering and Methodology* 28, 1 (2019), 3.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk. 2018. *REST-ler: Automatic Intelligent REST API Fuzzing.* Technical Report.

[3] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. 2009. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Intern. Conference on Software Testing Verification and Validation.* 326–335.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. on Software Engineering* 23, 7 (1997).

[5] WWW Consortium. 2007. *Web Services Description Language (WSDL) Version 2.0.* Retrieved May 2019 from https://www.w3.org/TR/wsdl20/

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

[7] R. T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation.

[8] A. Ivanchikj, C. Pautasso, and S. Schreier. 2018. Visual modeling of RESTful conversations with RESTalk. *Journal of Software & Systems Modeling* 17, 3 (2018), 1031–1051.

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

[10] M. S. Jokhio, G. Dobbie, and J. Sun. 2009. Towards Specification Based Testing for Semantic Web Services. In *Australian Software Engineering Conference.* 54–63.

[11] ProgrammableWeb. 2019. *RapidAPI API Directory.* Retrieved March 2019 from http://www.programmableweb.com/

[12] RapidAPI. 2019. *RapidAPI API Directory.* Retrieved March 2019 from https://rapidapi.com

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

[14] S. Segura, J.A. Parejo, J. Troya, and A. Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Trans. on Software Engineering* 44, 11 (2018), 1083–1099.

[15] SmartBear. 2019. *ReadyAPI.* Retrieved March 2019 from https://smartbear.com/product/ready-api/overview/

[16] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. 2005. WSDL-based Automatic Test Case Generation for Web Services Testing. In *IEEE Intern. Workshop on Service-Oriented System Engineering.* 207–212.