This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

Metamorphic Testing of RESTful Web APIs

Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés

Abstract—Web Application Programming Interfaces (APIs) allow systems to interact with each other over the network. Modern Web APIs often adhere to the REST architectural style, being referred to as RESTful Web APIs. RESTful Web APIs are decomposed into multiple resources (e.g., a video in the YouTube API) that clients can manipulate through HTTP interactions. Testing Web APIs is critical but challenging due to the difficulty to assess the correctness of API responses, i.e., the oracle problem. Metamorphic testing alleviates the oracle problem by exploiting relations (so-called metamorphic relations) among multiple executions of the program under test. In this paper, we present a metamorphic testing approach for the detection of faults in RESTful Web APIs. We first propose six abstract relations that capture the shape of many of the metamorphic relations found in RESTful Web APIs, we call these Metamorphic Relation Output Patterns (MROPs). Each MROP can then be instantiated into one or more concrete metamorphic relations. The approach was evaluated using both automatically seeded and real faults in six subject Web APIs. Among other results, we identified 60 metamorphic relations (instances of the proposed MROPs) in the Web APIs of Spotify and YouTube. Each metamorphic relation was implemented using both random and manual test data, running over 4.7K automated tests. As a result, 11 issues were detected (3 in Spotify and 8 in YouTube), 10 of them confirmed by the API developers or reproduced by other users, supporting the effectiveness of the approach.

Index Terms—Metamorphic testing, REST, RESTful Web services, Web API

1 Introduction

Web *Application Programming Interfaces (APIs)* specify how to access services and data over the network, typically using Web services [1], [2]. Web APIs are rapidly proliferating as a key element to foster reusability, integration, and innovation, enabling new consumption models such as mobile or smart TV apps. Companies such as Facebook, Twitter, Google, eBay or Netflix receive billions of API calls every day from thousands of different third-party applications and devices, which constitutes more than half of their total traffic [1]. Many companies are also exposing their existing assets as private APIs, enabling their own developers to build innovative mobile, social or cloud applications [1], [3]. Web APIs are usually compliant with the REpresentational State Transfer (REST) architectural style, being referred to as *RESTful* Web APIs [4]. RESTful Web APIs comprise of a set of so-called RESTful Web services, where each service implements one or more create, read, update, or delete (CRUD) operations over a resource, e.g., a shopping order in the eBay API. At the time of writing this paper, the ProgrammableWeb Web site [5], a popular API repository, indexes over 5k RESTful Web APIs from multiple domains such as shopping, finances, social networks, or telephony.

As Web APIs are progressively becoming the cornerstone of software integration, their validation is getting more critical. In this context, the fast detection of bugs is of utmost importance to increase the quality of internal products and third-party applications. However, testing Web APIs is challenging mainly due to the difficulty to assess whether the output of an API call is correct, i.e., the *oracle problem* [6]–[8]. For instance, consider the Web API of the popular music streaming service Spotify [9]. Suppose a search for albums with the query "redhouse" returning 21 total matches: Is this output correct? Do all the albums in the result set contain the keyword? Are there any albums containing the keyword not included in the result set? Answering these questions is difficult, even with small result sets, and often infeasible when the results are counted by thousands or millions.

Metamorphic testing alleviates the oracle problem by providing an alternative when the expected output of a test execution is complex or unknown [10], [11]. Rather than checking the output of an individual program execution, metamorphic testing checks whether multiple executions of the program under test fulfil certain necessary properties called *metamorphic relations*. For instance, consider the following metamorphic relation in Spotify: *two searches for albums with the same query should return the same number of total results regardless of the size of pagination*. Suppose that a new Spotify search is performed using the exact same query as before and increasing the maximum number of results per page from 20 (default value) to 50: This search returns 27 total albums (6 more matches than in the previous search), which reveals a bug. This is an example of a real and reproducible fault detected using the approach presented in this paper and reported to Spotify. According to Spotify developers[1], it was a regression fault caused by a fix with undesired side effects.

In this paper, we present a metamorphic testing approach for the automated detection of faults in RESTful Web APIs (henceforth also referred to as simply Web APIs). We observed that Web APIs have very clear semantics, specified as CRUD operations over resources, and a very consistent use of parameters for standard operations such as filtering, ordering and pagination [2], [12]–[14]. This suggests that

• *S. Segura, José A. Parejo, Javier Troya, and A. Ruiz-Cortés are with the Dept. of Computer Languages and Systems, Universidad de Sevilla, Spain. E-mail: sergiosegura@us.es*

1. https://github.com/spotify/web-api/issues/225

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

many of the metamorphic relations derived for a Web API, like the previously identified in Spotify, could be applicable to other Web APIs. As a result, we introduce the concept of metamorphic relation output patterns. A *Metamorphic Relation Output Pattern (MROP)* defines an abstract output relation typically identified in Web APIs, regardless of their application domain. Each MROP is defined in terms of set operations among test outputs such as equality, union, subset, or intersection. For each MROP, there may exist numerous input relations satisfying the output relation defined by the pattern. We show some of those input relations as examples, remaining the identification of new input relations open on each Web API under test. MROPs provide a helpful guide for the identification of metamorphic relations, broadening the scope of our work beyond a particular Web API. Based on the notion of MROP, a methodology is proposed for the application of the approach to any Web API following the REST architectural pattern. This methodology is intended to be used as a complement to the existing tools and state-of-the-art testing methods used by API developers and API testers, either in-house or in third-party API testing companies, e.g., API Fortress [15].

The approach was evaluated in several steps. First, we used the proposed methodology to identify 33 metamorphic relations in four Web APIs developed by undergraduate students. All the relations are instances of the proposed MROPs. Then, we assessed the effectiveness of the identified relations at revealing 317 automatically seeded faults (i.e., mutants) in the APIs under test. As a result, 302 seeded faults were detected, achieving a mutation score of 95.3%. Second, we evaluated the approach using real Web APIs and faults. In particular, we identified 20 metamorphic relations in the Web API of Spotify [9] and 40 metamorphic relations in the Web API of YouTube [16]. Each metamorphic relation was implemented and automatically executed using both random and manual test data. In total, 469K metamorphic tests were generated, out of which we selected a subset of 4,720 tests for the evaluation[2]. As a result, 21 metamorphic relations were violated, and 11 issues revealed and reported (3 issues in Spotify and 8 issues in YouTube). To date, 10 of the reported issues have been either confirmed by the API developers or reproduced by other users supporting the effectiveness of our approach.

The remainder of this paper is structured as follows. In Section 2 RESTful Web APIs and metamorphic testing are introduced. The MROPs proposed are presented in Section 3. Section 4 describes the proposed methodology. The experimental evaluation of our approach is reported in Section 5. Section 6 discusses the potential threats to the validity of our contribution. The related work is reviewed in Section 7. Finally, we summarize our conclusions in Section 8.

2 Background

In this section, we present the basics to understand our approach. We start with a brief introduction to RESTful Web APIs, and then we describe metamorphic testing.

2. The selection of the tests was made to avoid potential biases caused by approximated results or performance optimizations not detailed in the user documentation of the Web APIs under test (c.f. Section 5).

2.1 RESTful Web APIs

The *REpresentational State Transfer (REST)* is an architectural style for distributed hypermedia systems like the Web [4]. Web APIs that adhere to the REST architectural constraints are called *RESTful* Web APIs. RESTful Web APIs are decomposed into multiple RESTful Web services, where each service implements one or more CRUD operations over a resource. A *resource* is anything that can be exposed to the Web such as a video, a photo or a shopping order [17].

Resources are typically identified by a Uniform Resource Identifier (URI), which makes them addressable and manipulable using an application protocol, typically HTTP. An *API endpoint* is a unique URI that identifies one or more resources. Most RESTful Web APIs follow a well-known set of design guidelines [2], [12]–[14] that includes implementing the standard HTTP methods as follows:

- *GET*. It is used to retrieve one or more resources.
- *POST*. It is used to create a resource. If successful, it returns the newly created resource.
- *PUT*. It is used to update a resource. If successful, it returns the updated resource.
- *DELETE*. It is used to delete a resource.

As an example, the following request gets information about "The Rolling Stones" in Spotify: "`GET https://api.spotify.com/v1/artists/22bE4uQ6baNwSHPVcDxLCe`". The URI (i.e., API endpoint) identifies the artist by means of its *Artist Identifier*, and the HTTP method (GET) specifies the operation to perform over the resource (read). URIs can include parameters to perform certain operations over resources such as filtering, ordering, and pagination. Parameters can be included either as a part of the URI path (such as the Spotify Artist Identifier appended at the end of the previous URI) or using standard URI parameters of the form `param=value`. For instance, the following request adds a "like" rating to the YouTube video with id=9kWEk_RLAs: "`POST https://www.googleapis.com/youtube/v3/videos/rate?id=S9kWEk_RLAs&rating=like`"[3].

Resources can be represented using different formats such as JSON, XML or XHTML. We focus on JSON in this paper. The *JavaScript Object Notation (JSON)* is a lightweight and human-readable data-interchange format composed of property-value pairs. Fig. 1 shows an extract of the artist resource in JSON format obtained as a response to the Spotify request previously presented. Note that data values may include objects (delimited with curly brackets), arrays (delimited with square brackets) and references to other URIs, enabling navigating from one resource to another.

Web APIs typically provide online documentation describing how to use the API. This usually includes information about the available resources, URIs, HTTP methods, parameters, data exchange format, HTTP status codes, authentication data and possible errors. Additionally, some Web APIs include request and response examples or even a "Try it!" Web interface for clients to call the API from a Web browser and check the response.

Throughout the rest of the paper, we illustrate and evaluate our approach using, among others, the Web APIs of

3. API Key parameter omitted.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering


```
{
  ▸ external_urls: { … },
  ▸ followers: { … },
  ▾ genres: [
      "british blues",
      "classic rock"
    ],
    id: "22bE4uQ6baNwSHPVcDxLCe",
  ▸ images: [ … ],
    name: "The Rolling Stones",
    popularity: 77,
    type: "artist",
    uri: "spotify:artist:22bE4uQ6baNwSHPVcDxLCe"
}
```


Figure 1: JSON representation of an artist in Spotify

Spotify and YouTube. Spotify is a music streaming application with over 50M paying subscribers as of March 2017, and over 140M total active users as of June 2017[4]. The Spotify Web API [9] provides programmatic access to resources such as artists, albums, tracks, or playlists metadata in JSON format. YouTube is a video-sharing application owned by Google with over 1G users and around 300K videos uploaded daily. The YouTube Data API [16] is mainly used to programmatically interact with several resources such as videos, channels, or playlists in JSON format. Both Web APIs can be used freely but with restrictions such as a maximum number of API calls per user and day and a maximum number of calls in a period of time.

2.2 Metamorphic testing

Metamorphic testing provides an alternative to test a program when the expected output of a test case is unknown or hard to compare with the actual output [10]. Rather than checking the output of an individual test, metamorphic testing checks whether multiple test executions fulfil certain metamorphic relations. A *metamorphic relation* is a necessary property of the intended program's functionality that relates two or more input data and their expected outputs [18]. For instance, consider the mathematical function $\min(a, b)$ that calculates the minimum value of two integers a and b. The order of the inputs should not influence the output, which can be expressed as the following metamorphic relation: $\min(a, b) = \min(b, a)$. In this metamorphic relation, (a, b) is called the source input, and (b, a) is called the follow-up input. Let P_{min} be a program implementing the minimum function. P_{min} can be tested against the previous metamorphic relation by running some *metamorphic tests* where specific input values are used. For instance, we can first run $P_{min}(2, 3)$ and then run $P_{min}(3, 2)$ and then check whether the two outputs are equal. Here, $(2, 3)$ and $(3, 2)$ are called the source test case and follow-up test case, respectively. If the outputs of a source test case and its follow–up test case(s) violate the metamorphic relation, the metamorphic test is said to have failed, indicating that the program under test contains a bug.

Formally, a metamorphic relation for a function f is expressed as a relation among a series of function inputs $x_0, x_1, \ldots, x_n$ (with $n \geq 1$), and their corresponding outputs $f(x_0), f(x_1), \ldots, f(x_n)$, that is, a relation $\mathbb{R}(x_0, x_1, \ldots, x_n, f(x_0), f(x_1), \ldots f(x_n))$ [19]. We refer to x_0 as the source input, and to x_i ($i \in [1, n]$) as the follow-up input. For instance, in the min example $x_0 = (a, b)$, $x_1 = (b, a)$, $f(x_0) = \min(a, b)$ and $f(x_1) = \min(b, a)$. Therefore, the relation between x_0 and x_1 could be $x_0 = rev(x_1)$ (where rev is a function that reverses the order of items in an input list), and the relation between $f(x_0)$ and $f(x_1)$ would be equality, i.e.:

$$R = \{(x_0, x_1, \min x_0, \min x_1) \mid x_1 = rev(x_0) \rightarrow \min x_0 = \min x_1\}$$

As a further example, consider testing an online search engine [20]. Let $Count(q)$ be the number of results returned for a search query q. Intuitively, the number of returned results for q should be equal or smaller if sites containing a word w are excluded. This can be expressed as the following metamorphic relation: $Count(q) \geq Count(q\text{ -}w)$, where the '-' operator is used to exclude sites including w. Consider a source test case consisting in a search for the keyword "metamorphic", resulting in 3.3K results. Suppose that a follow-up test case is constructed by searching for the query "metamorphic -testing" returning 4.2K results: This is greater than the result for "metamorphic", and thus violates the relation, revealing a bug in the system.

Metamorphic relations implicitly define how, given an existing source test input (q), one has to transform this into a follow-up test input (q -w), such that both test cases can be executed and the relation checked on their inputs and outputs. As a result, metamorphic testing is also considered an effective test case generation technique. In the search engine example, for instance, metamorphic testing could be used together with a random word generator to automatically construct source test cases (e.g., "bright") and their respective follow–up test cases (e.g., "bright -tool") until a pair that reveals a bug is found, if any such pair exists. This idea has been used in our work to automatically generate over 469K metamorphic tests for the Web APIs of Spotify and YouTube (c.f. Section 5).

Metamorphic testing was introduced as an approach to reuse existing test cases back in 1998 by Chen et al. [10]. Since then, as reported in a recent survey [11], the literature on metamorphic testing has flourished and applications of the technique have been reported in numerous domains including Web services and applications [18], computer graphics [21], compilers [22], bioinformatics [23], and cybersecurity [24]. Additionally, there exist evidences of real bugs being detected by metamorphic testing in real-world tools such as the search engine Google [18], the compiler GCC [22], the machine learning system RapidMiner [25] and the NASA Data Access Toolkit (DAT) [26].

3 Metamorphic Relation Output Patterns

The identification of metamorphic relations is a manual task that requires creativity and a good knowledge of the program under test. A common approach to identify metamorphic relations is to check the program specification or user documentation, and consider how the program inputs can be modified so that they can produce a predictable change in the output [19]. However, this is a manual process

4. https://press.spotify.com/es/about/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

that must be repeated from scratch based on the intended functionality of each program under test, unless the program addresses a general problem domain where some metamorphic relations have already been identified, e.g., sorting [10].

To ease the burden of identifying metamorphic relations, in this section we present six Metamorphic Relation Output Patterns (MROPs) that capture the form of common metamorphic relations found in RESTful Web APIs. Each MROP describes an abstract output relation typically found in Web APIs, regardless of their application domain. For each MROP, there may exist a variety of input relations satisfying the output relation defined by the pattern. The input relation may depend on the specific API under test, which typically includes adding, removing, or changing input parameters and input resources in a certain way. When combined with appropriate input relations, each MROP can be instantiated into one or more concrete metamorphic relations on each API under test. The proposed patterns have been defined based on the key principles of the REST architectural design [4] and the guidelines for the design of RESTful Web APIs reported in the literature [2], [12]–[14]. More specifically, we assume that the Web API under test implements CRUD operations (typically using HTTP methods) using the default semantics described in Section 2.1, e.g., update operations should return the updated resource as an output. Also, we assume that the API under test supports standard operations such as filtering, sorting, or pagination. The idea of defining abstract relations from which concrete metamorphic relation instances are derived has been exploited in previous approaches on metamorphic testing of search engines [20], [27], machine learning applications [28], [29] and numerical programs [30].

Building on the definition of metamorphic relation given in the previous section, let $S = f(x_0)$ be the source output, and $F_i = f(x_i)$ the i-th follow-up output. By *output* we refer to a set of items, ordered or not, returned by a call to the Web API under test, e.g., a set of video resources. A MROP is expressed as a relation among the source and follow-up outputs: $\mathbb{P} = (S, F_1, F_2, \ldots, F_n)$. Hence, test inputs remain undefined until the pattern is instantiated. More specifically, a MROP is instantiated by defining the source and follow-up inputs and how they are related, in a way that it is expected to produce outputs satisfying the relation defined by the pattern $\mathbb{P}$. Next, each pattern is introduced and illustrated using examples from Spotify and YouTube.

3.1 Equivalence

This pattern represents those relations where the source and follow-up outputs are equivalent. We define two or more outputs as *equivalent* if they include the same items although not necessarily in the same order, i.e., $\forall i \in [1, n] \cdot S \equiv F_i$. For instance, ordering the results of a query by different criteria should produce equivalent outputs. Thus, a search for YouTube videos with the keywords "*metamorphic testing*" should return the same videos regardless of the ordering criteria specified (date, rating, relevance, title, or number of views).

3.2 Equality

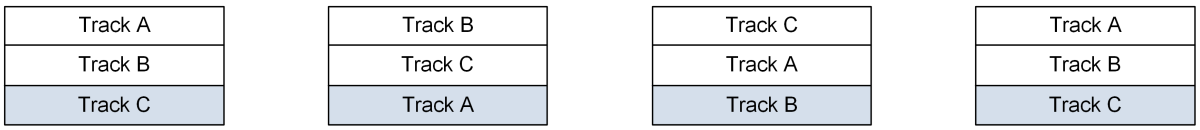
This pattern represents those relations where the source and follow-up outputs must contain the same items and in the same order, i.e., $\forall i \in [1, n] \cdot S = F_i$, where S and F_i are sequences of items. As an example, it should not matter whether or not default values of the query parameters are specified. Let us consider a search in YouTube specifying no order; it should produce exactly the same result as indicating the default ordering, which is based on their relevance to the search query (`order=relevance`). Analogously, a search in Spotify specifying no page size should produce the same result as indicating the default maximum number of results to return (`limit=20`). Instances of this pattern can be produced with the default value of each request parameter.

This pattern can be generalized to represent those metamorphic relations where the source output must be equal to at least one follow-up output, i.e., $\exists i \in [1, n] \cdot S = F_i$. As an example of this type of relation, consider reordering the tracks of a playlist in Spotify. The operation receives several input parameters including the identifier of the playlist to be updated and several integers indicating how the tracks will be reordered. The request returns the updated playlist as an output. Intuitively, the tracks could be reordered in several steps assuring that the final order is identical to the original one. Fig. 2 shows an illustrative example with a playlist containing three tracks. On each test case, the track in the first position is moved to the last position until the complete list has been reverted and the test output of the source test case and third follow-up test case are identical, i.e., $S = F_3$. Note that this relation can be easily generalized by considering the total number of tracks in the playlist and the number of tracks to be reordered at each step.

3.3 Subset

This pattern groups those relations where the follow-up outputs should be subsets (or strict subsets) of the source output and subsets among them, i.e., $S \supseteq F_1 \supseteq F_2 \supseteq \ldots \supseteq F_n$. This pattern can be instantiated with any parameter filtering the results of a query. For instance, let us suppose a source test case consisting in a search for YouTube videos with the keywords "marathon" and geolocated within 50km from New York (parameters `location` and `locationRadius`). Next, a follow-up test case is constructed using the same query and restricting the search to videos within 20km from New York. Intuitively, the videos returned by the follow-up test case, within 20km from New York, should be a subset of those returned in the source test case, within 50km from New York. Note that this relation could be easily generalized by using different distances. For instance, if we construct another follow-up test case with the same query and restricting the search to videos within 5km from New York, the set of videos retrieved now should be a subset of those retrieved by the previous follow-up test case and by the source test case. This pattern is very common in query operations where most of the parameters are filters. The YouTube search operation, for instance, provides more than 20 parameters to filter results by language, definition, duration, publication time, license, caption, or region, among others.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering



(a) Source test output (b) Follow-up test output 1 (c) Follow-up test output 2 (d) Follow-up test output 3

Figure 2: Equality metamorphic relation in Spotify

Common query search operators are also a source of this type of metamorphic relations [20], [27]. As an example, consider a source test case consisting in a search for Spotify albums including the keywords "roadhouse OR blues" that returns all the results that match either "roadhouse" or "blues". Next, a follow-up test case is constructed searching for albums including the keyword "roadhouse". Intuitively, the follow-up output should be a subset of the source output.

3.4 Disjoint

This pattern represents those relations where the intersection among the source and follow-up outputs should be empty, i.e., $\forall i, j \in [1, n] \cdot S \cap F_i = \varnothing \wedge (i \neq j) \rightarrow (F_i \cap F_j = \varnothing)$. This pattern is applicable when query results can be divided into disjoint partitions, i.e, result sets with no item in common. For instance, YouTube videos can be classified according to its dimension in *2D* and *3D* videos. Let us suppose a source test case implementing a search for YouTube 2D videos (parameter `videoDimension=2d`). Next, a follow-up test case is constructed using the exact same query and restricting the search to 3D videos only (`videoDimension=3d`). Intuitively, the results of both searches should be disjoint since a video cannot be 2D and 3D at the same time. As a further example, consider a search for Spotify albums of "michael buble" published in 2012. Next, suppose another search for albums of the same artist published in 2014. The results of both searches should be obviously disjoint since the same album cannot be published in two different years. Note that this relation could be generalized by considering n different years leading to the creation of a source test case and $n - 1$ follow-up test cases.

Search operators may also be used to instantiate this pattern. As an example, consider a source test case consisting in a search for Spotify artists including the keywords "michael". Suppose a follow-up test case is constructed searching for artists with the query "jackson NOT michael" to return items that match "jackson" but excludes those that also contain the keyword "michael". The output of both test cases should have no items in common.

3.5 Complete

This pattern includes those relations where the union of the follow-up outputs should contain the same items as the source output, i.e., $S = F_1 \cup F_2 \cup \ldots \cup F_n$. Sometimes it may be necessary to tighten the relation to detect duplicated results by also checking that the number of items in the source output is equal to the number of items in the follow-up outputs, i.e., $|S| = \sum_{i=1}^{n} |F_i|$. This pattern is typically applicable when search results can be divided into disjoint and complete partitions. For instance, YouTube videos are classified according to its duration in *short* (less than 4 minutes), *medium* (between 4 and 20 minutes) and *long* videos (longer than 20 minutes). Consider a source test case consisting in a search for YouTube videos with the keyword "testing". Suppose that three follow-up test cases are constructed by searching for the same query restricting the search to short, medium, and long videos, respectively (parameter `videoDuration`). Intuitively, the union of the follow-up test outputs (short, medium, and long videos) should contain the same videos as the source test output, where no duration filter was specified. Interestingly, this relation involves three follow-up inputs, unlike most metamorphic relations found in the literature where a single follow-up input is typically used [11].

It is noteworthy that for this pattern to be instantiated the disjoint pattern must also be applicable, but not the other way around. For instance, YouTube searches can be restricted to a particular type of video, *movie* or *episode*, creating disjoint partitions. However, there may be videos not included in any of those categories and therefore this pattern could not be applied, i.e., the partitions are disjoint but not complete.

3.6 Difference

This pattern includes those metamorphic relations where the source output and the follow-up output should differ in a specific set of items D. This pattern is formally defined as $F_1 \setminus S = D$. This pattern is common in creation and update operations where the resource that is created or updated is returned as output (cf. Section 2.1). Note that in this particular scenario the output is not a set of resources, but the set of properties $name = value$ that compose the returned resource. Consider, as an example, the operation to insert (upload) a video in YouTube. The operation receives the video to be uploaded and a few metadata properties (e.g., description) as inputs. The operation returns a JSON representation of the video with about 180 properties, most of which are automatically generated such as those related to video quality, video dimension and video size. Checking for faulty values in so many properties is an error-prone and time-consuming task, where metamorphic testing could be helpful.

Consider the following sample instantiation of this pattern. A source test case is constructed by uploading a ten-minute-long video to YouTube with the description "ICSE video". Next, a follow-up test case is created by uploading a shorter version of the video with a duration of two minutes and the description "TSE video". Intuitively, the output of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

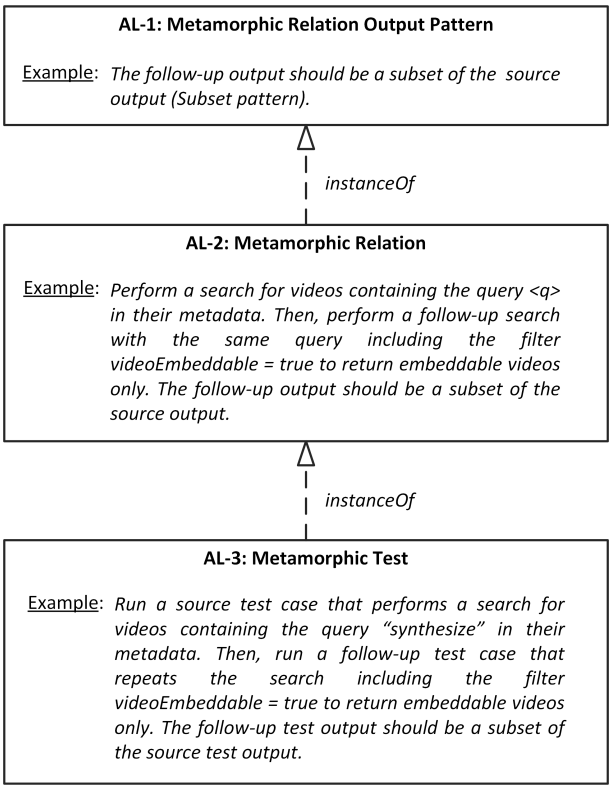


Figure 3: Different abstraction levels in our methodology

both test cases should only differ in the video description and the properties video duration and video size, remaining the rest of the properties equal such as those related to video quality, video dimension, default language and location. In practice, test outputs usually include several unique values that will also differ in every call (e.g., video identifier), but those properties can be simply ignored as explained in Section 5.2.3.

4 METHODOLOGY

In this section, we describe the proposed methodology to test a Web API using metamorphic testing. Roughly speaking, testers must go through different Abstraction Levels (AL), as illustrated in Fig. 3 using an example from the *Search* operation in the YouTube API. MROPs (AL-1) are the most abstract artefacts; they represent the relation among the source output and the follow-up output(s). Metamorphic relations (AL-2) are instantiated from the patterns by specifying the source input, follow-up input(s), and their relationship, in a way that it is expected to produce the output relation defined by the pattern. Finally, metamorphic tests (AL-3) instantiate the metamorphic relations by using specific input values and implementing the corresponding output assertion. Next, we present the specific steps to test a RESTful Web API using this methodology.

1) Select the API operation under test, e.g., reorder the tracks of a playlist in Spotify.
2) Select an input parameter or group of related input parameters of the API operation under test. For instance, when reordering tracks in Spotify three input parameters must be provided, namely, the position of the first track to be reordered (`range_start`), the number of tracks to be reordered (`range_length`), and the position where the tracks should be inserted (`insert_before`).
3) Check the input domain of the selected parameters and their default value. For instance, the parameter `range_length` is an integer in the domain $[0, n]$, where n is the number of tracks in the playlist being updated. Its default value is 1.
4) For each pattern, try to instantiate concrete metamorphic relations by analysing how the selected parameters could be modified so that their outputs satisfy the relation specified by the pattern. Unlike conventional metamorphic testing approaches, where the relation among inputs and outputs must be identified from scratch, MROPs simplify the process of identifying metamorphic relations significantly, by guiding the tester on the type of output relations typically found in Web APIs. For example, the following is the metamorphic relation instantiated from the *Equality* pattern depicted in Fig. 2:

   Run a source test case that moves one track (`range_length = 1`) from the position 0 (`range_start = 0`) to the last position (`insert_before = n`), where `n` is the number of tracks in the playlist being updated. Next, run `n` follow-up test cases that perform the exact same operation. The output playlist of the source test case and the n-th follow-up test case should contain the same tracks in the same order.

5) Implement one or more metamorphic test for each metamorphic relation by using concrete input values, e.g., specific playlists in the previous example. The generation of metamorphic tests can be automated by using standard input generators, e.g., a random word generator. This enables the generation of a parameterized number of tests.

5 EVALUATION

In this section, we assess the effectiveness of the proposed methodology and MROPs at revealing failures in RESTful Web APIs. In particular, we are interested in answering the following research questions (RQs).

- **RQ1: Are the proposed MROPs and methodology helpful for the identification of metamorphic relations in Web APIs?** We want to study whether the proposed patterns are representative of the types of metamorphic relations found in Web APIs, and whether the proposed methodology eases the identification of such relations.

- **RQ2: Are the identified metamorphic relations effective at revealing failures in Web APIs?** The capacity of the identified metamorphic relations to reveal failures is a key point to evaluate the value of the approach and we plan to answer this question

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

using both artificial and real faults.

- **RQ3: Is the approach cost-effective?** Testing approaches must not only be effective but also affordable. This will be evaluated in terms of the time and implementation effort needed to generate and execute the metamorphic tests in the subject Web APIs.

The rest of the section is structured as follows. Section 5.1 reports the evaluation with four Web APIs developed by computer science students, using seeded faults. Section 5.2 reports the results of the evaluation with Spotify and YouTube. Finally, the answer to the research questions is presented in Section 5.3.

5.1 Evaluation with artificial bugs

In this section, we study the effectiveness of our approach using artificial faults. In the following, we present the subject Web APIs, fault seeding process, testing setup and results.

5.1.1 Subject Web APIs

We selected four RESTful Web APIs developed by undergraduate students as a project assignment in the course on Architecture and Integration of Software Systems at the University of Seville (henceforth referred to as academic APIs). All APIs were written in Java using popular third-party libraries. The left-hand side of Table 1 shows the number of classes, lines of code (LoC), and number of creation, read and update operations provided by each API. Delete operations were not considered in our study since they accept the trivial oracle of searching for the deleted item and failing to find it, i.e., it does not suffer the oracle problem. The Comments API manages comment resources, composed of a text, a user name, and creation date. Events is an API for event management, where each event resource is composed of a starting and finishing date, a description, localization, and the name of the user who created it. Memes is an API for managing "memes", which are images with added text created for comedic purposes. The meme resource is composed of an URL, a weirdness value (calculated by checking the similarity with images indexed in Google), and a creation date. Finally, the Travel API combines travel information from different real sources. Each travel resource includes a set of flight booking, car sharing and lodging options.

5.1.2 Identification of metamorphic relations

The right-hand side of Table 1 shows the number of metamorphic relations instantiated from each MROP in the subject Web APIs. As illustrated, we identified between 5 and 14 metamorphic relations on each API, 33 relations in total. All the relations were identified by studying the documentation of the subject APIs, aiming to exercise all the operations and input parameters provided by the API at least once. We may remark, however, that it was not our intention to identify a complete set of metamorphic relations, and more relations could have been identified readily. To keep this paper at a reasonable size, the metamorphic relations identified are described in an online appendix [31].

5.1.3 Seeded faults

We introduced artificial faults into the subject Web API using mutation testing [32], [33]. More specifically, we used the tool muJava [34] to create many faulty versions (i.e., mutants) of the subject Web APIs, where each mutant was created from the original API by applying a syntactic change to its source code. Each syntactic change is determined by a so–called *mutation operator*. We used all the traditional (also called "method-level") mutation operators implemented in muJava, a total of 15. To keep the number of mutants manageable, we mutated the main class of each Web API only, leaving the rest of classes in their original form, e.g., data access classes.

The mutation process yielded 418 generated mutants. Out of these, 28 mutants (6.7%) were identified as semantically equivalent and were discarded. Equivalent mutants keep the program's semantics unchanged and therefore they cannot be detected by any test. Additionally, we discarded 73 mutants (17.5%) introducing faults in the exception handling code of the subject APIs, which is out of the scope of our approach. These bugs accept the trivial oracle of calling the service with wrong inputs and checking the corresponding error messages and HTTP error codes, and therefore they do not suffer the oracle problem. Table 2 (third column) depicts the final number of mutants used for the evaluation with each Web API, 317 mutants in total.

5.1.4 Testing setup

Each metamorphic relation was implemented into several metamorphic tests, where each metamorphic test runs the source and follow-up test cases with specific input values and checks the corresponding assertions. Source test cases were created in two steps. First, we manually inspected the data repositories of each subject API, and selected a set of between 1 and 5 random valid values for each input parameter. Then, we generated combinatorial combinations of the previous values in those operations receiving more than one parameter. As a result, we created 71 source test cases for the Comments API, 104 for the Events API, 54 for the Memes API, and 58 for the Travel API, yielding a total of 287 source test cases.

Once the metamorphic tests of each API were generated, they were executed on the original API to check if they all passed. This allowed us to identify one bug in the Comments API (descending ordering in query results working incorrectly) and one bug in the Memes API (storage API account expired, error not handled). Both faults were fixed and the mutants generated from scratch. Then, the metamorphic tests of each API were executed on each mutant. If any of the tests failed, the mutant was labelled as *killed*, otherwise it was labelled as *alive*. We found a few surviving mutants not detected by our tests. After analysing them, we realized that they could be easily killed by tightening the metamorphic relations. For instance, we found that in order to kill some mutants it was necessary to check that a result set was a strict subset of another result set, rather than a subset (which is true when both sets are equal). Similarly, some bugs required to check not only that two sets contained the same items but also the same number of them, to detect duplicated items. This led us to refine the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

Web API	Classes	LoC	Operations			Metamorphic Relation Output Patterns						
			Create	Read	Update	Equality	Equivalence	Subset	Disjoint	Complete	Difference	Total
Comments	5	438	1	2	1	2	0	2	1	1	2	8
Events	6	594	1	2	1	0	0	12	0	0	2	14
Memes	7	614	1	2	1	1	1	0	1	1	2	6
Travel	8	1,641	0	2	0	1	2	1	0	1	0	5
Total	**26**	**3,287**	**3**	**8**	**3**	**4**	**3**	**15**	**2**	**3**	**6**	**33**

Table 1: Metamorphic relations identified in the academic APIs

patterns *Subset* and *Complete*, as presented in Section 3.3 and Section 3.5, respectively.

We ran the evaluation on an Ubuntu 14.04 machine equipped with INTEL i7 with 8 cores running at 3.4 Ghz and 16 GB of RAM.

5.1.5 Testing results

Table 2 shows the results of the evaluation using mutation testing. For each subject API, the table shows the number of metamorphic test generated, total number of generated mutants, killed mutants, alive mutants, and mutation score, respectively. The mutation score is calculated as the ratio of killed mutants over the total number of mutants, excluding equivalent mutants [33]. As illustrated, the mutation score ranged between 72.7% in the Memes API, and 100% in the Comments and Events APIs, with an average value of 95.3%. All the identified metamorphic relations killed at least one mutant.

Web API	Met. tests	Mutants			
		Total	Killed	Alive	Score
Comments	71	81	81	0	100%
Events	104	58	58	0	100%
Memes	54	44	32	12	72.7%
Travel	58	134	131	3	97.8%
Total	**287**	**317**	**302**	**15**	**95.3%**

Table 2: Mutation testing results

We identified two types of faults that remained undetected. The first of them, which accounts for 8 out 15 alive mutants (53.3%), caused changes in the outputs that are independent of the input values. In particular, 8 mutants in the Memes API changed a line that affects a random id generator for the created memes. For instance, line `Math.random() * 10` was changed in a mutant to `Math.random() / 10`. This mutant changes the semantics of the program since the assigned ids will not be the expected ones. However, the change equally affects the source and follow-up test cases, and so the bug cannot be detected using metamorphic testing. We may remark that this is an intrinsic limitation of metamorphic testing, and not a limitation of the approach itself.

The second type of seeded fault in the surviving mutants, which affects 7 out of 15 non-killed mutants (46.6%), made the APIs return, for any input (i.e., both for the source and follow-up inputs), either an empty set or a set including all existing items. In this case, the comparison of the outputs satisfies most of the relations instantiated from the MROPs. For instance, an empty set is a subset of another empty set (*Subset* pattern satisfied). It is noteworthy, however, that any bug causing an API to always return an empty set or a set including all items should be trivially detected by any sensible manual test.

The set of metamorphic tests of each Web API took between 2 and 4 seconds to be executed in the original version of the corresponding API.

5.2 Evaluation with real bugs

This section reports the evaluation results with real Web APIs. In the following, we present the subject Web APIs, testing setup and results including a list of detected issues.

5.2.1 Subject Web APIs

We evaluated the effectiveness of metamorphic relations at detecting faults in the Web APIs of Spotify [9] and YouTube [16]. We selected these APIs due to their worldwide popularity, available issue tracking systems and good documentation. In particular, we tested six different API endpoints implementing creation, update and read operations over videos, playlists, and search resources (see Table 3). We placed a special emphasis on testing the search capabilities provided by Spotify and YouTube because they are the read operations with the largest number of input parameters, and therefore the ones from which more diverse metamorphic relations are likely to be derived.

5.2.2 Identification of metamorphic relations

Table 3 depicts the type and number of metamorphic relations identified. For each Web API, Spotify and YouTube, the table shows the operations under test and the number of relations instantiated from each MROP. In total, we identified 60 metamorphic relations, 20 in Spotify and 40 in YouTube. Each relation is composed of a source test case and from one to three follow-up test cases. All the metamorphic relations were identified by studying the online English documentation of the APIs and trying to have a representative number of instances of each MROP. As expected, most of the metamorphic relations were identified in the search operations due to the larger variety of input parameters. We recall that it was not our intention to identify an exhaustive set of metamorphic relations and, based on our experience, it would be straightforward to identify more instances of the proposed patterns. The metamorphic relations identified are described in an online appendix [31].

5.2.3 Testing setup

As in the evaluation with artificial faults, each metamorphic relation was implemented into several metamorphic tests,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

Web API	Operation	Metamorphic Relation Output Patterns						**Total**
		Equality	Equivalence	Subset	Disjoint	Complete	Difference	
Spotify	Create playlist	0	0	0	0	0	2	**2**
	Reorder playlist	4	0	0	0	0	0	**4**
	Search	1	0	10	2	1	0	**14**
YouTube	Insert video	0	0	0	0	0	10	**10**
	Search	4	4	6	6	6	0	**26**
	Update video	0	0	0	0	0	4	**4**
Total		**9**	**4**	**16**	**8**	**7**	**16**	**60**

Table 3: Metamorphic relations identifed in Spotify and YouTube

where each metamorphic test runs the source and follow-up test cases and checks whether the target relation is violated. In those services receiving numbers, strings or dates as input parameters, source test cases were automatically generated. In those services receiving videos and playlists as input parameters, we resorted to manual source test cases. In total, we report the results of 4,720 metamorphic tests (4,400 using random source test cases and 320 using manual source test cases). Several issues were considered during the implementation of the tests, namely:

- Input strings like search queries were constructed using one to five random words (single or compound) from an English dictionary. Linking words such as "of", "the" or "a" were excluded. Unfortunately, we noticed that this procedure rarely returned valid albums in Spotify searches. To address this issue, we constructed a custom database of album-related words as follows. First, we manually selected 100 artists among the top Spotify artists of each decade from 1960 to 2010. Then, we searched all their albums in Spotify and split their titles into a total of 1,897 different single words, excluding linking words.
- The data stored by Spotify and YouTube is frequently updated, which may lead to inconsistencies between the outputs of the source and follow-up test cases when performing searches. To address this issue, when a violation of a metamorphic relation was detected, we ran the very same metamorphic test case again, recording the result only if the violation was repeatable. This very same approach was used by Zhou et al. in their work on metamorphic testing of online search engines [18].
- To avoid inaccuracy caused by empty or too large search result sets (such as results being intentionally omitted to improve response time), only test cases returning between 1 and 50 items were used. Exceptionally, this threshold was raised to 500 items in two metamorphic relations (MR-47 and MR-54 in the online appendix [31]), and to 1,000 items in one relation (MR-19), where the limit of 50 was too restrictive to get valid results in a reasonable time. For instance, it was extremely difficult to find random search queries that return less than 50 YouTube results including both 2D and 3D videos. Metamorphic tests violating the constraint in the number of output items were discarded. A similar strategy was followed by Zhou et al. [18].
- During some preliminary tests we noticed that YouTube searches return duplicated items (see detected issues in Section 5.2.5). To avoid inaccuracy caused by this issue, duplicated items were removed from the results sets before checking each metamorphic relation.
- When a video is uploaded to YouTube, it takes some time (typically a few seconds) for it to be fully processed and all its data to be available. This could lead to inconsistencies between source and follow-up test cases not caused by actual faults. To address this issue, the output videos of source and follow-up test cases were compared only when the videos were labelled as *processed* by YouTube. Note that this was compatible with executing metamorphic tests twice when a violation was detected to confirm that the result was repeatable.
- Metamorphic relations derived from the *Difference* pattern check whether the outputs of source and follow-up test cases differ in a specific set of properties (cf. Section 3.6). We observed, however, that the output of insert and update operations may include up to 15 unique property values that change in every call, e.g., thumbnails' paths. Those properties were ignored when calculating the differences among outputs to avoid misleading results.
- Spotify and YouTube allow a limited number of API calls per user and day, as well as a maximum number of calls per period of time. Furthermore, testing Web services is noticeably slower than testing conventional applications due to the remote communication overhead. To avoid re-executing tests, the testing process was divided into two steps. First, we ran the tests recording the complete output of source and follow-up test cases as JSON files, available online [31]. Second, the files were loaded and analysed in detail. This makes our results fully replicable.

For the implementation of the tests we used the Google APIs Client Library for Java [35], the Spotify Web API library for Java [36], the Extended Java WordNet library [37] and the framework JUnit v4 [38].

5.2.4 Testing results

Table 4 summarizes the quantitative results of our evaluation. For each subject Web API and MROP, the table shows the number of metamorphic relations (total and violated), number of metamorphic tests and failure rate. Tests revealed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

	Spotify						YouTube					
	Met. Relations			Failure Rate (%)			Met. Relations			Failure Rate (%)		
MRP	Total	Violated	Met. Tests	Min	Avg	Max	Total	Violated	Met. Tests	Min	Avg	Max
Equality	5	0	180	0	0	0	4	3	400	0	2	3
Equivalence	-	-	-	-	-	-	4	4	400	99	99.50	100
Subset	10	2	1,000	0	4.80	24	6	4	600	0	1.67	4
Disjoint	2	0	200	0	0	0	6	1	600	0	0.16	1
Complete	1	0	100	0	0	0	6	6	600	1	13.17	63
Difference	2	0	40	0	0	0	14	1	600	0	7.14	100
Total	**20**	**2**	**1,520**	**0**	**0.96**	**24**	**40**	**19**	**3,200**	**0**	**5.53**	**100**

Table 4: Summary of results

2 violated metamorphic relations in Spotify (out of 20) and 19 violated metamorphic relations in YouTube (out of 40). Interestingly, violated metamorphic relations in YouTube are distributed among the six proposed patterns, which supports their usefulness. It is also worth noting that 17 out of the 20 YouTube metamorphic relations instantiated from the patterns *Equality*, *Equivalence*, *Subset* and *Complete* were violated. We observed that these violations were mainly caused by missing results in YouTube responses, perhaps to improve response time (specific examples are presented in the next section). This was unexpected considering the number of items returned in most searches was less than 50, and that source and follow-up test cases were executed sequentially and from the same computer. To the best of our knowledge, the YouTube API documentation does not mention the possibility of missing results in searches.

The failure rate measures the percentage of metamorphic tests that revealed a failure on each metamorphic relation. For each MROP, Table 4 depicts the minimum, maximum and average failure rates of the metamorphic relations instantiated from the pattern. For instance, the average percentage of failed metamorphic tests in the 10 metamorphic relations derived from the *Subset* pattern in Spotify was 4.80%, as shown in the sixth column of the "Subset" row. Failure rates varied among the different patterns, being especially low in the metamorphic relations derived from the *Equality*, *Subset* and *Disjoint* patterns in YouTube. This suggests that running more tests could reveal new issues. Conversely, the metamorphic relations instantiated from the *Equivalence* pattern were trivially violated in 99.5% of the metamorphic tests.

As previously mentioned, metamorphic tests were generated iteratively discarding those where the source or follow-up test cases produced results sets with zero or more than 50 (exceptionally 500 or 1,000) items. In total, 469,029 metamorphic tests were generated on the search for 4,720 that fulfil the previous constraints. The total execution time was about 286 hours, although most of the time was spent on the generation of discarded tests (464,309 in total). The execution of each metamorphic test took at most a few seconds.

5.2.5 Detected issues

This section describes the issues detected using our approach, reported in the issue tracking systems of Spotify [39] and YouTube [40]. All the issues were detected when investigating the causes of violations in metamorphic relations. At the time of writing this paper, most of the issues are easily reproducible through a browser or the "Try it!" Web interfaces provided by Spotify and YouTube as a part of the API documentation. A complete list of the detected issues is provided as supplemental material, including links to related or duplicated issues as well as screenshots of how the problems are reproduced [31]. The following issues have been detected in YouTube.

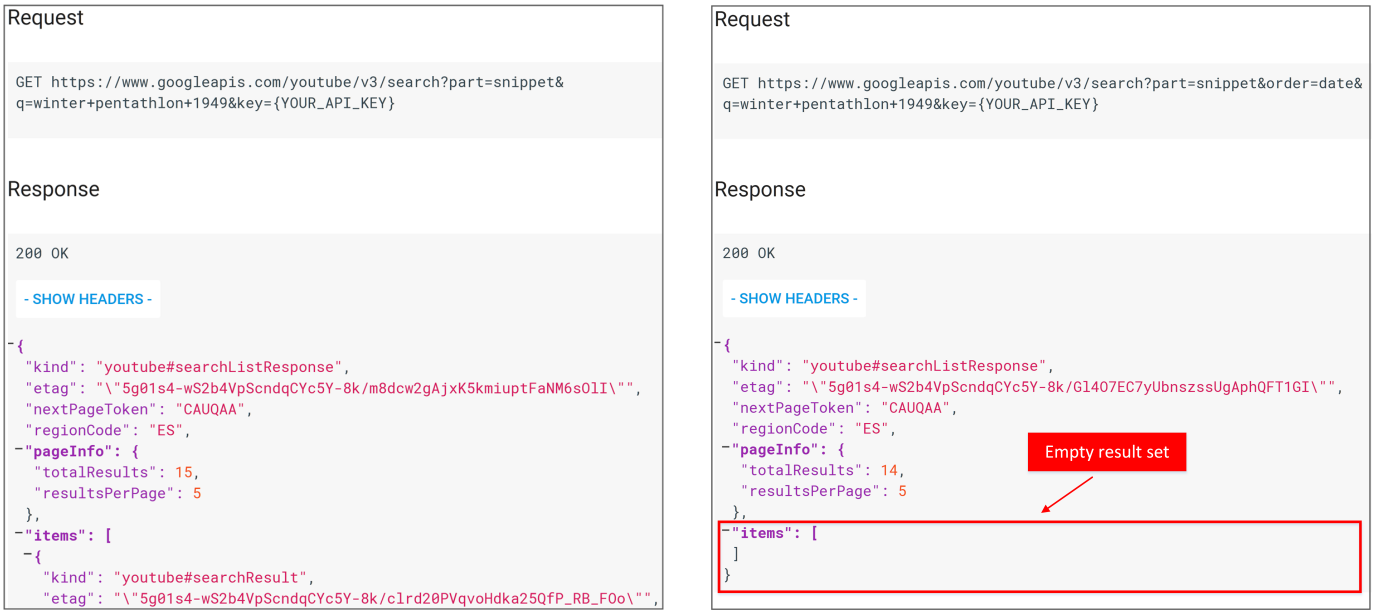
Issue 1. Missing results when using ordering parameters. Results are missing when requesting search results to be ordered by date, title, rating, or view counts. For instance, a search for "mistrustfully" returned 46 items. Immediately after, a new search was performed with the exact same query requesting the results to be ordered by date (`order=date`), returning 4 items. Fig. 4 depicts another metamorphic test reproducing this issue in the YouTube Web API interface. A search for "winter pentathlon 1949" returned 15 items (Fig. 4a). Then, a new search was performed with the same query requesting the results to be ordered by date, returning an empty result set (Fig. 4b). We reiterate that these inconsistencies are repeatable, and thus not caused by dynamic changes in the YouTube database. This issue was detected by all the metamorphic relations derived from the *Equivalence* pattern involving ordering parameters, with an average failure rate of 99.5%. The issue[5] was filed as a defect by YouTube back in 2013 and it has been voted (i.e., starred) by more than 30 users since then (see supplemental material for a list of duplicated issues [31]). The issue was labelled as "WontFix"[6] by YouTube in July 2016, without further explanation.

Issue 2. Missing results when using filters. 10 out of the 12 metamorphic relations instantiated from the patterns *Subset* and *Complete* were violated due to missing results either in the source or the follow-up test cases. For instance, a source test case was run searching for videos containing the keyword "*equilibrise*" in its metadata, returning 49 videos. Then, a follow-up test case was executed with the same query and restricting the search to short videos only (less than 4 minutes), returning 40 videos. It was observed that 8 of the videos found in the follow-up test case, where the search was restricted to short videos only, were not found in the source test case, where no video duration filter was introduced. Conversely, 2 of the short videos found

5. https://code.google.com/p/gdata-issues/issues/detail?id=5173

6. According to Bugzilla [41], the "WontFix" label indicates that "*the problem described is a bug which will never be fixed*", e.g., because it is too expensive to repair.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering



(a) Source test case (b) Follow-up test case (empty result set)

Figure 4: Metamorphic test revealing a bug in Youtube

in the source test case were not found in the follow-up test case, where the search was restricted to short videos only. This behaviour was observed in filter parameters such as `locationRadius`, `channelType`, `videoEmbeddable`, `videoDuration` and `videoType`, among others. We reported an issue for each parameter reproducing the problem (see supplemental material [31]). In October 2016, some of these issues were labelled as duplicated by YouTube developers and linked to another issue[7] reporting several acknowledged search-related bugs including, as described by YouTube developers, "*incomplete results - results (under the 500 results maximum) that should have been returned but were not*".

Issue 3. Publication time filters return unexpected results. Searches return items published outside the time range indicated by the `publishedBefore` filter parameter. For example, a source and follow-up test cases were generated searching for videos with the query "direfully" published before and after 1st May 2015 respectively. The issue was revealed when observing that the result sets of both searches were not disjoint (*Disjoint* pattern), that is, some videos were returned in both result sets. For instance, a video with publication date 25th May 2015 was returned in a search for videos published before 1st May 2015 (24 days of difference). This issue[8] was first reported in September 2015 and it has been voted by 6 users since then, including us.

Issue 4. Location filter returns unexpected results. The location filter returns results outside of the distance range requested. For instance, a search for videos containing the keyword "ingest" and geolocated within 100km from New York (40.7058,-74.2581) returned a video located 212km away from the target location (42.5722,-73.6985), according to Google Maps. This unexpected behaviour was observed while investigating the causes of the violations in the *Subset* metamorphic relations involving location parameters (Issue 2). This issue[9] was reported in April 2015 and reproduced in our work.

Issue 5. PublishAt property not editable. It is not possible to set a value for the property `status.publishAt` when updating a video, despite the fact that the documentation clearly states that the property is editable. This issue was detected by a *Difference* metamorphic relation, comparing the output of two test cases setting different values for the conflicting property. This issue[10] was first reported in December 2015 and it has been voted by 4 users since then, including us.

Issue 6. Inaccurate number of total results. The number of items in search results often deviates from the value of the JSON property `pageInfo.totalResults`, which should contain the approximate number of results in the response. This issue can be observed in Fig. 4b, where the response to the search indicates there are 14 results but, in fact, no item is returned. This was unexpected considering we restricted most of our tests to searches returning only 50 items as a maximum. In our evaluation, we found deviations in 26.8% of test cases (including source and follow-up test cases) being these divergences sometimes substantial. For instance, a search for YouTube videos of episodes including the keyword "straightaway" in its metadata returned a value for the property `totalResults` of 3,899, whereas the actual number of items in the result set was only 16. This behaviour was repeatedly observed on the metamorphic relations derived from the patterns *Equality*, *Equivalence*, *Subset*, *Disjoint* and *Complete*. To avoid any bias produced by this issue in our evaluation, we ignored the value of the

7. https://code.google.com/p/gdata-issues/issues/detail?id=8698
8. https://code.google.com/p/gdata-issues/issues/detail?id=7650
9. https://code.google.com/p/gdata-issues/issues/detail?id=7086
10. https://code.google.com/p/gdata-issues/issues/detail?id=7792

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

property and iterated on the number of real items in the results set, a slower and more expensive strategy. This issue[11] was acknowledged by YouTube developers in 2014, who in response updated the API documentation to clarify that the value of the property is only an approximation. The issue, voted by 50 users so far, was labelled as "Fixed" in July 2016. Incidentally, we found that the Spanish version of the API documentation describes the property `totalResults` as "*number of total results in the result set*" and does not mention that the value is an approximation. We reported this as a separated issue[12].

Issue 7. Duplicated results. Search results included duplicated items in 2% of the test cases, i.e, items with the same identifier. This percentage increases up to 21.6% in the metamorphic tests where the number of items in the result set was limited to 500, instead of 50. Although the documentation does not explicitly say that the returned items must be unique, it seems counter intuitive to provide redundant information to API users. This does not only consume extra usage quota but it also delegates in the user the burden of removing duplicates. This unexpected behaviour was recurrently observed when investigating violated metamorphic relations from the patterns *Equality*, *Equivalence*, *Subset*, *Disjoint* and *Complete*. This led us to adapt our implementation to ignore repeated items. This issue[13] was first reported in 2015 and it has been voted by 4 other users since then, including us.

The following issues were detected in Spotify:

Issue 8. Market filter returns unexpected results. The number of result items is expanded, rather than filtered, when restricting the search to only artists, albums, and tracks with content playable in a given market, i.e., country. This unexpected behaviour was detected by a metamorphic relation derived from the *Subset* pattern. Fig. 5 illustrates the issue being reproduced in the Spotify Web API interface: A search for albums with the query "banana boat" returned 35 items (Fig. 5a). Immediately after, a follow-up test case was generated searching for albums with the same query and content playable in Spain (`market=ES`), returning 45 items, 10 more items than in the previous search (Fig. 5b). We reported this issue[14] in May 2016 and it was labelled as "bug" shortly after.

Issue 9. Missing results when using AND/NOT search operators. This issue was detected, among others, by the following metamorphic test derived from the *Complete* pattern: A source test case searching for albums including "*mended*" returned 27 items. Two subsequent follow-up test cases were then generated searching for albums containing "*mended AND with*", returning 1 item, and "*mended NOT with*", returning 25 items. Unexpectedly, the union of the follow-up test cases was not complete ($27 \neq 1 + 25$), revealing the failure. In other words, the search for "mended" returned an album not found when searching for "mended AND with" and not found when searching for "mended NOT with" either. We reported this issue[15] to Spotify in July 2016. In November 2016, Spotify developers indicated that they had performed several improvements to the search operation and the issue was no longer reproducible.

The following issues were detected and reported to Spotify and YouTube during the preparation of our work, but they were fixed before we could reproduce them in the evaluation presented in this paper. It is noteworthy, however, that both issues have been confirmed by Spotify and YouTube developers, which supports the effectiveness of our approach.

Issue 10. Missing results on pagination. The number of items returned in a Spotify search varied along with the size of pagination. This bug was detected by a metamorphic relation derived from the *Equality* pattern. For instance, a search for albums with the query "redhouse" and a page size of 20 returned 21 total items (iterating over two pages). Immediately after, a new search was performed with the same query and a page size of 30, returning 27 items (6 more items than in the previous search). In practice, this was a critical bug that made not possible to iterate over the results of a search reliably. We reported this issue[16] in April 2016 and it was fixed and confirmed by Spotify developers two weeks later with the following message: "*We've rolled out a fix which should stabilise this, and I ran your examples successfully. Thanks for the detailed report.*".

Issue 11. Embeddable property not editable. The embeddable property was not editable when uploading a video to YouTube, which contradicted the API documentation. Its value was always set to true regardless of the input value provided by the user. This unexpected behaviour was detected by a *Difference* metamorphic relation, comparing the output of two test cases setting different values for the conflicting property. This issue[17] was filed as a defect by YouTube in 2013 and voted by 25 users so far. We reproduced the problem and reported it in April 2016. The issue's status was changed to "Fixed" by YouTube developers in July 2016.

5.3 Discussion

Based on the evaluation results, the research questions are answered as follows.

5.3.1 RQ1: Identification of metamorphic relations

In a classical metamorphic testing approach, metamorphic relations would have been identified from scratch, relying on the tester's creativity and knowledge about each API under test. In contrast, in this paper we propose several patterns sketching the general form of typical output relations found in RESTful Web APIs, regardless of the application domain. Using these patterns and the proposed methodology, the identification of metamorphic relations was straightforward, turning the general problem of identifying metamorphic relations into a much simpler one: deciding, for each API operation and input parameter, which pattern or patterns (out of the six proposed) fit best, and instantiating them. Following this procedure, we easily identified 33 metamorphic relations in the 4 academic Web APIs, and 60 metamorphic relations in the Web APIs of Spotify and YouTube. It is worth mentioning that we found a fair number

11. https://code.google.com/p/gdata-issues/issues/detail?id=6125
12. https://code.google.com/p/gdata-issues/issues/detail?id=8101
13. https://code.google.com/p/gdata-issues/issues/detail?id=7033
14. https://github.com/spotify/web-api/issues/230
15. https://github.com/spotify/web-api/issues/269
16. https://github.com/spotify/web-api/issues/225
17. https://code.google.com/p/gdata-issues/issues/detail?id=4861

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

(a) Source test case

(b) Follow-up test case (market filter applied)

Figure 5: Metamorphic test revealing a bug in Spotify

of instances of all the proposed patterns, which means that they are representative of the metamorphic relations found in Web APIs.

As explained in Section 3, each pattern defines a relation among the outputs of several service invocations, remaining the input relation undefined until the pattern is instantiated. We found that this approach is highly generic and intuitive, avoiding to narrow the scope of each pattern to a few input relations. For each MROP, Table 5 summarizes the input relations identified during the evaluation of our work, indicating the type of CRUD operation where they were identified. We remark that these represent just a sample of the types of input relations compatible with the proposed patterns. It is noteworthy, however, that these input relations are found in most Web APIs, which supports their applicability beyond the subject APIs.

Most of the metamorphic relations were identified in read operations. In particular, 67% (40 out of 60) of the relations in Spotify and YouTube were identified in the "search" operation. This was expected since the search operation was selected for being the read operation with the largest number of input parameters, and thus the one with a larger testing space. Also, this is in line with the proportion of read operations in the Web APIs under test and related APIs at the time of writing this manuscript: 64% (41 out of 64) in Spotify [9], 36% (18 out of 50) in YouTube [16], 69% (150 out of 218) in Flickr [42], and 57% (69 out of 121) in Twitter [43].

We hypothesize that the proposed patterns could also be helpful to automatically infer metamorphic relations for a given API. However, the automated discovery of metamorphic relations beyond numerical programs is still in an early stage, and so this remains as a challenge for future work [29], [30], [44]. Too keep the evaluation affordable, we did not follow a systematic adequacy coverage criterion such as testing certain combinations of input parameters e.g., pairwise testing. Instead, we tried to have a representative number of instances of each MROP in order to evaluate their effectiveness. Based on our experience, more relations could have been readily identified.

Pattern	Input relation	CRUD
Equivalence	Set ordering criterion	Read
Equality	Set page size	Read
	Use default input values	Read
	Set new ordering for the resource's items	Update
Subset	Set one ore more filters	Read
	Refine the search query	Read
Disjoint	Set disjoint filtering parameters	Read
	Perform disjoint search queries	Read
Complete	Set disjoint and complete filters	Read
Difference	Update a resource's property	Update
	Create resource with a specific property	Create

Table 5: Input relations identified in the evaluation

5.3.2 RQ2: Fault-detection effectiveness

The evaluation with seeded faults yielded a mutation score of 95.3% (302 mutants killed out of 317), plus two real bugs being detected in the academic APIs under test. This is a remarkable result considering metamorphic testing can only alleviate the oracle problem, but not remove it completely. This supports the effectiveness of our approach in revealing failures, and its applicability to different RESTful Web APIs.

Regarding the real issues detected, they are relevant both in qualitative and quantitative terms. Qualitatively, 10 of the issues (out of 11) have been confirmed either by developers (7 issues) or other users (8 issues), which means that they are meaningful and have a negative impact on the user experience. Most of the issues were detected in query operations since they are predominant in the Web APIs under test, however, some bugs were also revealed in creation (Issue #11) and update operations (Issue #5), supporting the effectiveness of the approach beyond read operations. The fact that some of the issues were fixed so quickly also supports their relevancy (Issue #10). Quantitatively, the number of detected issues is notorious considering

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

that only a small portion of the subject APIs were tested. More specifically, we tested three operations in the Spotify API (out of 64) and three operations in the YouTube API (out of 50), which represents a coverage of 4.7% and 6% respectively.

5.3.3 RQ3: Cost of the approach

The generation of mutants and metamorphic tests in the four academic APIs required an effort of about 0.2 person-month. Most of the time was invested to work around some limitations of the mutation tool (e.g., muJava removes all code annotations in mutated classes), and to implement code for running mutants in parallel. Regarding the evaluation with Spotify and YouTube, the initial setup for the generation of metamorphic tests required about 0.5 person-month in order to implement issues such as user authentication, random input generators, JSON files' handlers, integration of third-party libraries, and so on. It is noteworthy, however, that once everything was in place, the generation of metamorphic tests was fully automated and new metamorphic relations were implemented readily, paying off the initial cost. Also, based on our experience, much of the work made to implement metamorphic tests in one Web API can be reused when testing other APIs, e.g., authentication code. More importantly, once the metamorphic relations are identified and implemented, the degree of automation achieved can be total if source test cases are automatically generated, e.g., randomly. This includes not only the automated generation of inputs, but also the generation of their corresponding output assertions, i.e., oracles.

Regarding the execution of the metamorphic tests in Spotify and YouTube, it took about 12 days (286 hours) to generate almost half million test cases for the Web APIs of Spotify and YouTube. Several factors justified the time invested. First, testing Web services is noticeably slower than testing conventional applications due to the inevitable communication overhead. Second, as discussed in Issue #6, we noticed that the number of search results indicated in the outputs of YouTube was unreliable. As a workaround, we iterated over all the items of each result set to get accurate results, both in Spotify and YouTube, which added a significant time overhead. Last, most of the time was invested in the generation of tests that were discarded (464,309 out of 469,029), due to the hard constraints imposed in our evaluation, e.g., limiting the number of results in searches. These constraints were imposed to avoid potential biases caused by approximated results or performance optimizations not detailed in the user documentation of the Web APIs under test. Note, however, that these constraints could be loosen, or totally removed, if the Web APIs were tested by their own developers or third-party organisations with access to the specification, a very common scenario. This was demonstrated in the evaluation with academic APIs, where the specification and the code were available, and therefore no constraints were needed: 287 metamorphic tests were generated and run, without discarding a single test.

6 Threats to validity

The factors that could have influenced our work are summarized in the following internal and external validity threats.

6.1 Internal validity

Are there factors that might affect the results of this evaluation? The main internal threat of our work is related to the correct identification of failures. To mitigate this threat, we carefully analysed violated metamorphic relations until the failure was clearly reproduced both programmatically and, when possible, through the API Web interfaces of the applications under test. Next, we checked whether the detected issues had been already reported in the corresponding issue tracking system. If so, we added a comment and details about how to reproduce the problem. If no related issue existed, a new issue was created. Finally, we were cautious to indicate, for each issue, whether it shows an inconsistency between the API behaviour and its documentation (verification failure) or an unexpected behaviour from the user perspective, although not strictly contradictory with the documentation (validation failure).

6.2 External validity

What are the main limitations of the approach? The application of metamorphic testing requires identifying metamorphic relations, which is a manual task that demands a good knowledge of the functionality of the program under test. To ease the burden, we introduce the concept of metamorphic relation output pattern to describe abstract output metamorphic relations commonly found in Web APIs, regardless of their application domain. These patterns provide a helpful guidance for the identification of metamorphic relations, unlike standard metamorphic testing applications where the relations must be constructed from scratch. Besides this, the mutation testing results revealed a few bugs (7 out of 317) not detected by the relations instantiated from the patterns, e.g., bugs causing the operation to return an empty set for any input. It is noteworthy, however, that those are bugs that should be trivially detected by any sensible manual test.

To what extent is it possible to generalize the findings? The proposed patterns were instantiated in four academic Web APIs and two real Web APIs, Spotify and YouTube, and therefore they might not completely generalise further. We remark, however, that the proposed patterns are based on the principles of the REST architectural style and the standard practices for the design of effective RESTful Web APIs. Hence, the patterns should be applicable to any resource-oriented API, regardless of the type and domain of the resource. This means, for instance, that we could have as resources shopping orders (as in the eBay API), invoices (as in the PayPal API) or recording transcriptions (as in the Twilio API), among others. It is also noteworthy that the list of patterns is not complete and thus more patterns could be introduced when studying new Web APIs. Finally, although the effectiveness of the approach at detecting failures was assessed using a significant number of both artificial and real bugs, the results might not be generalized to other types of faults. It is remarkable, however, that 11 bugs were detected in the Web APIs of Spotify and YouTube, which are mature and commercial products with millions of users worldwide, which supports the effectiveness of our approach in real settings.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2764464, IEEE Transactions on Software Engineering

7 Related work

In this section, we summarize the pieces of work that are more closely related to our approach. We divide them into metamorphic testing of Web services and applications and general testing of Web services.

7.1 Metamorphic testing

Chan et al. [45] presented a metamorphic testing methodology for Service–Oriented Applications (SOA) built upon traditional Web services using the Web Service Description Language (WSDL). Their method relies on the use of so-called *metamorphic services* to encapsulate the services under test, execute source and follow-up test cases and check their results. Their work was evaluated on a service-oriented calculator with seeded faults. Similarly, Sun et al. [46] proposed to manually derive metamorphic relations from the WSDL description of Web services. Their technique automatically generates random source test cases from the WSDL specification and applies the metamorphic relations. They presented a tool to partially automate the process, and evaluated it with three Web services developed by the authors with seeded faults. In a related project, Castro-Cabrera and Medina-Bulo [47] presented a metamorphic testing-based theoretical approach for Web service compositions using the Web Service Business Process Execution Language (WS-BPEL). They proposed to analyse the XML description of the service composition to select adequate metamorphic relations. In contrast with their approaches, our work focuses on testing RESTful Web services as the dominant technological trend for software integration. Furthermore, we propose several patterns supporting the tester on the identification of metamorphic relations. Last, and more importantly, our approach was evaluated by detecting real issues in two commercial Web APIs.

In a related set of papers, Zhou et al. [18], [20], [27] used metamorphic testing for the detection of inconsistencies in online Web search applications. In their most recent work [18], the authors performed an extensive empirical study on the Web search engines Google, Bing, Chinese Bing and Baidu. Five metamorphic relations were manually identified and implemented using randomly generated source test cases. In [20], [27], Zhou et al. first introduced the concept of "general metamorphic relation" as an abstract metamorphic relation from which multiple metamorphic relations were derived. This is in line with the proposed patterns, and supports the effectiveness of providing abstract relations to facilitate the identification of metamorphic relations in a given domain. RESTful Web APIs often provide search functionalities and thus the metamorphic relations identified in their approach can also be used to test programmatic searches in Web APIs (Zhou et al. mentioned the use of "search APIs" to interact with the search engines [20]). However, several aspects make our work differ from theirs. First, their approach addresses the detection of failures in Web search engines while ours focuses on the detection of failures in RESTful Web APIs. Second, our approach aims to test create, read and update operations over resources, while theirs focuses on testing page retrieval and ranking in search engines, i.e, read operations only. Last, and more importantly, we propose six patterns, and an associated methodology, to ease the identification of metamorphic relations within the Web API domain. Out of these patterns, we instantiated 60 different relations in the Web APIs of Spotify and YouTube, which were effective at detecting real bugs. Overall, however, we believe that both lines of research are complementary and strengthen one another supporting the effectiveness of metamorphic testing at detecting faults in the Web domain.

Lindvall et al. [26], [48] presented a metamorphic testing approach to address acceptance testing of NASA's Data Access Toolkit (DAT). DAT is a huge database of telemetry data collected from different NASA missions, and an advanced query interface to search and mine the available data. Metamorphic testing was used by formulating the same query in different equivalent ways, and asserting that the resulting datasets are the same. Note that this is equivalent to the *Equality* pattern presented in this paper. A RESTful Web interface was used to interact with the DAT system. Compared to their work, we propose a richer catalogue of metamorphic relations exploiting other types of set relations as subset, disjoint and completeness. More importantly, they used a RESTful Web API to interact with the system under test (DAT), whereas in our work the Web API itself is the system under test.

7.2 Testing Web services

There exist some approaches that have proposed the testing of RESTful Web services and APIs. Pontes Pinheiro et al. [49] presented a model-based testing approach to automatically generate test cases for RESTful Web services. In their approach, the user needs to define a behavioural model of the RESTful Web service under test, specified with a protocol state machine model in terms of preconditions and post-conditions [50], [51]. The major drawback of this approach is that the generated test cases are based on the scenario defined and modelled manually by the user. Chakrabarti and Kumar presented *Test-the-REST* [52], an HTTP testing tool designed for testing RESTful Web services. The input test case has to be written in a test specification language based on XML, what may hinder its usability. A major difference of these approaches with ours is the effort required by the tester, who needs to deal with domain-specific languages for defining behavioural models and tests specifications, whereas this is not needed at all in our approach.

There are some other tools aimed at the testing of RESTful Web services, such as REST-assured [53], Postman [54], HttpMaster [55], vREST [56], soapUI [57] and API Fortress [15]. These tools enable the automated execution of tests (like JUnit does in Java) but it is still the tester who has to write the tests and deal with the oracle problem. In contrast, we propose a metamorphic testing approach to alleviate the oracle problem in Web APIs. Furthermore, we showed how metamorphic relations can be combined with random test data to achieve full test automation, i.e., input generation and output checking. We believe that integrating our approach into state-of-the-art tools for testing Web APIs would encourage testers to use it and we plan to work on that direction.

There are some surveys that compile several works on testing service-oriented architectures in general and Web

services in particular [58], [59]. Among the cited works for Web services testing, Coyote [60] is an XML-based object-oriented framework for such purpose, where test scripts are written in XML. Tsai et al. [61] proposed to extend the WSDL to include information relative to Web services testing, such as dependence information. Bai et al. [62] did not propose to extend WSDL, but they generated Web services test cases automatically from WSDL specifications. Ma et al. [63] also proposed the automated test data generation for Web services based on WSDL specifications. In particular, they focused their automation on the XML Schema datatypes found in the WSDL specifications. Differently from these approaches, focused on testing WSDL-based Web Services, we address the detection of faults on RESTful Web Services, as the emerging standard in industry.

8 Conclusions

In this paper, we presented a metamorphic testing approach for the detection of faults in RESTful Web APIs. In particular, we presented six patterns (MROPs) capturing the shape of typical metamorphic relations found in Web APIs. Each pattern is defined in terms of set relations among API responses and can be instantiated into one or more concrete metamorphic relations on each Web API under test. A methodology is proposed for the identification of metamorphic relations based on the proposed patterns, broadening the scope of our contribution beyond a particular Web API. For the evaluation of our work, we identified 33 metamorphic relations in four academic Web APIs and 60 metamorphic relations in the Web APIs of Spotify and YouTube. The identified relations were implemented using random and manual test data running thousands of automated metamorphic tests. These tests were effective at revealing both, automatically seeded and real bugs. In particular, 11 issues were revealed in Spotify and YouTube, most of them confirmed, supporting the effectiveness and the value of the approach in realistic settings.

Verifiability

For the sake of verifiability, we provide an online appendix including the source code of the academic Web APIs, mutants and description of the metamorphic relations [31]. Additionally, the appendix includes the following data from the evaluation with real APIs: (1) links to the issues reported in the tracking systems of Spotify and YouTube (including links to related and duplicated issues), (2) list of metamorphic relations revealing each issue, (3) screenshots illustrating how each issue is reproduced, and (4) inputs and outputs of the metamorphic tests used in the evaluation (JSON and CSV format).

Acknowledgment

We would like to thank Spotify and YouTube developers for their outstanding work and active support in the resolution of issues. We would also like to thank the anonymous reviewers for their helpful comments and suggestions. This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project BELI (TIN2015-70560-R), and the Andalusian Government project COPAS (P12-TIC-1867).

References

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

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

[3] D. Jacobson and S. Narayanan, "Netflix api : Top 10 lessons learned," in *Open Source Convention (OSCON)*, Porland, Oregon, July 2014. [Online]. Available: http://www.slideshare.net/danieljacobson/top-10-lessons-learned-from-the-netflix-api-oscon-2014

[4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000.

[5] "ProgrammableWeb API Directory," accessed November 2016. [Online]. Available: http://www.programmableweb.com/

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *Software Engineering, IEEE Transactions on*, vol. 41, no. 5, pp. 507–525, May 2015.

[7] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *Software Engineering, IEEE Transactions on*, vol. 40, no. 1, pp. 4–22, Jan 2014.

[8] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[9] "Spotify Web API," accessed November 2016. [Online]. Available: https://developer.spotify.com/web-api/

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, Tech. Rep., 1998.

[11] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortes, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, Sept 2016.

[12] M. Masse, *REST API Design Rulebook*. O'Reilly Media, 2011. [Online]. Available: http://books.google.ch/books?id=eABpzyTcJNIC

[13] L. Richardson and S. Ruby, *Restful Web Services*, 1st ed. O'Reilly, 2007.

[14] S. Allamaraju, *RESTful Web Services Cookbook*. O'Reilly, 2010.

[15] "API FORTRESS," accessed November 2016. [Online]. Available: http://apifortress.com

[16] "YouTube Data API v3," accessed November 2016. [Online]. Available: https://developers.google.com/youtube/v3/

[17] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*, 1st ed. O'Reilly Media, Inc., 2010.

[18] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 264–284, March 2016.

[19] T. Y. Chen, P. Poon, and X. Xie, "METRIC: METamorphic Relation Identification based on the Category-choice framework," *Journal of Systems and Software*, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001624

[20] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.437

[21] W. K. Chan, J. C. F. Ho, and T. H. Tse, "Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 89–120, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1002/stvr.v20:2

[22] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[23] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BioMed Central Bioinformatics Journal*, vol. 10, no. 1, p. 24, 2009. [Online]. Available: http://www.biomedcentral.com/1471-2105/10/24

[24] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic testing for cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, June 2016.

[25] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," in *Second International Conference on Software Testing Verification and Validation, ICST 2009*, 2009.

[26] M. Lindvall, D. Ganesan, R. Ardal, and R. Wiegand, "Metamorphic model-based testing applied on nasa dat – an experience report," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2, May 2015, pp. 129–138.

[27] Z. Q. Zhou, T. H. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of web search engines in the absence of an oracle," Department of Computer Science, The University of Hong Kong, Tech. Rep. TR-2007-06, 2007.

[28] C. Murphy, G. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," Department of Computer Science, Columbia University, New York NY, Tech. Rep., 2008.

[29] F. Su, J. Bell, C. Murphy, and G. Kaiser, "Dynamic inference of likely metamorphic properties to support differential testing," in *Proceedings of the 10th International Workshop on Automation of Software Test*, ser. AST '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 55–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819261.2819279

[30] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Software Testing, Verification and Reliability*, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1594

[31] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Cortés, "Automated metamorphic testing of restful web APIs: Supplemental material," December 2016. [Online]. Available: https://gestionproyectos.us.es/projects/restfulmt/wiki

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218136

[33] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.62

[34] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system," *Software Testing Verification and Reliability*, vol. 15, no. 2, pp. 97–133, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1002/stvr.v15:2

[35] "Google APIs Client Library for Java," accessed November 2016. [Online]. Available: https://developers.google.com/api-client-library/java

[36] "Spotify Web API library for Java," accessed November 2016. [Online]. Available: https://github.com/thelinmichael/spotify-web-api-java

[37] "Extended Java WordNet Library," accessed November 2016. [Online]. Available: http://extjwnl.sourceforge.net/

[38] "JUnit 4," accessed November 2016. [Online]. Available: http://junit.org/junit4/

[39] "Spotify Web API Issue Tracking System," accessed November 2016. [Online]. Available: https://github.com/spotify/web-api/issues

[40] "YouTube Data API Issue Tracking System," accessed November 2016. [Online]. Available: https://code.google.com/p/gdata-issues/issues/list?q=label:APi-YouTube

[41] "Bugzilla," accessed November 2016. [Online]. Available: https://bugzilla.mozilla.org

[42] "Flickr API," accessed July 2017. [Online]. Available: https://www.flickr.com/services/api/

[43] "Twitter REST API," accessed July 2017. [Online]. Available: https://dev.twitter.com/rest

[44] J. Troya, S. Segura, and A. Ruiz-Cortés, "Automated inference of likely metamorphic relations for model transformations," *Journal of Systems and Software*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300870

[45] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung, "A metamorphic testing approach for online testing of service-oriented software applications." *International Journal of Web Services Research*, vol. 4, no. 2, pp. 61–81, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/jwsr/jwsr4.html#ChanCL07

[46] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "A metamorphic relation-based approach to testing web services without oracles," *International Journal of Web Services Research*, vol. 9, no. 1, pp. 51–73, Jan. 2012. [Online]. Available: http://dx.doi.org/10.4018/jwsr.2012010103

[47] C. Castro-Cabrera and I. Medina-Bulo, "Application of metamorphic testing to a case study in web services compositions," in *E-Business and Telecommunications*, ser. Communications in Computer and Information Science, M. Obaidat, J. Sevillano, and J. Filipe, Eds. Springer Berlin Heidelberg, 2012, vol. 314, pp. 168–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35755-8_13

[48] M. Lindvall, D. Ganesan, S. Bjorgvinsson, K. Jonsson, H. S. Logason, F. Dietrich, and R. E. Wiegand, "Agile metamorphic model-based testing," in *Proceedings of the 1st International Workshop on Metamorphic Testing (in conjuction with ICSE)*, ser. MET '16. New York, NY, USA: ACM, 2016, pp. 26–32. [Online]. Available: http://doi.acm.org/10.1145/2896971.2896979

[49] P. V. Pontes Pinheiro, A. Takeshi Endo, and A. Simao, "Model-based testing of restful web services using uml protocol state machines," in *7th Brazilian Workshop on Systematic and Automated Software Testing*, ser. SAST'13, 2013.

[50] I. Porres and I. Rauf, "Modeling behavioral restful web service interfaces in uml," in *Proc. of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. ACM, 2011, pp. 1598–1605.

[51] I. Rauf and I. Porres, "Designing level 3 behavioral restful web service interfaces," *SIGAPP Appl. Comput. Rev.*, vol. 11, no. 3, pp. 19–31, 2011.

[52] S. K. Chakrabarti and P. Kumar, "Test-the-rest: An approach to testing restful web-services," in *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, ser. COMPUTATIONWORLD '09. IEEE Computer Society, 2009, pp. 302–308.

[53] "REST Assured," accessed November 2016. [Online]. Available: http://rest-assured.io

[54] "POSTMAN," accessed November 2016. [Online]. Available: https://www.getpostman.com

[55] "HttpMaster," accessed November 2016. [Online]. Available: http://www.httpmaster.net

[56] "vREST," accessed November 2016. [Online]. Available: https://vrest.io

[57] C. Kankanamge, *Web services testing with soapUI*. Packt Publishing Ltd, 2012.

[58] G. Canfora and M. Di Penta, *Service-Oriented Architectures Testing: A Survey*. Springer, 2009, pp. 78–105.

[59] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing web services: A survey," Tech. Rep., 2010.

[60] W. T. Tsai, R. Paul, W. Song, and Z. Cao, "Coyote: an xml-based framework for web services testing," in *Proc. 7th IEEE International Symposium on High Assurance Systems Engineering*, 2002, pp. 173–174.

[61] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending wsdl to facilitate web services testing," in *Proc. 7th IEEE International Symposium on High Assurance Systems Engineering*, 2002, pp. 171–172.

[62] 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)*, 2005, pp. 207–212.

[63] C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai, "Wsdl-based automated test data generation for web service," in *International Conference on Computer Science and Software Engineering*, vol. 2, 2008, pp. 731–737.