RESTful API Automated Test Case Generation with EvoMaster

ANDREA ARCURI, Kristiania University College, Norway and University of Luxembourg, Luxembourg

RESTful APIs are widespread in industry, especially in enterprise applications developed with a microservice architecture. A RESTful web service will provide data via an API over the network using HTTP, possibly interacting with databases and other web services. Testing a RESTful API poses challenges, because inputs/outputs are sequences of HTTP requests/responses to a remote server. Many approaches in the literature do black-box testing, because often the tested API is a remote service whose code is not available. In this article, we consider testing from the point of view of the developers, who have full access to the code that they are writing. Therefore, we propose a fully automated white-box testing approach, where test cases are automatically generated using an evolutionary algorithm. Tests are rewarded based on code coverage and fault-finding metrics. However, REST is not a protocol but rather a set of guidelines on how to design resources accessed over HTTP endpoints. For example, there are guidelines on how related resources should be structured with hierarchical URIs and how the different HTTP verbs should be used to represent well-defined actions on those resources. Test-case generation for RESTful APIs that only rely on white-box information of the source code might not be able to identify how to create prerequisite resources needed before being able to test some of the REST endpoints. Smart sampling techniques that exploit the knowledge of best practices in RESTful API design are needed to generate tests with predefined structures to speed up the search. We implemented our technique in a tool called EvoMaster, which is open source. Experiments on five open-source, yet non-trivial, RESTful services show that our novel technique automatically found 80 real bugs in those applications. However, obtained code coverage is lower than the one achieved by the manually written test suites already existing in those services. Research directions on how to further improve such an approach are therefore discussed, such as the handling of SQL databases.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*; *Search-based software engineering*;

Additional Key Words and Phrases: Software engineering, REST, web service, testing

ACM Reference format:
Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 28, 1, Article 3 (January 2019), 37 pages.
https://doi.org/10.1145/3293455

This work is supported by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No. 274385) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 694277).
Author's address: A. Arcuri, Kristiania University College, Oslo, Norway, University of Luxembourg, Luxembourg; email: andrea.arcuri@kristiania.no.

1049-331X/2019/01-ART3 $15.00
https://doi.org/10.1145/3293455

1 INTRODUCTION

Modern web applications often rely on external *web services*, such as REST [26] or SOAP [23]. Large and complex enterprise applications can be split into individual web service components, in what is typically called *microservice* architectures [41]. The assumption is that individual components are easier to develop and maintain compared to a monolithic application. The use of microservice applications is a very common practice in industry, done in companies such as Netflix, Uber, Airbnb, eBay, Amazon, Twitter, and Nike [45].

Besides being used internally in many enterprise applications, there are many web services available on the Internet. Websites such as *ProgrammableWeb*[1] currently list more than 16,000 Web APIs. Many companies provide APIs to their tools and services using REST, which is currently the most common type of web service, such as Google[2], Amazon[3], Twitter[4], Reddit[5], and LinkedIn[6]. In the Java ecosystem, based on a survey[7] of 1,700 engineers, better REST support (together with HTTP/2 support) was voted as the most desired feature in the next version of Java Enterprise Edition (at that time, JEE 8). This is because, according to that survey, "The current practice of cloud development in Java is largely based on REST and asynchrony." Furthermore, according to a StackOverflow survey, Cloud Backend is the type of job for which there is the most demand but least supply of qualified candidates[8].

Testing web services—in particular, RESTful web services—poses many challenges [17, 20]. Different techniques have been proposed. However, most of the work so far in the literature has been concentrating on black-box testing of SOAP web services and not REST. SOAP is a well-defined protocol based on XML. However, most enterprises are currently shifting to REST services, which usually employ JavaScript Object Notation (JSON) as the data format for message payloads. Furthermore, there is not much research on white-box testing of web services because that requires having access to the source code of those services.

In this article, we propose a novel approach that can automatically generate integration tests for RESTful web services. Our technique has two main goals: maximizing code coverage (e.g., statement coverage) and finding faults using the HTTP return statuses as an automated oracle. We aim at testing RESTful services in isolation, which is the typical type of testing done directly by the engineers while developing those services. This is a first step needed before system testing of whole microservices can be done. Furthermore, we do not deal with the black-box testing driven by users that want to verify if third-party web services available on the Internet work correctly as publicized. The need for automating the writing of white-box integration tests is based on our personal experience of working as test engineers in industry [7], writing this kind of test manually.

To generate the tests, we employ an evolutionary algorithm, in particular, the MIO algorithm [4]. Furthermore, we aim at improving the performance of the search algorithm by exploiting typical characteristics of how RESTful APIs are designed. A web service that provides resources over HTTP can be be written in many different ways, without necessarily following the semantics of HTTP. For example, resources could be created with a HTTP DELETE method, and HTTP GETs could have side-effects. However, REST specifies a set of *guidelines* [2, 26] on how resources should be structured with hierarchical URIs, with operations on those resources based on the semantics

[1] https://www.programmableweb.com/api-research.
[2] https://developers.google.com/drive/v2/reference/.
[3] http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.
[4] https://dev.twitter.com/rest/public.
[5] https://www.reddit.com/dev/api/.
[6] https://developer.linkedin.com/docs/rest-api.
[7] http://www.infoworld.com/article/3153148/java/oracle-survey-java-ee-users-want-rest-http2.html.
[8] https://stackoverflow.blog/2017/03/09/developer-hiring-trends-2017/.

of the HTTP verbs (e.g., new resources should be created only with POSTs or PUTs). This type of domain knowledge can be exploited to improve the testing process. For example, if to test an HTTP endpoint X we first need to have a resource Y, this information can be derived by analyzing their URIs; thus, we can create test templates in which we first create Y (by doing an HTTP POST or PUT, if any exists for Y). A test case will be a sequence of HTTP calls, either chosen at random (to enable the exploration of the search landscape) or following one of our predefined templates. Note: even if the structure of a test is fixed, there is still the need to search for its right input data, for example, HTTP headers, URL parameters, and payloads for POST/PUT/PATCH methods (typically in JSON and XML formats).

We implemented a tool prototype called EvoMaster and carried out experiments on five open-source RESTful web services. Those systems range from 718 to 7,534 lines of code, for a total of 22,420 (not including tests). Results of our experiments show that our novel technique did automatically find 80 real faults in these systems. However, code coverage results are relatively low compared to the coverage obtained by the existing manually written tests. This is due mainly to the presence of interactions with SQL databases. Further research will be needed to address these issues to improve performance even further.

This article provides the following research and engineering contributions:

- We designed a novel technique that is able to generate effective test cases for RESTful web services.
- We propose a method to automatically analyze, export, and exploit white-box information of these web services to improve the generation of test data.
- We propose a novel approach in which search-based techniques can be extended by using smart sampling of test template structures that exploit domain knowledge of RESTful API design.
- We presented an empirical study on non-trivial software showing that even if our tool is in an early prototype stage, it can automatically find 80 real faults in those RESTful web services.
- To enable replicability of our results, tool comparisons, and reuse of our algorithm implementations, we released our tool prototype under the open-source LGPL license and provided it on the public hosting repository GitHub[9].

This article is an extension of a previous conference version [5]. In addition to a larger case study, this article also introduces a new smart sampling technique that successfully exploits domain knowledge of RESTful API design. Furthermore, a tool paper describing how to use EvoMaster was presented in [6].

The article is organized as follows. Section 2 provides background information on the HTTP protocol, RESTful APIs, and search-based software testing. Section 3 discusses related work. Our novel approach for test case generation of RESTful APIs is presented in Section 4. Details of our novel smart sampling technique are discussed in Section 5. Section 6 discusses our empirical study. Threats to validity are discussed in Section 7. Conclusions are presented in Section 8.

2 BACKGROUND

2.1 HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol for communications over a network. HTTP is the main protocol of communication on the World Wide Web. The HTTP

[9]https://github.com/arcuri82/EvoMaster.

protocol is defined in a series of Requests for Comments (RFC) documents maintained by Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), such as RFC 7230[10] and RFC 7231[11].

An HTTP message is usually sent over TCP, and is composed of four main components:

Verb/Method: The type of operation to perform, such as getting a specific web page.

Resource path: An identifier to specify on which resource the HTTP operation should be applied, such as the path of an HTML document to get.

Headers: Extra metadata, expressed as a list of key/value pairs. An example of metadata is the *accept* header, which is used to specify the format (e.g., HTML, XML, or JSON) in which the resource should be returned (a resource could be available in different formats).

Body: The payload of the message, such as the HTML text of a web page that is returned as a response to a get request.

The HTTP protocol allows the following operations (i.e., verbs/methods) on the exposed resources:

GET: The specified resource should be returned in the body part of the response.

HEAD: Like GET, but the payload of the requested resource should not be returned. This is useful if one needs to check only whether a resource exists or whether one needs to get only its headers.

POST: Send data to the server, for example, the text values in a web form. Often, this method is the one used to specify that a new resource should be created on the server.

DELETE: Delete the specified resource.

PUT: Replace the specified resource with a new one, provided in the payload of the request.

PATCH: Do a partial update on the given resource. This is in contrast to PUT, where the resource is fully replaced with a new one.

TRACE: Echo the received request. This is useful to determine whether a given HTTP request has been modified by intermediates (e.g., proxies) between the client and the server.

OPTIONS: List all available HTTP operations on the given resource. For example, it could be possible to GET an HTML file but not DELETE it.

CONNECT: Establish a tunneling connection through an HTTP proxy, usually needed for encrypted communications.

When a client sends an HTTP request, the server will send back an HTTP response with headers and possibly a payload in the body. Furthermore, the response will also contain a numeric, three-digit status code. There are five groups/families of codes, specified by the first digit:

1xx: Used for provisional responses, such as confirming the switching of protocol (101) or that a previous conditional request in which only the headers were sent should continue to send the body as well (100).

2xx: Returned if the request was handled successfully (200). The server could further specify that a new resource was created (201), for example, as a result of a POST command, or that nothing is expected in the response body (204), for example, as a result of a DELETE command.

[10] https://tools.ietf.org/html/rfc7230.
[11] https://tools.ietf.org/html/rfc7231.

3xx: Those codes are used for redirection, for example, to tell the client that the requested resource is now available at a different location. The redirection could be just temporary (307) or permanent (301).

4xx: Used to specify that the user request was invalid (400). A typical case is requesting a resource that does not exist (404), or trying to access a protected resource without being authenticated (401) or authorized (403).

5xx: Returned if the server cannot provide a valid response (500). A typical case is if the code of the business logic has a bug and an exception is thrown during the request processing, which is then caught by the application server (i.e., the whole server is not going to crash if an exception is thrown). However, this kind of code could also be returned if the needed external services (e.g., a database) are not responding correctly. For example, if the hard-drive of a database breaks, a server could still be able to respond with a 500 code HTTP, even though it cannot use the database.

The HTTP protocol is *stateless*: each incoming request needs to provide all the information needed to be processed, because the HTTP protocol does not store any previous information. To maintain the state for a user doing several related HTTP requests (e.g., think about a shopping cart), *cookies* are a commonly employed solution.

Cookies are just HTTP headers with a unique id created by the server to recognize a given user. Users need to include such headers in all of their HTTP requests.

2.2 REST

For many years, the main way to write a web service was to use the Simple Object Access Protocol (SOAP), which is a communication protocol using XML-enveloped messages. However, recently, there has been a clear shift in industry toward Representational State Transfer (REST) when developing web services. All major players are now using REST, such as Google[12], Amazon[13], Twitter[14], Reddit[15], and LinkedIn[16].

The concepts of REST were first introduced in a highly influential PhD thesis [26] in 2000. REST is not a protocol (as is SOAP) but rather is a set of architectural guidelines on building web services on top of HTTP. Such client–server applications need to satisfy some constraints to be considered RESTful, such as being stateless, and the resources should explicitly state if they are cacheable or not. Furthermore, resources should be identified with a URI. The representation of a resource (JSON or XML) sent to the client is independent from the actual format of the resource (e.g., a row in a relational database). These resources should be managed via the appropriate HTTP methods; for example, a resource should be deleted with a DELETE request and not a POST request.

Let us consider an example of a RESTful web service that provides access to a product catalog. Possible available operations could be:

GET/products (return all available products)
GET/products?k=v (return all available products filtered by some custom parameters)
POST/products (create a new product)
GET/products/{id} (return the product with the given id)

[12]https://developers.google.com/drive/v2/reference/.
[13]http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.
[14]https://dev.twitter.com/rest/public.
[15]https://www.reddit.com/dev/api/.
[16]https://developer.linkedin.com/docs/rest-api.

GET/products/{id}/price (return the price of a specific product with a given id)
DELETE/products/{id} (delete the product with the given id)

Elements within curly brackets {} are *path parameters*, that is, they represent variables. For example, an endpoint for "/products/{id}" would match requests for "/products/5" and "/products/foo," but not "/products/5/bar."

Given hierarchical URIs, a collection of elements (e.g., "/products") could be represented with a resource with a plural name. Creating a new element on such a collection would be done with a POST. Individual elements could be identified by id (e.g., the same id used in the database). As such elements are hierarchically related to the collection, accessing them would be done with a URI that extends the one of the collection, that is, "/products/{id}." Given an empty state (e.g., an empty database), before testing a resource like "GET /products/{id}," a tester would need first to do a "POST /products" to create the element that is needs.

Note that those URIs do not specify the format of the representation returned. For example, a server could provide the same resource in different formats, such as XML or JSON, which should be specified in the headers of the request.

Another aspect of REST is the so-called Hypermedia As The Engine Of Application State (HATEOAS), where each resource representation should also provide links to other resources (in a similar way as links in web pages). For example, when calling a *GET /products*, all products should be returned and there should also be links to what other methods are available. Ideally, given the main entry point of an API, the API should be fully discoverable by using those links. However, the use of HATEOS is quite rare in practice [46], mainly owing to the lack of a proper standard on how links should be defined (e.g., a JSON or XML schema) and the extra burden that it would put on the clients.

As REST is not a protocol but just a set of architectural guidelines [26], the ubiquitous term "REST" has often been misused in practice. Many Web APIs over HTTP have been called and marketed as REST, although strictly speaking they cannot be considered as fully REST [26, 46]. Although in this article we use the term REST, the presented novel techniques would work as well to any Web API that is accessed via HTTP endpoints and where the payload data is expressed in a language such as JSON or XML.

2.3 Search-Based Software Testing

Test data generation is a complex task, because software can be arbitrarily complex. Furthermore, many developers find it tedious to write test cases. Therefore, there has been a lot of research on how to automate the generation of high-quality test cases. One of the easiest approaches is to generate test cases at random [9]. Although it can be effective in some contexts, random testing is not a particularly effective testing strategy, as it might cover just small parts of the tested software. For example, it would not make much sense to use a naïve random-testing strategy on a RESTful API, because it would be extremely unlikely that a random string would result in a valid, well-formed HTTP message.

Among the different techniques proposed throughout the years, search-based software engineering has been particularly effective at solving many different kinds of software engineering problems [32]—in particular, software testing [1]—with tools such as EvoSuite [27] for unit testing and Sapienz [38] for Android testing.

Software testing can be modeled as an optimization problem, in which one aims to maximize the code coverage and fault detection of the generated test suites. Then, once a fitness function is defined for a given testing problem, a search algorithm can be employed to explore the space of all possible solutions (test cases in this context).

There are several kinds of search algorithms, Genetic Algorithms (GAs) perhaps being the most famous. In a GA, a population of individuals is evolved for several generations. Individuals are selected for reproduction based on their fitness value and then go through a crossover operator (mixing the material of both parents) and mutations (small changes) when sampling new offspring. The evolution ends either when an optimal individual is evolved or the search has run out of the allotted time.

In the particular case of generating test suites, there are different specialized search algorithms, such as the Whole Test Suite (WTS) [29], MOSA [43], and MIO [4].

2.4 The MIO Algorithm

The Many Independent Objective (MIO) algorithm [4] is an evolutionary algorithm designed to improve the scalability of test-suite generation for non-trivial programs with a very large number of testing targets (e.g., in the order of thousands/millions). It is tailored around the following three main assumptions in test-case generation: (1) testing targets (e.g., lines and branches) can be sought independently because test-suite coverage can be increased by adding a new test case; (2) testing targets can be either strongly related (e.g., nested branches) or completely independent (e.g., when covering different parts of the system under test [SUT]); and (3) some testing targets can be *infeasible* to cover.

Based on the above assumptions, at a high level the MIO algorithm works as follows: it keeps one population of tests for *each* testing target (e.g., branches). Individuals within a population are compared and ranked based on their fitness value computed *exclusively* for that testing target. At the beginning of the search, all populations are empty and are iteratively filled with generated tests. At each step, with a given certain probability, MIO either samples new tests at random or samples (and then mutates) one test from one of the populations related to uncovered targets. A sampled test is added to *all* of the populations for uncovered targets and is evaluated and ranked independently in each population. Once the size of a population increases over a certain threshold (e.g., 10 test cases), the worst test (based on its fitness for that population) is removed. Whenever a target is covered, its population size is shrunk to one and no more sampling is done from that population. At the end of the search, a test suite is created based on the best tests in each population.

To improve performance, MIO employs a technique called *feedback-directed sampling*. For each population, there is a counter, initialized to zero. Every time an individual is sampled from a population X, its counter is increased by one. Every time a new, better test is successfully added to X, the counter for that population is reset to zero. When sampling a test from one of the populations, the population with the lowest counter is chosen. This helps focus the sampling on populations (one per testing target) for which there has been a recent improvement in the achieved fitness value. This is particularly effective to prevent spending significant search time on infeasible targets [4]. Furthermore, to dynamically balance the trade-off between *exploration* and *exploitation* of the search landscape, MIO changes its parameters during the search. Similarly to Simulated Annealing, MIO allows a wider exploration of the landscape at the beginning of the search. Then, with the passage of time, it becomes more *focused*. For example, by the time the focused search starts, the population sizes are shrunk to 1, and the probability of sampling a random test is 0. The pseudo-code of MIO is listed in Algorithm 1.

It is important to note that, although MIO is specialized for test-suite generation, it is still a search algorithm that needs to be instantiated for each problem domain. This means that, for each domain (e.g., the testing of RESTful APIs), there is still the need to define an adequate *problem representation*, custom *search operators* on such representation, and, finally, a *fitness function* to drive the search.

ALGORITHM 1 : High-Level Pseudo-code of the Many Independent Objective (MIO) Algorithm [4]

```
Input: Stopping condition $C$, Fitness function $\delta$, Population size limit $n$, Probability of random
    sampling $P_r$, Start of focused search $F$
Output: Archive of optimized individuals $A$
 1: $T \leftarrow$ SetOfEmptyPopulations()
 2: $A \leftarrow \{\}$
 3: while $\neg C$ do
 4:     if $P_r >$ rand() then
 5:         $p \leftarrow$ RandomIndividual()
 6:     else
 7:         $p \leftarrow$ SampleIndividual($T$)
 8:         $p \leftarrow$ Mutate($p$)
 9:     end if
10:     for all $k \in$ ReachedTargets($p$) do
11:         if IsTargetCovered($k$) then
12:             UpdateArchive($A, p$)
13:             $T \leftarrow T \setminus \{T_k\}$
14:         else
15:             $T_k \leftarrow T_k \cup \{p\}$
16:             if $|T_k| > n$ then
17:                 RemoveWorstTest($T_k, \delta$)
18:             end if
19:         end if
20:     end for
21:     UpdateParameters($F, P_r, n$)
22: end while
23: return $A$
```

3 RELATED WORK

Canfora and Di Penta provided a discussion on the trends and challenges of Service-Oriented Architecture (SOA) testing [19]. Later, they provided a more in depth survey [20]. There are different kinds of testing for SOA (unit, integration, regression, robustness, etc.), which also depend on which stakeholders are involved, for example, service developers, service providers, service integrators, and third-party certifiers. Bertolino et al. [13, 15] also discussed the trends and challenges in SOA validation and verification.

Successively, Bozkurt et al. [17] carried out a survey as well on SOA testing, in which 177 papers were analyzed. One of the interesting results of this survey is that, although the number of papers on SOA testing has been increasing over the years, only 11% of those papers provide any empirical study on actual real systems. In 71% of the cases, no experimental result at all was provided, not even on toy case studies.

A lot of the work in the literature has been focusing on black-box testing of SOAP web services described with Web Services Description Language (WSDL). Different strategies have been proposed, such as [10, 14, 31, 36, 37, 39, 42, 48, 49, 51]. If those services also provide a semantic model (e.g., in OWL-S format), that can be exploited to create more "realistic" test data [16]. When in SOAs the service compositions are described with Web Services Business Process Execution

Language (BPEL), different techniques can be used to generate tests for those compositions [33, 50].

Black-box testing has its advantages, but also has its limitations. Coverage measures could improve the generation of tests but, often, web services are remote and there is no access to their source code. For testing purposes, Bartolini et al. [12] proposed an approach in which feedback on code coverage is provided as a service without exposing the internal details of the tested web services. However, the insertion of the code coverage probes had to be done manually. A similar approach has been developed by Ye and Jacobsen [52]. In our approach in this article, we do provide code coverage as a service as well, but our approach is fully automated (e.g., based on on-the-fly bytecode manipulation).

Regarding RESTful web services, Chakrabarti and Kumar [21] provided a testing framework with "automatic generation test cases corresponding to an exhaustive list of all valid combinations of query parameter values." Seijas et al. [35] proposed a technique to generate tests for RESTful API based on an idealized, property-based test model. Chakrabarti and Rodriquez [22] defined a technique to formalize the "connectedness" of a RESTful service and generated tests based on that model. When formal models are available, techniques such as in [44] and in [25] can be used as well.

Segura et al. [47] presented a set of metamorphic relations that could be used as automated oracles when testing RESTful APIs. Our technique is significantly different from those approaches, because it does not need the presence of any formal model, can automatically exploit white-box information, and uses an evolutionary algorithm to guide the generation of effective tests. To the best of our knowledge, there is no other work that aims at generating white-box integration tests for RESTful APIs.

Regarding the use of evolutionary techniques for testing web services, Di Penta et al. [24] proposed an approach for testing Service Level Agreements (SLAs). Given an API in which a contract is formally defined stating "for example, that the service provider guarantees to the service consumer a response time less than 30 ms and a resolution greater or equal to 300 dpi," an evolutionary algorithm is used to generate tests to break those SLAs. The fitness function is based on how far a test is from breaking the tested SLA, which can be measured after its execution.

4 PROPOSED APPROACH

In this article, we propose a novel approach to automatically generate test cases for RESTful API web services. We consider the testing of a RESTful service in isolation and not as part of an orchestration of two or more services working together (e.g., as in a microservice architecture). We consider the case in which our approach to automatically generate test cases is used directly by the developers of such RESTful services. As such, we assume the availability of the source code of these developed services. The goal is to generate test cases with high code coverage and that can detect faults in the current implementation of those services, which is a typical case sought in industry [7].

To generate test cases, we use the MIO algorithm [4], which is a search algorithm tailored for test-suite generation for integration/system testing (see Section 2.4). To use MIO (or any search algorithm) on a new problem, one needs to define the *problem representation*, the *search operators* on such representation, and, finally, the *fitness function* to drive the search. The final output of our technique is a test suite, which is a collection of executable test cases.

In the rest of this section, we will discuss those details, together with several technical challenges when dealing with RESTful APIs.

4.1 Problem Representation

In the integration testing of RESTful APIs, our final solution is a test suite composed of one or more test cases. The MIO algorithm evolves individual test cases that are stored in an archive. Only at the end of the search is the final test suite created. Thus, we need to define how to represent test cases for RESTful APIs.

In our context, a test case is one or more HTTP requests to a RESTful service. The test data can hence be seen as a string or byte array, representing the HTTP request. We need to handle all of the components of an HTTP request: HTTP verb, HTTP headers, path parameters, query parameters, and body payloads. Such test data can be arbitrarily complex. For example, the payload in the body section could be in any format. As currently JSON is the main format of communication in RESTful APIs, in this article, we will focus just on that format, which we fully support. Handling other less popular formats, such as XML, would be a matter of engineering effort.

At any rate, before being able to make an HTTP request, we need to know what API methods are available. In contrast to SOAP, which is a well-defined protocol, REST does not have a standard to define the available APIs. However, a very popular tool for REST documentation is Swagger[17], which is currently available for more than 25 different programming languages. Another tool is RAML[18], but it is less popular. When a RESTful API is configured with Swagger, it will automatically provide a JSON file as a resource that will fully define which APIs are available in that RESTful service. Therefore, the first step, when testing a RESTful service, is to retrieve its Swagger JSON definition.

Figure 1 shows an extract from a Swagger definition of one of the systems that we use in the empirical study. The full JSON file is more than 2,000 lines of code. The figure shows the definition for two HTTP operations (GET and PUT) on the same resource. To execute a GET operation on a resource, two values are needed: a numeric “id,” which will be part of the resource path, and an optional query parameter called “attrs.” For example, given the template */v1/activities/{id}*, one could make a request for */v1/activities/5?attrs=x*.

The PUT operation also needs an “id” value but not the optional parameter “attrs.” However, in its HTTP body, it can have a JSON representation of the resource to replace, which is called “ActivityProperties” in this case. Figure 2 shows the object definition. This object has many fields of different types, such as numeric (e.g., “id”), strings (e.g., “name”), dates (e.g., “date_published”), arrays (e.g., “tags”), and other objects (e.g., “author”). When a test case is written for a PUT operation, in addition to specifying an “id” in the path, one would also need to instantiate the “Activity-Properties” object and marshall it as a JSON string to add in the body of the HTTP request.

To handle all of these cases, when the Swagger definition is downloaded and parsed, for each API endpoint we automatically create a set of *chromosome templates*. These will consist of both fixed, non-mutable information (e.g., the IP address of the API, its URL path if it does not have any variable path parameter, and all needed HTTP headers, such as *content-type* when sending payloads) and a set of mutable *genes*. The latter represent the different variable parts of the HTTP requests, such as the query parameters and the content of the body payload. For each gene, we explicitly specify whether it is used to represent a query parameter (including the name of such parameter), a path parameter, body payload, or an HTTP header.

To fully represent what is available from the Swagger schema and the JSON specification, we needed to define the following kinds of *genes* (listed in alphabetical order). To greatly simplify the design of the search operators (discussed in the next section), some specific kinds of genes might contain other genes inside them in a tree-like structure. The phenotype of a gene will be based

[17] http://swagger.io.
[18] http://raml.org.

```
"/v1/activities/{id}": {
      "get": {
        "tags": ["activities"],
        "summary": "Read a specific activity",
        "description": "",
        "operationId": "get",
        "produces": ["application/json"],
        "parameters": [
          {"name": "id",
            "in": "path",
            "required": true,
            "type": "integer",
            "format": "int64"},
          {"name": "attrs",
            "in": "query",
            "description": "The attributes to include in the response. Comma-separated list.",
            "required": false,
            "type": "string"}
        ],
        "responses": { "default": { "description": "successful operation"}}},
      "put": {
        "tags": ["activities"],
        "summary": "Update an activity with new information.",
        "description": "",
        "operationId": "update",
        "produces": ["application/json"],
        "parameters": [
          {"name": "id",
            "in": "path",
            "required": true,
            "type": "integer",
            "format": "int64"},
          {"in": "body",
            "name": "body",
            "required": false,
            "schema": {"$ref": "#/definitions/ActivityProperties"}
          }
        ],
        "responses": {
          "200": {
            "description": "successful operation",
            "schema": {"$ref": "#/definitions/Activity"}
          }
        }
      }
```

Fig. 1. Swagger JSON definition of two operations (GET and PUT) on the */v1/activities/{id}* resource.

on all of its contained genes (if any). For example, a *DateTime* is composed of two different genes (*Date* and *Time*) that, in turn, are composed of three *Integer* genes for a total of $1 + 2 + 3 + 3 = 9$ genes. When a *DataTime* gene is used in an HTTP call, a well-formed string representing the date with the time will be built (using the RFC3339 format[19]).

Genes can also have specific constraints based on the Swagger schema, such as minimum and maximum values for each numeric variable. The search operators will take that information into account when sampling and mutating the genes. Definitions for the following terms are provided to further an understanding of the sections to come.

- *Array*: Containing zero or more genes, all of the same type (e.g., all *Integer* genes). To avoid test cases that are too large (e.g., millions of elements), an upper bound to the size of the array is specified (e.g., 5).
- *Base64String*: A gene representing a string, whose phenotype must be encoded in the Base-64 format.
- *Boolean*: Either representing *true* or *false*.

[19]https://www.ietf.org/rfc/rfc3339.txt.

```
"ActivityProperties": {
  "type": "object",
  "properties": {
    "id": {"type": "integer", "format": "int64"},
    "name": {"type": "string", "minLength": 0, "maxLength": 100 },
    "date_published": {"type": "string", "format": "date-time"},
    "date_created": {"type": "string", "format": "date-time"},
    "date_updated": {"type": "string", "format": "date-time"},
    "description_material": {"type": "string","minLength": 0, "maxLength": 20000},
    "description_introduction": {"type": "string","minLength": 0, "maxLength": 20000},
    "description_prepare": {"type": "string","minLength": 0, "maxLength": 20000},
    "description_main": {"type": "string","minLength": 0, "maxLength": 20000},
    "description_safety": {"type": "string","minLength": 0, "maxLength": 20000},
    "description_notes": {"type": "string","minLength": 0, "maxLength": 20000},
    "age_min": {"type": "integer", "format": "int32","maximum": 100.0},
    "age_max": {"type": "integer", "format": "int32","maximum": 100.0},
    "participants_min": {"type": "integer", "format": "int32"},
    "participants_max": {"type": "integer", "format": "int32"},
    "time_min": {"type": "integer", "format": "int32"},
    "time_max": {"type": "integer", "format": "int32"},
    "featured": {"type": "boolean", "default": false},
    "source": {"type": "string"},
    "tags": {
       "type": "array",
       "xml": {"name": "tag", "wrapped": true},
       "items": {"$ref": "#/definitions/Tag"}},
    "media_files": {
       "type": "array",
       "xml": {"name": "mediaFile", "wrapped": true},
       "items": {"$ref": "#/definitions/MediaFile"}},
    "author": {"$ref": "#/definitions/User"},
    "activity": {"$ref": "#/definitions/Activity"}
  }
}
```

Fig. 2. Swagger JSON definition of a complex object type.

- *CycleObject*: JSON objects are expressed in a tree structure, where one object can reference/contain other objects. However, it might happen that an object A has reference to an object B, and B has reference to A', where A' might or might not be equal to A. To avoid this kind of infinite loop when generating JSON data representing graphs, we use this special gene when in a chain of references we encounter the same object type for the second time.
- *Date*: Composed of three *Integer* genes, representing the *year*, *month*, and *date*. All of these genes are constrained within valid values (e.g., months can have values only from 1 to 12) and a couple of non-valid ones (e.g., invalid month 0 and 13).
- *DateTime*: A gene composed of a *Date* and a *Time* gene.
- *Disruptive*: A gene containing another gene, but where the probability of mutating the gene is controlled with a specific probability, which can be set to 0 (and thus prevent all mutations on it once initialized with some specific value). This type of gene was important to handle some cases of URL path parameters that should not be modified once set (this will be discussed in more detail in Section 5).
- *Double*: Representing double-precision numbers.
- *Enum*: Representing one value from a fixed set of possibilities (typically, strings or integers).
- *Float*: Representing float numbers.
- *Integer*: Representing integer numbers.
- *Long*: Representing long numbers.
- *Map*: Representing a map of gene values of the same type, where the map keys are strings. Note that this is a specialization of the *Object* gene type.
- *Object*: Representing a JSON object, which is represented with a map of heterogeneous genes, where each field name in the object is a key in the map.

- *Optional*: A gene containing another gene, whose presence in the phenotype is controlled by a Boolean value. This is needed to represent optional query parameters.
- *String*: Representing a sequence of characters, bounded within a minimum (e.g., 0) and maximum (e.g., 16) length.
- *Time*: Composed of three *Integer* genes, representing the *hour*, *minute*, and *second*. Similarly to the *Date* gene, values are constrained among valid ones and a couple of invalid ones.

4.2 Search Operators

When a new test case is randomly sampled, 1 to n (e.g., $n = 10$) HTTP calls are created, selected randomly based on the set of chromosome templates derived from the Swagger schema (see Section 4.1). The values in the *genes* will be chosen at random, within the constraints (e.g., min and max values) defined for each gene.

During the search, the MIO algorithm does not use crossover, and the modification of test cases happens only via the *mutation* operator. The mutation operator will do small modifications on each test case.

When a test case is mutated, two different kinds of mutations can be applied: either a mutation that changes the *structure* (adding or removing a HTTP call) or the values in the existing genes are mutated. Each gene has a probability of $1/k$ to be mutated, where k is the total number of genes in all of the HTTP calls in the test case.

Mutation operations of the different kind of genes are similar to what has been done in the literature of unit testing. For example:

- Boolean values are simply flipped.
- Integer numbers get modified by a $\pm 2^i$ delta, where i is randomly chosen between 0 and a max value that decreases during the search (e.g., from an initial 30 to down to 10).
- For float/double values, the same kind of $\pm 2^i$ delta is applied, but multiplied by a Gaussian value (mean 0 and standard deviation 1), where $i = 0$ has higher chances (e.g., 33%) to be selected compared to the other values.
- Strings need special treatment because how to best modify them strongly depends on the fitness function. In our context, either one character is randomly modified, the last character is dropped, or a newly created character is appended at the end of the string.

4.3 Fitness Function

Each test case will cover one or more testing targets in the SUT. In our case, we consider three types of testing targets for the fitness function: (1) coverage of statements in the SUT; (2) coverage of bytecode-level branches; and (3) returned HTTP status codes for the different API endpoints (i.e., we want to cover not only the "happy-day" scenarios such as 2xx, but also user errors and server errors, regardless of the achieved coverage). When two test cases have the same coverage, MIO prefers the shorter one.

Each target will have a numerical score in $[0, 1]$, where 1 means that it is covered. Values different from 1 means that the target is not covered, where a heuristics is defined to check whether it is close to be covered (i.e., values close to 1) or far from it (i.e., values close to 0), once all of the HTTP calls in a test case are executed.

As MIO keeps a population of individuals *for each* testing target separately, there is no need to aggregate the scores on all targets in a test case. In other words, individuals (i.e., test cases) in a population for target t are compared based solely on the heuristic value $[0, 1]$ collected for t.

To generate high-coverage test cases, coverage itself needs to be measured. Otherwise, it would not be possible to check whether a test has higher coverage than another one. Albeit returned

HTTP status codes can be easily checked in the response messages, white-box metrics such as statement and branch coverage require access to the source/bytecode. Thus, when the SUT is started, it needs to be *instrumented* to collect code coverage metrics. How to do it will depend on the programming language. In this article, for our prototype, we started by focusing on JVM languages, for example, Java.

Coverage metrics can be collected by automatically adding probes in the SUT. This is achieved by instantiating a Java Agent that intercepts all class loadings and then add probes directly in the bytecode of the SUT classes. This process can be fully automated by using libraries such as *ea-agent-loader*[20] (for Java Agent handling) and *ASM*[21] (for bytecode manipulation). This approach is the same as that used in unit test generation tools for Java, such as EvoSuite [27].

Measuring coverage is not enough. Knowing that a test case covers 10% of the code does not tell us how more code could be covered. Often, code is not covered because it is inside blocks guarded by *if* statements with complex predicates. Random input data is unlikely to be able to solve the constraints in such complex predicates. This is a very well-known problem in search-based unit testing [40]. A solution to address this problem is to define *heuristics* that measure how far test data is to solve a constraint. For example, given the constraint $x == 0$, although neither 5 nor 1,000 solves the constraint, the value 5 is *heuristically* closer than 1,000 to solving it. The most famous heuristic in the literature is the so-called *branch distance* [34, 40]. In our approach, we use the same kind of branch distance used in unit testing by automatically instrumenting the Boolean predicates when the bytecode of a class is loaded for the first time (same way as for the code coverage probes). In addition to primitive types, we also use customized branch distances for handling string objects [3]. This is achieved by replacing all Boolean methods on string objects (e.g., `equals`, `startsWith` and `contains`) with instrumented versions that calculate custom heuristics (e.g., based on Euclidian distance when comparing whether two strings are equal) besides returning a Boolean value.

The branch distance from the literature has been shown to be a good approach to help smooth the search landscape for branch coverage. However, it does not help to handle the case of statement coverage when an exception is thrown in a code block. For example, consider a code block composed of 10 statements, with test case A throwing an exception at line 3, whereas test case B throws it at line 8. The branch distance will give no gradient to prefer B over A to be able to fully cover all of the 10 statements in the block.

Our novel (as far as we know) yet quite simple approach to handle it is to give, for each statement (or, more precisely, each method call in the bytecode), a heuristic value of $h = 0.5$ when it is reached and $h = 1$ when its execution is completed with no thrown exception. If a statement is never reached, then its heuristic value is by default $h = 0$. When both test cases A and B are executed, all of the targets for lines 1 to 7 will have a maximum heuristic score $h = 1$ (i.e., they are covered), where line 8 would have the score $h = 0.5$ (with test case B), and lines 9 and 10 will have $h = 0$ because they were never reached. Because line 8 has a test case with $h = 0.5$, the search will try to mutate the test to see whether it can avoid throwing an exception and possibly reach line 9. However, there is no gradient to remove the cause for which the exception is thrown, apart from penalizing tests such as A that do not even reach line 8. This would require further research and likely more sophisticated heuristics.

Regarding the returned HTTP status codes, at the moment there is no gradient in the fitness function: we either cover them ($h = 1$) or not ($h = 0$). However, explicitly modeling those codes as testing targets will enable the search to avoid losing tests that cover new status codes (because

[20] https://github.com/electronicarts/ea-agent-loader.
[21] http://asm.ow2.org/.

such tests would then be saved in the archive). It is important to note that, if returning a specific code depends on the execution of a specific part of the SUT code we want to test (e.g., a statement that directly completes the handling of the HTTP request by returning a specific HTTP status code), there might be a gradient provided by the code coverage heuristics (e.g., the branch distance). However, not all HTTP requests lead to executing the business logic of the SUT. When applications are written with frameworks such as Spring and JEE, an HTTP request could be completed directly in such frameworks before any of developers' code that we want to test is executed. This could happen due to input validation when constraints are expressed with Java annotations, for example. By explicitly modeling the HTTP status codes as testing targets, we can handle these situations where code coverage would not help retain tests covering new HTTP status codes.

Most of the work on white-box instrumentation in the literature targets unit testing. However, when dealing with integration/system testing, there are several engineering research challenges to address to make the new approaches scalable to real-world systems. For example, even if one can measure code coverage and branch distances by using bytecode manipulation (e.g., for JVM languages), there is still the question of how to retrieve the values. In unit testing, a test data generation tool would run in the same process as the one where the tests are evaluated; thus the values could be read directly. It would be possible to do the same for system testing: the testing tool and the SUT could run in the same process, for example, the same JVM. However, such an approach is not optimal because it would limit the applicability of a test-data generation tool to only RESTful services written in the same language. Furthermore, there could be third-party library version conflicts between the testing tool and the SUT. As the test cases would be independent of the language in which a RESTful API is written (as they are just HTTP calls), focusing on a single language is an unnecessary limitation.

Our solution is to have the testing tool and the SUT running in different processes. When the SUT is run, we provide a library with functionalities to automatically instrument the SUT code. Furthermore, the library itself would automatically provide a RESTful API to export all of the coverage and branch distance information in a JSON format. The testing tool, when generating and running tests, would use the API to determine the fitness of these tests. The testing tool would be just one; then, for each target programming language (e.g., Java, C#, and JavaScript), we would just need its library implementation for the code instrumentation.

This approach would not work well with unit testing: the overhead of an HTTP call to an external process would be simply too great compared to the cost of running a unit test. On the other hand, in system-level testing, an entire application (a RESTful web service in our case) runs at each test execution. Although non-zero, the overhead would be more manageable, especially when the SUT itself has complex logic and interacts with external services (e.g., a database).

Although the overhead of instrumentation is more manageable, it still needs to be kept under control. In our novel approach, we consider the following two optimizations:

- When the SUT starts, the developer has to specify which packages to instrument. Instrumenting all the classes loaded when the SUT starts would be far too inefficient. For example, there is no point in collecting code coverage metrics on third-party libraries, such as application servers (e.g., Jetty or Tomcat), or ORM libraries, such as Hibernate.
- By default, when querying the SUT for code coverage and branch distance information, not all information is retrieved: only the information of newly covered targets, or better branch distance, is returned. The reason is that if the SUT is 100,000 lines of code, then you do not want to un/marshal JSON data with 100,000 elements at each single test execution. The testing tool will ask explicitly for which testing targets it needs information for. For example,

if a target is fully covered with an existing test, there is no point in collecting information for that target when generating new tests aimed at covering the other remaining targets.

4.4 Oracle

When automatically generating test cases with a white-box approach—for example, trying to maximize statement coverage—there is the problem of what to use as an automated *oracle* [11]. An oracle can be considered as a function that tells whether the result of a test case is correct or not. In manual testing, the developers decide what should be the expected result for a given test case and write the expectation as an assertion check directly in the test cases. In automated test generation, where many hundreds (if not thousands) of test cases are generated, asking the developers to write such assertions is not a viable option.

There is no simple solution for the oracle problem, just different approaches with different degrees of success and limitations [11]. In system-level testing, the most obvious automated oracle is to check whether the whole system under test crashes (e.g., a segmentation fault in C programs) when a test is executed. The test case would have detected a bug in the software, but not all bugs lead to a complete crash of an application (likely, just a small minority of bugs are of this kind). Another approach is to use formal specifications (e.g., pre/post conditions) as automated oracles, but those are seldom used in practice.

In unit testing, one can look at thrown exceptions in the tested classes/methods [30]. However, one major problem here is that, often, thrown exceptions are not a symptom of a bug but rather are a violation of an unspecified precondition (e.g., inserting a null input when the target function is not supposed to work on null inputs).

Even if no automated oracle is available, generated tests are still useful for *regression testing*. For example, if a tested function foo takes as input an integer value and then returns an integer as a result of the computation, then an automatically generated test could capture the current behavior of the function in an assertion. For example:

```
int x = 5;
int res = foo(x);
assertEquals(9, res);
```

Now, a test generation tool could choose to create a test with input value $x = 5$, but it would not be able to tell whether the expected output should really be 9 (which is the actual value returned when calling $foo(x)$). A developer could look at the generated test and then confirm whether 9 is the expected output. However, even if the developer does not check it, the test could be added to the current set of test cases and then run at each new code change as part of a Continuous Integration process (e.g., Jenkins[22]). If a modification to the source code leads foo to return a value other than 9 when called with input 5, then that test case will fail. At this point, the developer would need to check whether the recently introduced change breaks the function (i.e., it is a bug) or if the semantics of that function has changed.

In the case of test cases for RESTful APIs, the generated tests can be used for regression testing as well. This is also particularly useful for security: for example, an HTTP invocation in which the returned status is 403 (unauthorized) can detect regression faults in which the authorization checks are wrongly relaxed. Furthermore, the status codes can be used as automated oracles. A 4xx code does not mean a bug in the RESTful web service, but a 5xx code can. If the environment (e.g., databases) of the tested web service is working correctly, then a 5xx status code would often mean a bug in the service. A typical example is thrown exceptions: the application server will not

[22] https://jenkins.io.

```
@GET @Timed
@Path("{id}/file")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
@UnitOfWork
@ApiOperation(value = "Download media file. Can resize images (but images will never be enlarged).")
public Response downloadFile(
    @PathParam("id") long id,
    @ApiParam(value = "" +
      "The maximum width/height of returned images. " +
      "The specified value will be rounded up to the " +
      "next 'power of 2', e.g. 256, 512, 1024 and so on.")
    @QueryParam("size") int size) {
    MediaFile mediaFile = dao.read(id);
    try {
        URI sourceURI = new URI(mediaFile.getUri());
        ...
    } catch (IOException e) {
        ...
    }
    ...
```

Fig. 3. An example (in Java, using DropWizard) of endpoint definition to handle a GET request, where requesting a missing resource, instead of resulting in a 404 code, leads to a 500 code due to a null pointer exception.

crash if an exception is thrown in the business logic of a RESTful endpoint. The exception would be caught, and a 5xx code (e.g., 500) would be returned. Note that if the user sends invalid inputs, the user should get back a 4xx code, not a 5xx one. Not doing input validation and letting the endpoint throw an exception would generate two main problems:

- The user would not know that it is the user's fault, and thus think it is a bug in the web service. Inside an organization, the developer might end up wasting time in filing a bug report. Furthermore, the 5xx code would not give the developer any hint as to how to fix the way in which to call the RESTful API.
- The RESTful endpoint might do a sequence of operations on external resources (e.g., databases and other web services) that might require being atomic. If an exception is thrown due to a bug after some, but not all, of those operations are completed, the environment might be left in an inconsistent state, making the entire system work incorrectly.

Figure 3 shows a simple example of an endpoint definition that contains bugs. This code is from one of the projects used in the empirical study. In that case, a resource (a media file) is referenced by id in the endpoint path (i.e., *@Path("{id}/file")*). The id is used to load the resource from a database (i.e., *dao.read(id)*), but there is no check to see whether it exists (e.g., if different from *null*). Therefore, when a test is created with an invalid id, the statement *mediaFile.getUri()* results in a null pointer exception. The exception is propagated to the application server (Jetty, in this case), which will create an HTTP response with status 500. The expected correct result here should had been a 404 (not found) code.

Because the fitness function rewards the covering of new HTTP status codes for each SUT endpoint, the search stores the test cases returning 5xx into the archive. However, the question remains of what to do with this kind of test. When we generate the final test suite to give as output to the user (e.g., in JUnit format), we could make those test cases fail (e.g., asserting that the returned status code should never be 5xx). But, because a 5xx is not always a symptom of a bug, we prefer to simply capture the behavior of the SUT, asserting that the returned status code is indeed 5xx (we will see examples of this later in the article when we show some of the generated tests on our case study, in Figure 7). These tests could point to faults in the SUT, though, and thus could be more valuable than the other tests. How to best convey this information to the user (e.g., warning comment messages in the generated tests, prioritization of their order in the generated JUnit files, or special test names) will be a matter of future research.

Checking for 5xx returned codes is the easiest way to define a (partial) oracle for RESTful APIs. It will not catch all types of bugs, but it is a reasonable start. More research will be needed to check whether other oracles could be added as well, such as some of the metamorphic relations defined in [47].

4.5 Tool Implementation

We have implemented a tool prototype in Kotlin to experiment with the novel approach discussed in this article. The tool is called EvoMaster, released under the LGPL open-source license.

For handling the SUT (start/stop it, and code instrumentation), we have developed a library for Java that, in theory, should work for any JVM language (Java, Kotlin, Groovy, Scala, etc.). However, we have tried it only on Java and Kotlin systems. Our tool prototype can output test cases in different formats, such as JUnit 4 and 5, in both Java and Kotlin. The test suites will be fully self-contained, that is, they will also deal with the starting/stopping of the SUT. The test cases are configured in such a way that the SUT is started on an ephemeral TCP port, which is an essential requirement for when tests are run in parallel (i.e., to avoid a SUT trying to open a TCP port that is already bound). The generated tests can be directly called from an IDE (e.g., IntelliJ or Eclipse), and can be added as part of a Maven or Gradle build.

In the generated tests, to make the HTTP calls to the SUT, we use the highly popular Rest-Assured[23] library. Assertions are currently generated only for the returned HTTP status codes. To improve the regression testing ability of the generated tests, it will be important to also have assertions on the headers and the payload of the HTTP responses. However, the presence/absence of such assertions would have no impact on the type of experiments run in this article.

4.6 Manual Preparation

In contrast to tools for unit testing such as EvoSuite, which are 100% fully automated (a user just needs to select for which classes tests should be generated), our tool prototype for system/integration testing of RESTful APIs requires some manual configuration.

The developers of the RESTful APIs need to import our library (which is published on Maven Central[24]) and then create a class that extends the *EmbeddedSutController* class in the library. The developers will be responsible for defining how the SUT should be started, where the Swagger schema can be found, which packages should be instrumented, and so forth. This will vary based on how the RESTful API is implemented, for example, if with Spring[25], DropWizard[26], Play[27], Spark[28] or JEE.

Figure 4 shows an example of a class that we had to write for one of the SUTs in our empirical study. That SUT uses Spring. The class is quite small and needs to be written only once. It does not need to be updated when there are changes internally in the API. The code in the superclass *EmbeddedSutController* will be responsible for doing the automatic bytecode instrumentation of the SUT, and it will also start a RESTful service to enable our testing tool to remotely call the methods of the class.

[23]https://github.com/rest-assured/rest-assured.
[24]http://repo1.maven.org/maven2/org/evomaster/.
[25]https://github.com/spring-projects/spring-framework.
[26]https://github.com/dropwizard/dropwizard.
[27]https://github.com/playframework/playframework.
[28]https://github.com/perwendel/spark.

```java
public class ControllerForFS extends EmbeddedSutController {
    public static void main(String[] args) {
        int port = 40100;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        }
        ControllerForFS controller = new ControllerForFS(port);
        InstrumentedSutStarter starter = new InstrumentedSutStarter(controller);
        starter.start();
    }
    private ConfigurableApplicationContext ctx;
    private Connection connection;
    public ControllerForFS(int port) {
        setControllerPort(port);
    }
    @Override public String startSut() {
        ctx = SpringApplication.run(Application.class, new String[]{"--server.port=0",});
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
        JdbcTemplate jdbc = ctx.getBean(JdbcTemplate.class);
        try {
            connection = jdbc.getDataSource().getConnection();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return "http://localhost:" + getSutPort();
    }
    protected int getSutPort() {
        return (Integer) ((Map) ctx.getEnvironment()
                                .getPropertySources().get("server.ports").getSource())
                            .get("local.server.port");
    }
    @Override public boolean isSutRunning() {
        return ctx != null && ctx.isRunning();
    }
    @Override public void stopSut() {
        ctx.stop();
    }
    @Override public String getPackagePrefixesToCover() {
        return "org.javiermf.features.";
    }
    @Override public void resetStateOfSUT() {
        DbCleaner.clearDatabase_H2(connection);
    }
    @Override
    public List<AuthenticationDto> getInfoForAuthentication() {
        return null;
    }
    @Override public String getUrlOfSwaggerJSON() {
        return "http://localhost:" + getSutPort() + "/swagger.json";
    }
}
```

Fig. 4. Example of EvoMaster driver class written for the *features-service* SUT.

However, in addition to starting/stopping the SUT and providing other information (e.g., location of the Swagger file), there are two other tasks that the developers need to perform:

- RESTful APIs are supposed to be stateless (so that they can easily scale horizontally), but they can have side effects on external actors, such as a database. In such cases, before each test execution, we need to reset the state of the SUT environment. This needs to be implemented inside the *resetStateOfSUT()* method, which will be called in the generated test suite before running any test. In the case of the class in Figure 4, the only state is in the

database; thus, we use a support function to clean its content (but without changing the existing schema).
- If a RESTful API requires some sort of authentication, the information has to be provided by the developers in the *getInfoForAuthentication()* method. For example, even if a testing tool would have full access to the database storing the passwords for each user, it would not be possible to reverse engineer those passwords from the stored hash values. Given a set of valid credentials, the testing tool will use them as any other variable in the test cases, for example, to do HTTP calls with and without authentication.

5 SMART SAMPLING

5.1 Problem Definition

In our context, a test case is a sequence of one or more HTTP calls. To test a particular endpoint, there might be the need for previous calls to set the state of the application (e.g., a database) in a specific configuration. Conceptually, this is similar to unit testing of object-oriented software, where one might need a sequence of function calls to set the internal state of an object instance.

Search algorithms need to be able to sample individuals randomly from the search space. Random sampling is required to initialize the first population of a Genetic Algorithm. Other algorithms, such as MIO, do sample random tests at each step, given a certain probability (this is done to avoid premature convergence of the algorithm). When a test case is sampled at random, it will have a random number of HTTP calls between 1 and n (e.g., $n = 10$). Each HTTP call will have its parameters (e.g., headers and payloads) initialized at random.

For a given API, what endpoints are available, and their properties, is usually defined with schema files, where Swagger[29] is arguably the most popular kind. The sampling of new tests will be done according to the constraints of such schemas. Using complete random strings to sample HTTP calls would not be viable because such random strings would have an extremely low probability of representing valid HTTP messages.

Once a test case has been sampled, it will be mutated throughout the search. Mutation operators can modify the parameters of an HTTP call, as well as change the structure of the test, by adding or removing HTTP calls. This is similar to how test sequences are evolved in unit testing tools such as EvoSuite [27]. However, there is a major difference, which is related to how *related state* can be identified. Consider the following example:

```
Foo foo = new Foo();
foo.something();
foo.somethingElse();
Bar bar = new Bar();
int x = 42;
bar.targetMethod(x);
```

If for a moment we ignore the possibility of static state, to fully test the method `targetMethod` we might need to not only modify its input `x` but also its internal state. Its internal state would be defined in the variable `bar`. A testing tool could add new calls just on `bar` and ignore (or even remove) `foo`. Even if one does not know the semantics of the methods in the classes `Foo` and `Bar`, we know that calling methods on `bar` *might* affect the state needed to fully test `targetMethod`. Unfortunately, this is not the case for RESTful APIs. Recall the example discussed in Section 2.2 and consider this test representation of three HTTP calls in sequence (for simplicity, let us ignore the full syntax of how HTTP calls are actually made in a test with their payloads):

[29]https://swagger.io/.

```
POST /products
POST /products
GET  /products/100
```

As it is now, the last GET call would likely result in a 404 "not found" message. To fully cover the code of the handler of that endpoint, we might need actual data for the product resource represented by the id 100. However, there might be no operation (e.g., POST or PUT) to create a resource directly on its endpoint "/products/100." A randomly sampled test might have calls to "POST /products," which likely would generate product resources. However, there would be no direct gradient to generate a resource with id 100. Even worse, it might not even be possible to generate a resource with a specific id because the id could be generated dynamically by the SUT (e.g., if the same id is used to store the record of the product in an SQL database). Even if by chance we would get a "POST /products" that creates a product with id 100, running the test twice could result in a different id, with the test behaving differently (thus likely becoming *flaky*). The standard search-based heuristics, such as the *branch distance*, might not be enough to guide the search here, as the state of the system would likely be handled in a different process (e.g., SQL and NoSQL databases).

5.2 Proposed Solution

To overcome the problem of test cases with HTTP calls that are unlikely to interact on the same state, we propose the following technique. Every time a new test case needs to be randomly sampled, with a given probability P we use a predefined *template* to define the structure of the test.

A random endpoint in the SUT (based on its API schema) is chosen, and one of the available HTTP methods on it is chosen at random. A test case will be created where the action on this endpoint is the main target we want to test, which will be the last HTTP call in the test. Other HTTP calls will be added before it to try to bring the state of the SUT into the right configuration. What kind of HTTP calls will be added depends on the HTTP verb of the target and its URI. In the following, we will discuss these templates. The full details can be found in the source code of the `RestSampler` class in EvoMaster.

5.2.1 GET Template. There are two main cases to handle for GET: whether the last path element is a variable or a constant. Consider the example:

```
GET  /products
```

The endpoint might represent a single named resource or a collection (whether the name is in plural form might provide a hint about it). To properly test a GET on a collection (there might be different query parameters representing filters), we might need several elements in it. Thus, we add a random number k of POST calls (if POST is possible on the resource). Therefore, in this template, we will have k (e.g., $k = 3$) POSTs followed by a single GET, for example:

```
POST /products
POST /products
POST /products
GET  /products
```

Note that for simplicity in these examples we are not showing headers, query parameters, or payloads. If there is no POST/PUT operation on the resource, then we will have $k = 0$.

Now, for the other case of parameters in the URIs, consider the example

```
GET  /products/{id}
```

where $\{id\}$ is a placeholder, that is, the controller handling that endpoint will accept any request for a URI in the form of "/products/$*$," where the last path element will be stored in a variable called id (i.e., a so-called path parameter). In this case, we check in the schema whether there is any creation operation (i.e., POST or PUT) on the resource. If none is available, we look at the closest *ancestor* resources in the URI, which, in this case, is "/products," and see whether there is a creation operation on it. If so, we add it, and we end up with a test with two HTTP calls, for example:

```
POST /products
GET  /products/{id}
```

How to link the resource created by the POST with the id variable in the GET will be explained in Section 5.2.7.

5.2.2 POST Template. As POST is used to create resources, it does not need special handling. Thus, we will have a test with a single POST call.

5.2.3 PUT Template. PUT is an idempotent method that fully replaces the state of a resource. However, if such resource does not exist, then the PUT could either create it or return an 4xx error. For example, this is the case when a user cannot choose the id for a resource, such as when the id in the URI is the same used in an SQL database (where ids might be generated by the database according to some specific rules). Here, we consider these two different cases, that is, given a certain probability we either sample a test with a single PUT or a test in which we first try to create a resource with a POST (if any is available on it or on an ancestor), for example:

```
POST /products
PUT  /products/{id}
```

5.2.4 PATCH Template. A PATCH does a partial update on a resource. To test it, we need to create the resource first. Note that this is different from PUT, where the PUT itself could create the resource if missing. Thus, we first need to do a POST/PUT on the endpoint (or one of its ancestors), followed by the PATCH. To test whether the partial updates are working correctly, it might be necessary to have more than one PATCH in sequence on the same resource. Thus, in this template, we randomly choose between having one or two PATCHes in the test, for example:

```
POST   /products
PATCH  /products/{id}
PATCH  /products/{id}
```

5.2.5 DELETE Template. To properly test a DELETE, we need a case in which the resource exists. Therefore, as in the other templates, we need a POST/PUT to first create the resource, for example,

```
POST    /products
DELETE  /products/{id}
```

5.2.6 Intermediate Resources. In the templates discussed in the previous sections, there was the need to create a resource before being able to test different actions on it (e.g., PATCH and DELETE). However, a resource might depend on other resources. Consider when we need to generate a test for "GET /products/{id}/price." By applying the GET Template discussed in Section 5.2.1, we would have a test with a single GET directly on that resource, as its last path element (i.e., "price") is a constant, and there is no POST on the endpoint. However, the GET call would likely result in a 404 *not found* error.

To avoid this kind of problem, in all the templates discussed so far, we add the pre-step of recursively generating all needed intermediate resources. This can be identified by checking whether there is any path parameter in the URI of the endpoint. In this example, our technique would detect that we might need to create the "/products/{id}" resource before we can operate on the endpoint "/products/{id}/price." Thus, it would add a POST/PUT method to create it (either directly or on one of its ancestors). In this case, we would end up with the following test:

```
POST /products
GET  /products/{id}/price
```

A more complex example is when we want to test an endpoint such as "GET /products/{pid}/subitems/{sid}." The GET Template would generate the following structure:

```
POST /products/{pid}/subitems
GET  /products/{pid}/subitems/{sid}
```

However, the POST itself would depend on an intermediate resource that might not exist yet. By applying the pre-step discussed in this section, we would obtain the following test structure:

```
POST /products
POST /products/{pid}/subitems
GET  /products/{pid}/subitems/{sid}
```

In other words, we first need to create a product, then we create a subitem in that product, and, finally, we retrieve the subitem.

5.2.7 *Chained Locations.*

Recall the previous example in which we have the following test:

```
POST /products
POST /products/{pid}/subitems
GET  /products/{pid}/subitems/{sid}
```

Somehow, we need to guarantee that the `pid` used in the second POST is the same as the resource created with the first POST and that `pid` and `sid` in the GET are the same as the two resources created with the POSTs. If the API allows creation of resources directly, then we would have a test like this:

```
POST /products/{pid}
POST /products/{pid}/subitems/{sid}
GET  /products/{pid}/subitems/{sid}
```

Here, we would choose the ids (e.g., randomly), and then make sure in the test-generation tool that those ids are the same in each HTTP call (e.g., not mutated during the search).

On the other hand, when a new resource is created with a POST on a collection (e.g., "POST/products"), we need to know where to find the newly generated resource. The recommended approach in HTTP (RFC 7231) is to put the information in the *Location* header of the response. This could be either a relative URI to the newly created resource or a full URL. For example, if "POST/products" creates a resource with id 100, it could specify in the HTTP response the header "location:/products/100." Thus, in the test, instead of hardcoding the values for the variables `pid` and `sid`, we dynamically extract them from the location headers of the POST calls that generated the resources. The path parameters of the second POST will depend on the response of the first POST, whereas the path parameters of the GET will depend on the response of the second POST.

However, there might be cases (e.g., in our case study) of APIs in which the location header is not used to specify where the newly created resources can be located. A complementary approach is to return the newly created resource as payload of the POST response itself. If the payload of the response is structured data (e.g., JSON or XML), we check whether it has any field that seems related to any of the path parameters in the endpoint (e.g., a field called `id`). If that is the case, we heuristically try to extract the value from the POST response and use it as a path parameter in the following HTTP calls. As for any heuristics, this approach is not guaranteed to work in all cases, but it is better than doing nothing if the location header is missing.

5.2.8 *Search Operators.*

During the search, test cases will be mutated. As previously discussed, a mutation can either change the properties of an HTTP call (e.g., the payload) or change the structure of the tests by adding/removing those HTTP calls. However, if a test was sampled with our smart sampling strategy (and not at random), we forbid mutations that change the structure. The reason is that removing calls would likely break the chain of dependent locations (see Section 5.2.7) or add calls on unrelated endpoints or states, which would make the test more expensive to run. The only exception is the parameter k in the GET Template (see Section 5.2.1), that is, we allow adding/removing POST calls on the collection.

Another important consideration is that if two calls have a chained location (i.e., one depends on the other, see Section 5.2.7), then there would be no point in mutating their path parameters, as they would not be used anyway (as we dynamically assign them). Therefore, on a test sampled with our smart strategy, we allow mutations that can change only the payloads, HTTP headers, and query parameters.

Note that regardless of how tests are sampled (i.e., with smart strategy or at random), all of them will be evaluated with exactly the same fitness function. Thus, if tests sampled with our strategy lead to worse fitness, they will die out during the search compared to the tests originally sampled at random.

5.2.9 *Premature Stopping.*

The use of our templates is warranted for when we need to create resources before doing actions on them. However, doing a call to create such a resource might fail. This could happen owing to bugs in the SUT or to invalid input parameters (e.g., when we make a POST, we still need to create a valid payload). A failure would be identified by either a 4xx (user error) or a 5xx (server error) status code in the HTTP responses. If that is the case, then we stop the execution of the test, as there would be no need to execute the subsequent HTTP calls that depend on resources that we failed to create.

5.3 Tool Support for Smart Sampling

To handle the case of chained locations (see Section 5.2.7), the generated test files (e.g., in JUnit for Java software) need to be able to dynamically extract the location headers. To this end, we have created a small library that is linked in the generated tests.

Figure 5 shows an example of a test generated by EvoMaster with our novel technique when applied on the *scout-api* SUT (one of the SUTs used in our case study). Such a test can be represented by the following two HTTP calls using the DELETE Template:

```
POST    /api/v1/categories
DELETE  /api/v1/categories/{id}
```

As can be seen in Figure 5, the needed `id` is extracted from the payload of the POST call. The support method `resolveLocation()` is used to reconstruct a whole URL, as there are many edge cases that need to be handled.

Figure 6 shows the results of running EvoMaster on an artificial example, where there is the need of first creating three resources whose locations are extracted from the HTTP headers. Note the use of `assertTrue(isValidURIorEmpty())`: a location header might be missing (that is not a bug), but, if it is present, it must be a valid URI. If the location header is missing, `resolveLocation()` still tries to create a valid URL.

```
@Test
public void test9() throws Exception {

  String location_categories = "";

  String id_0 = given().accept("*/*")
      .header("Authorization", "ApiKey moderator") // moderator
      .contentType("application/json")
      .body("{\"group\":\"9 oRyB\", \"name\":\"6 oh9\", \"activities_count\":719063}")
      .post(baseUrlOfSut + "/api/v1/categories")
      .then()
      .statusCode(200)
      .extract().body().path("id").toString();

  location_categories = "/api/v1/categories/" + id_0;

  given().accept("*/*")
     .header("Authorization", "ApiKey moderator") // moderator
     .delete(resolveLocation(location_categories,
          baseUrlOfSut + "/api/v1/categories/-1679225313"))
     .then()
     .statusCode(204);
}
```

Fig. 5. An example of test (in Java, using JUnit and RestAssured) generated by EvoMaster where the location of the newly created resource is extracted from the payload of the POST call that created it.

```
@Test
public void test4() throws Exception {

  String location_x = "";
  String location_y = "";
  String location_z = "";

  location_x = given().accept("*/*")
      .post(baseUrlOfSut + "/api/chl/x")
      .then()
      .statusCode(201)
      .extract().header("location");

  assertTrue(isValidURIorEmpty(location_x));

  location_y = given().accept("*/*")
      .post(resolveLocation(location_x, baseUrlOfSut +
         "/api/chl/x/-2122041469/y"))
      .then()
      .statusCode(201)
      .extract().header("location");

  assertTrue(isValidURIorEmpty(location_y));

  location_z = given().accept("*/*")
      .post(resolveLocation(location_y, baseUrlOfSut +
         "/api/chl/x/-2122041469/y/974/z"))
      .then()
      .statusCode(201)
      .extract().header("location");

  assertTrue(isValidURIorEmpty(location_z));

  given().accept("*/*")
      .get(resolveLocation(location_z, baseUrlOfSut +
          "/api/chl/x/-2122041469/y/974/z/721279551/value"))
      .then()
      .statusCode(200);
}
```

Fig. 6. An example of test (in Java, using JUnit and RestAssured) generated by EvoMaster where there is a chain of generated resources whose locations are dynamically extracted from the *Location* header of the HTTP responses.

6 EMPIRICAL STUDY

In this article, we have carried out an empirical study aimed at answering the following research questions.

RQ1: Can our technique automatically find real faults in existing RESTful web services?
RQ2: How do our automatically generated tests compare, in terms of code coverage, with the already existing manually written tests?
RQ3: How much improvement can our novel sampling technique achieve?
RQ4: What are the main factors that impede the achievement of better results?

6.1 Artifact Selection

To achieve sound, reliable conclusions from an empirical study, ideally we would need a large set of artifacts for experimentation, selected in an unbiased way [28]. However, for experiments on system testing of web services, this is not viable. First, system level testing requires some manual configuration. Second, a major issue is that RESTful web services, although very popular among enterprises in industry, are less common among open-source projects. Finding the right projects that do not require complex installations (e.g., special databases and connections to third-party tools) to run is not a trivial task.

We used Google BigQuery[30] to analyze the content of the Java projects hosted on GitHub[31], which is the main repository for open-source projects. We searched for Java projects using Swagger. We excluded the projects that were too small and trivial. We downloaded and tried to compile and run several of these projects with different degrees of success. In the end, for the empirical study in this article, we manually chose four different RESTful web services for which we could compile and run their test cases with no problems. These services are called: *catwatch*[32], *features-service*[33], *proxyprint*[34] and *scout-api*[35]. We also used a small artificial RESTful service written in Kotlin called *rest-news*, which we developed previously for educational purposes as part of a course on enterprise development.

Data about these five RESTful web services is summarized in Table 1. To enable the replicability of the experiments in this article, we gather together all of these services in a single GitHub repository[36], including all the needed configuration files to run EvoMaster on these web services.

Those five RESTful web services contain up to 7,500 lines of codes (tests excluded). This is a typical size for a RESTful API, especially in a microservice architecture [41]. The reason is that, to avoid the issues of monolithic applications, these services usually split when they grow too large to make them manageable by a single small team. This, however, also implies that enterprise applications can end up being composed of hundreds of different services. Currently, EvoMaster focuses on the testing of RESTful web services in isolation and not their orchestration in a whole enterprise system.

For each SUT, we had to *manually* write a driver class. For example, recall Figure 4, where the driver used for the *features-service* SUT is shown. The class uses a client library from EvoMaster (which is published on Maven Central[37]). This class extends the `EmbeddedSutController` class

[30] https://cloud.google.com/bigquery.
[31] https://github.com.
[32] https://github.com/zalando-incubator/catwatch.
[33] https://github.com/JavierMF/features-service.
[34] https://github.com/ProxyPrint/proxyprint-kitchen.
[35] https://github.com/mikaelsvensson/scout-api.
[36] https://github.com/EMResearch/EMB.
[37] http://repo1.maven.org/maven2/org/evomaster/.

Table 1. Information about the Five RESTful Web Services Used in the Empirical Study

Name	# Classes	LOCs	Endpoints
catwatch	69	5442	23
features-service	23	1247	18
proxyprint	68	7534	74
rest-news	10	718	7
scout-api	75	7479	49
Total	245	22420	171

We report their number of Java/Kotlin class files and lines of code. We also specify the number of endpoints, i.e., the number of exposed resources and HTTP methods applicable on them. All of these SUTs interact with an SQL database.

and starts the driver functionality with `InstrumentedSutStarter`. The latter will start a RESTful API used by the EvoMaster main process (which, by default, tries to connect to port 40100) to control the SUT (e.g., to send commands to start/stop/reset it). Note the specific configurations for *features-service*, for example, which Java packages should be instrumented (`getPackagePrefixesToCover()`) and how to reset the state of the SUT after/before each test run (`resetStateOfSUT()`).

6.2 Experiment Settings

On each of the five SUTs, we ran EvoMaster 100 times, with two different search budgets (10k HTTP calls, and 100k HTTP calls) and with six different values for the probability P of using our novel smart sampling, in particular, $P \in \{0,\ 0.2,\ 0.4,\ 0.6,\ 0.8,\ 1\}$. In total, we had $5 \times 100 \times 2 \times 6 =$ 6,000 independent searches, for a total of $5 \times 100 \times (10{,}000 + 100{,}000) \times 6 = 330m$ HTTP calls.

We did not use the number of fitness evaluations as a stopping criterion because each test can have a different number of HTTP calls. Considering that each HTTP call requires sending data over a TCP connection (plus the SUT that could write/read data from a database), their cost is not negligible (even if still in the order of milliseconds when both EvoMaster and the SUT run on the same machine). As the cost of running a system test is significantly larger than the overhead of the search algorithm code in EvoMaster, we did not use time as a stopping criterion. This will help future comparisons and replications of these experiments, especially when run on different hardware. However, notice that, by default, EvoMaster uses time as a stopping criterion, as that is more user friendly.

For each run of the algorithm, not only do we need to run the EvoMaster core process but also the EvoMaster driver and the SUT itself (thus, three different processes). Owing to the large number of experiments, a cluster of computers was necessary. Each experiment was configured to have three CPU cores at its disposal. In total, these experiments required 1,348 days of computational resources.

For these experiments, all SUTs were configured to use an embedded SQL database, such as H2[38] and Derby[39].

[38] www.h2database.com.
[39] db.apache.org/derby.

Table 2. Results of the Experiments, Based on 100 Runs Per SUT

SUT	#Tests	# Codes	#5xx	
			Avg.	Max
catwatch	39.5	1.6	5.0	5
features-service	49.6	2.6	13.4	14
proxyprint	213.7	2.8	28.0	28
rest-news	24.7	1.9	0.0	0
scout-api	177.2	2.8	33.0	33

We considered EvoMaster with a 100,000 HTTP call budget, with $P = 0.6$ for the sampling probability. We report the average number of test cases in the final test suites, the average number of different HTTP status responses per endpoint, and the average and max number of distinct endpoints per service with at least one test leading to a 5xx status code response.

6.3 Experiment Results

Table 2 shows the results of the experiments on the five different RESTful web services when using EvoMaster with a 100,000 HTTP calls budget and with $P = 0.6$ for the sampling probability. Although during the search we evaluated 100,000 HTTP calls per run, the final test suites are much smaller, on average, between 24 (*rest-news*) and 214 (*proxyprint*) tests. This is because we keep only those tests that contribute to covering our defined testing targets (e.g., code statements and HTTP return statuses per endpoint). On average, each endpoint is called with inputs that lead to more than 1 different returned HTTP status code.

These tests, on average, can lead the SUTs to return 5xx status code responses in 80 distinct endpoints. On the one hand, an endpoint might fail in different ways owing to several bugs in it. On the other hand, two or more endpoints in the same web service might fail due to the same bug. Without an in depth analysis of each single test with a 5xx status code, it is not possible to state the exact number of faults found. However, a manual inspection of 20 test cases seems to point out that, in most cases, it is a 1-1 relationship between failing endpoints and actual faults. Therefore, in this article, we will consider the number of failing endpoints as an estimate of the number of detected faults.

A simple example of detected fault is the one that appears in Figure 3. A generated test revealing such a bug is shown in Figure 7. Note that for that particular endpoint, there are only three decisions to make: (1) whether or not to call it with a valid authentication header; (2) the numeric value of the "id" in the resource path; and (3) the numeric value of the "size" query parameter.

Not all faults have the same severity. It can be argued that faults leading to returning the wrong answers (or simply a 5xx) when the inputs are *valid* might be considered more severe than just returning the wrong error status code for *invalid* inputs (as in the case of Figure 3 and Figure 7). However, defining what is *valid* and *invalid* is not necessarily obvious. Swagger definitions provide the possibility to write some forms of *preconditions*, but complex constraints (e.g., involving relations among more than one input field) cannot be currently expressed. Such constraints might still exist, but they might be expressed in natural language in the description of the endpoint functionalities.

Without an in-depth manual analysis of all found faults (which is not in the scope of this article), it is not possible to state whether the found faults are critical or not. However, a preliminary analysis of some of these faults seems to point out that the majority of the found faults are owing to improper handling of *invalid* inputs. This is not unexpected. A developer might concentrate on

```
static EMController controller = new EMController();
static String baseUrlOfSut;

@BeforeClass
public static void initClass() {
    baseUrlOfSut = controller.startSut();
    assertNotNull(baseUrlOfSut);
}

@AfterClass
public static void tearDown() {
    controller.stopSut();
}

@Before
public void initTest() {
    controller.resetStateOfSUT();
}

@Test
public void test0() throws Exception {

    given().header("Authorization", "ApiKey user")
            .accept("*/*")
            .get(baseUrlOfSut + "/api/v1/media_files/-4203492812/file?size=-141220")
            .then()
            .statusCode(500);
    }
}
```

Fig. 7. Generated RestAssured test (Java, JUnit 4) for the endpoint shown in Figure 3. We also show the scaffolding code used to automatically start/stop/reset the SUT.

testing and fixing the basic functionalities of one's APIs in "happy day" scenarios, and not considering all of the different ways that they can be called with invalid inputs. Thus, it is reasonable to expect that finding this kind of fault would be more common when applying an automated tool on this kind of existing API. A precise classification of what kind of faults (and their proportion) can be found in RESTful APIs is an important direction for future search.

Besides improper handling of invalid inputs, bugs can also happen owing to invalid assumptions on the state of the application and its third-party dependencies. An example is the endpoint `/statistics/projects` in the *catwatch* SUT. There, the API makes the assumption that the database contains at least 10 projects. When a test is generated with an empty database, the endpoint returns the 500 HTTP status code due an uncaught `IndexOutOfBoundsException`.

RQ1: *Our novel technique automatically identified 80 different faulty endpoints in the analyzed web services.*

In addition to finding faults, the generated test suites can also be used for regression testing. To be useful in this context, it is best that these test suites have high code coverage. Otherwise, a regression in a non-executed statement would not fail any of the tests.

Table 3 shows the statement coverage results of the generated tests. Such results are compared against the ones of the already existing manually written tests. Code coverage was measured by running the tests directly from the IDE IntelliJ, using its code coverage tools. This also helped to check whether the generated tests worked properly. For each project, we chose one generated test suite with high coverage. We did not calculate the average over all generated test suites in the 100 repetitions, as running the tests in IntelliJ and their coverage analysis was done manually.

Table 3. Statement Coverage Results for the Generated Tests Compared to the Ones Obtained by the Already Existing Manually Written Tests

SUT	Coverage	Manual Coverage		
		Total	Unit/Int.	HTTP API
catwatch	32%	56%	80%	53%
features-service	64%	82%	16%	78%
proxyprint	43%	40%	40%	0%
rest-news	65%	58%	0%	58%
scout-api	38%	52%	11%	41%

Coverage was measured by running the tests inside IntelliJ IDEA. For the manual tests, we distinguish on whether they do HTTP calls to the API or are lower-level unit or integration tests.

Note that whether a test generation tool can achieve better results than existing manual tests obviously depends on the quality of the latter. Also, it is not uncommon to expect worse test cases in an open-source project compared to an industrial one.

Regarding the manual tests, in addition to their total coverage, we report values separately based on two non-overlapping groups: (1) if the tests do HTTP calls to the API (i.e., the same kind of tests that we generate with EvoMaster), or (2) if they are lower-level unit or integration tests. The results are very different among the SUTs. On the one hand, *proxyprint* has only unit tests, whereas the coverage in *features-service* is nearly given just by the tests with HTTP calls. For *scout-api*, unit/integration tests and HTTP call tests have no overlapping coverage (i.e., the total coverage is exactly the sum of these two groups). The case of *catwatch* is rather peculiar, as the total coverage decreases when all of the tests are run together in sequence. This might happen if there are dependencies among tests and the state of the application is not properly cleaned up after each test execution.

The results in Table 3 clearly show that, for the generated tests, the obtained code coverage is generally lower. There are definitively several challenges in automated-system test generation that need to be addressed before we can achieve higher coverage.

RQ2: *On average, the generated test suites obtained between 32% and 65% statement coverage. This is generally lower than the coverage of the existing test cases in those SUTs.*

Table 4 shows the results of our experiments when comparing the base configuration of EvoMaster with and without smart sampling. In particular, we measure the effectiveness of a configuration based on the number of *testing targets* it covers in the generated test suites. Currently, EvoMaster does target statements in the bytecode, branches, call method executions (in case they throw an exception), and HTTP return statuses for each endpoint. Looking only at statement/branch coverage with existing coverage tools would be misleading, as those would ignore tests that can crash the SUT but that do not increase coverage.

Table 5 is based on the data from Table 4. However, instead of showing the raw numbers of covered targets, it shows the relative improvement of smart sampling compared to the base EvoMaster configuration. In other words, if smart sampling achieves X targets, and base achieves Y targets, then the relative improvement (if any) is calculated as $\frac{X-Y}{Y}$.

As can be seen in Table 5, smart sampling improves performance between 12.5% and 15.1%, on average, with peaks of over 70% in some cases (e.g., in the *features-service* SUT). Large improvements are obtained for two of the SUTs. On the others, the improvements, if any, are modest.

Table 4. Average Numbers of Covered Test Targets

SUT	Budget	Base	Smart Sampling Probability				
		0	0.2	0.4	0.6	0.8	1
catwatch	10k	964.2	964.5	964.9	965.8	965.7	**968.8**
	100k	971.5	971.5	**972.0**	971.8	971.7	971.7
features-service	10k	282.5	486.1	**506.1**	500.5	501.6	496.5
	100k	446.9	**551.5**	542.8	536.2	533.0	520.1
proxyprint	10k	1308.2	1357.3	**1376.1**	1342.4	1343.2	1343.3
	100k	1414.9	1416.9	1407.6	**1418.5**	1400.2	1403.1
rest-news	10k	251.1	246.5	**251.8**	248.0	250.4	239.3
	100k	262.0	259.7	264.6	261.9	**265.5**	254.4
scout-api	10k	1328.1	1398.4	1464.5	1506.2	1555.3	**1587.2**
	100k	1414.6	1745.3	1837.7	1869.5	**1893.4**	1891.8

Values in bold are the highest in their row.

Table 5. Relative Improvements (in Terms of Average Numbers of Covered Targets) of Smart Sampling Technique Compared with Default Version of EvoMaster

SUT	Budget	Smart Sampling Probability				
		0.2	0.4	0.6	0.8	1
catwatch	10k	+0.0%	+0.1%	+0.2%	+0.2%	**+0.5%**
	100k	−0.0%	**+0.0%**	+0.0%	+0.0%	+0.0%
features-service	10k	+72.1%	**+79.1%**	+77.2%	+77.5%	+75.7%
	100k	**+23.4%**	+21.5%	+20.0%	+19.3%	+16.4%
proxyprint	10k	+3.8%	**+5.2%**	+2.6%	+2.7%	+2.7%
	100k	+0.1%	−0.5%	**+0.3%**	−1.0%	−0.8%
rest-news	10k	−1.8%	**+0.3%**	−1.2%	−0.3%	−4.7%
	100k	−0.9%	+1.0%	−0.1%	**+1.3%**	−2.9%
scout-api	10k	+5.3%	+10.3%	+13.4%	+17.1%	**+19.5%**
	100k	+23.4%	+29.9%	+32.2%	**+33.8%**	+33.7%
Average		+12.5%	+14.7%	+14.4%	**+15.1%**	+14.0%

Values in bold are the highest in their row.

Whether our technique is successful depends on the schema of the RESTful APIs. If an API does not have any hierarchical resource (as is the case in some of the SUTs), then no improvement can be obtained.

In our experiments, we considered six different values for the smart sampling probability P, from 0 (no smart sampling) to 1. However, the results shown in Table 5 do not clearly point to which P is best, that is, a low value or a high value.

Given the current data, it could be advisable to choose a middle value, such as $P = 0.6$. With that P value, Table 6 shows the comparisons with the base configuration (i.e., $P = 0$). To check whether indeed our experiments show that smart testing provides better results, Table 6 also reports standardized effect sizes and P values of statistical tests [8]. IWe used the Vargha-Delaney $\hat{A}_{12}$ effect size and the non-parametric Mann-Whitney-Wilcoxon U-test. Results in Table 6 clearly show, with high statistical confidence, that smart testing with $P = 0.6$ significantly improves performance compared with the base $P = 0$ configuration.

The longer a search algorithm is run, the better the results will be. In theory, given infinite time, even random testing could achieve maximum coverage. On the other hand, for small search

Table 6. Detailed Comparison of Smart Sampling (using $P = 0.6$) with Default Version of EvoMaster, where the Standardized Effect Sizes $\hat{A}_{12}$ and P-values of Mann-Whitney-Wilcoxon U-tests are Reported as Well

SUT	Budget	Base	$P = 0.6$	$\hat{A}_{12}$	p-value
catwatch	10k	964.2	965.8	0.66	<0.001
	100k	971.5	971.8	0.51	0.830
features-service	10k	282.5	500.5	1.00	<0.001
	100k	446.9	536.2	0.75	<0.001
proxyprint	10k	1308.2	1342.4	0.71	<0.001
	100k	1414.9	1418.5	0.46	0.376
rest-news	10k	251.1	248.0	0.41	0.041
	100k	262.0	261.9	0.43	0.080
scout-api	10k	1328.1	1506.2	0.91	<0.001
	100k	1414.6	1869.5	1.00	<0.001

budgets, evolutionary algorithms would not have time to evolve, and thus could behave similarly to or even worse than a random search. Therefore, when analyzing the performance of a search algorithm, it is very important to specify for how long it is run.

In this article, we considered two different search budgets: maximum 10,000 HTTP calls, and maximum 100,000. In other words, the second settings use 10 times the resources of the first settings. The choice of search budget should be based on how practitioners use the search-based tools in their daily jobs. Do they expect results in a few seconds? Are they willing to wait a few minutes if that is going to give much better results? What about running a test-generation tool in the background while an engineer is attending a meeting? What about letting the test generation run after working hours or on a remote continuous integration server [18]? As we currently do not have such information for system-level testing tools, the choice of 10,000 and 100,000 was rather arbitrary, based on our experience. Another issue is that search budgets are also strongly related to the performance of the used hardware, which increases year after year.

As we can see in Table 4 and Table 5, smart sampling improves performance with both search budgets but its effect varies. For example, in *features-service*, with a budget of only 10,000, improvement is very large, around +70%. However, when a larger budget is used, although there is still improvement, the improvement is much smaller, in the order of just 16%–23%. This can be explained by the fact that, although smart sampling gives an initial boost to performance, with enough search budget the current fitness function is already good enough to give gradient to cover most of those newly covered targets. On the other hand, for *scout-api*, it is exactly the other way around. With a 10,000 search budget, improvements are *at most* 19.5%, whereas for 100,000 they are *at least* 23.4%, with the peak at 33.8%. A possible explanation is that the smart sampling allows covering of more endpoints; however, you still need the right input parameters (i.e., HTTP headers, body payloads, and URL query parameters). With only 10,000 as a search budget, it seems that there is not enough time to evolve the right input parameters; thus, the boost of smart sampling cannot fully express itself.

Another interesting phenomenon that we can observe in Table 4 is that, for both *features-service* and *scout-api*, smart sampling configuration at a 10,000 search budget gives better (506.1 vs. 446.9 for *features-service* and 1,587.2 vs. 1,414.6 for *scout-api*) performance than not using smart sampling with a 10 times larger search budget (i.e., 100,000).

Table 7. Average Numbers of Faults Detected Based on 5xx Status Responses

SUT	Budget	Base	Smart Sampling Probability				
		0	0.2	0.4	0.6	0.8	1
catwatch	10k	**5.0**	**5.0**	**5.0**	**5.0**	**5.0**	**5.0**
	100k	**5.0**	**5.0**	**5.0**	**5.0**	**5.0**	**5.0**
features-service	10k	**13.4**	13.4	13.3	13.2	13.3	13.2
	100k	**14.0**	13.5	13.4	13.4	13.4	13.4
proxyprint	10k	26.8	**27.4**	27.3	27.2	27.2	27.0
	100k	28.0	**28.0**	**28.0**	28.0	27.9	27.5
rest-news	10k	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	100k	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
scout-api	10k	**33.0**	33.0	**33.0**	33.0	32.9	32.7
	100k	**33.0**	**33.0**	**33.0**	**33.0**	**33.0**	33.0

Values in bold are the highest in their row (before trunking the values to 1 digit precision).

Table 7 shows the same kind of analysis done in Table 4, but considering those faults due to 5xx return statuses instead of all testing targets. In the table, it is clear that smart sampling does not impact much on the fault detection. This is not totally unexpected. As previously discussed in **RQ1**, most found faults seem to involve the handling of invalid inputs. The state of the application would not matter much in such cases. Furthermore, achieved coverage is still not so high (Table 3), and better fault detection could be achieved if we could get even higher code coverage. After all, if the code of a fault is never executed, the fault cannot be found regardless of the used automated oracle.

However, it is also important to consider the fact that you cannot find faults if there is no fault. We used actual systems from open-source projects on GitHub, with unknown numbers and types of faults in them. In the future, it would be important to study how results would vary in controlled empirical studies in which artificial faults are injected with, for example, a Mutation Testing tool.

RQ3: *When a RESTful API has hierarchical resources, our novel smart sampling technique can increase coverage performance by even more than 70%.*

The results in Table 2 clearly show that our novel technique is useful for software engineers, as it can automatically detect real faults in real systems. However, albeit promising, code coverage results could be further improved (Table 3). Therefore, we *manually* analyzed some of the cases in which only low coverage was obtained. We found out that one possible main reason for such results is the handing of interactions with SQL databases.

Even if a database was initialized with valid and useful data, our technique has currently no way to check what is inside the databases. Recall the example of Figure 3: even if there is valid data in the database, our testing tool has no gradient to generate an id matching an existing key in the database. The testing tool should be extended to be able to check all SQL commands executed by the SUT and directly read the content of the database. Then, such information should be exploited with different heuristics in the fitness function when generating the HTTP calls.

Another problem is that, currently, we cannot generate or modify such data directly in the databases. All modifications need to go through the RESTful APIs. This is a showstopper when the RESTful APIs do not provide endpoints to modify the database. This could happen if the API is read-only (e.g., only GET methods), where the content in the database is written by other tools and scripts (this was the case for some of the endpoints in the *catwatch* SUT). In this case, the generation of data in the database should be part of the search as well: a test case would no longer be just HTTP calls but also SQL commands to initialize the database.

RQ4: *Interactions with SQL databases is one of the main challenges preventing the achievement of higher code coverage.*

7 THREATS TO VALIDITY

Threats to internal validity come from the fact that our empirical study is based on a tool prototype. Faults in the tool might compromise the validity of our conclusions. Although EvoMaster has been carefully tested, we cannot provide a guarantee that it is bug free. However, to mitigate the risk and enable independent replication of our experiments, our tool and case study are freely available online at www.evomaster.org.

As our techniques are based on randomized algorithms, such randomness might affect the results. To mitigate such problem, each experiment was repeated 100 times with different random seeds, and the appropriate statistical tests were used to analyze the results.

Threats to external validity come from the fact that only five RESTful web services were used in the empirical study. This is due to the difficulty of finding this kind of application among open-source projects. Furthermore, this is also due to the fact that running experiments on system-level test-case generation is very time consuming. For example, it required 1,348 days of computational resources on a cluster of computers. Although those five services are not trivial (i.e., up to 7,500 lines of code), we cannot generalize our results to other web services.

Our approach is not compared with any other existing technique, as we are not aware of any existing and available tool that can generate *system* tests for RESTful APIs. Comparing with existing *unit* test-generation tools could provide some interesting insight, but it would be out of the scope of this article.

8 CONCLUSION

RESTful web services are popular in industry. Their ease of development, deployment, and scalability makes them one of the key tools in modern enterprise applications. This is particularly the case when enterprise applications are designed with a microservice architecture [41].

However, testing RESTful web services poses several challenges. In the literature, several techniques have been proposed for automatically generating test cases in many different testing contexts. But, as far as we know, we are aware of no technique that could automatically generate integration, white-box tests for RESTful web services. These tests are often what engineers write during the development of their web services [7], using, for example, the very popular library RestAssured.

In this article, we have proposed a technique to automatically collect white-box information from running web services and then exploit such information to generate test cases using an evolutionary algorithm. We implemented our novel approach in a tool prototype called EvoMaster, written in Kotlin/Java, and ran experiments on five different web services for a total of more than 22,000 lines of code.

Our technique was able to generate test cases that found 80 bugs in those web services. However, compared to the existing test cases in those projects, achieved coverage was lower. A manual analysis of the results pointed out that interactions with SQL databases is what is currently preventing the achievement of higher coverage. Future work will need to focus on designing novel heuristics to address such issues.

Furthermore, to achieve a wider impact in industry, it will be important to extend our tool to also handle other popular languages in which RESTful web services are often written, such as JavaScript/NodeJS and C#. Due to a clean separation between the testing tool (written in Kotlin) and the library to collect and export white-box information (written in Java, but technically usable for any JVM language), supporting a new language is just a matter of reimplementing that library, not the whole tool. To make the integration of different languages simpler, our library itself is designed as a RESTful web service where the coverage information is exported in JSON format. However, code instrumentation (e.g., bytecode manipulation in the JVM) can be quite different among languages.

EvoMaster and the employed case study are freely available online on GitHub, under the LGPL 3.0 and Apache 2.0 open-source licenses. To learn more about EvoMaster, visit our webpage at www.evomaster.org.

ACKNOWLEDGMENT

We would like to thank Bogdan Marculescu and Andreas Biørn-Hansen for valuable feedback.

REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.

[2] Subbu Allamaraju. 2010. *Restful Web Services Cookbook: Solutions for Improving Scalability and Simplicity.* O'Reilly Media, Inc.

[3] M. Alshraideh and L. Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability* 16, 3 (2006), 175–203.

[4] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 3–17.

[5] Andrea Arcuri. 2017. RESTful API automated test case generation. In *IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 9–20.

[6] Andrea Arcuri. 2018. EvoMaster: Evolutionary multi-context automated system test generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'18)*. IEEE.

[7] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: We need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959–1981.

[8] A. Arcuri and L. Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[9] A. Arcuri, M. Z. Iqbal, and L. Briand. 2012. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering* 38, 2 (2012), 258–277.

[10] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*. IEEE, 207–212.

[11] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[12] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. 2011. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software* 84, 4 (2011), 655–668.

[13] Cesare Bartolini, Antonia Bertolino, Francesca Lonetti, and Eda Marchetti. 2012. Approaches to functional, structural and security SOA testing. In *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. IGI Global, 381–401.

[14] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. 2009. WS-TAXI: A WSDL-based testing tool for web services. In *International Conference on Software Testing Verification and Validation (ICST'09)*. IEEE, 326–335.

[15] Antonia Bertolino, Guglielmo De Angelis, Antonino Sabetta, and Andrea Polini. 2012. Trends and research issues in SOA validation. In *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. IGI Global, 98–115.

[16] Mustafa Bozkurt and Mark Harman. 2011. Automatically generating realistic test input from web services. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE'11)*. IEEE, 13–24.

[17] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2013. Testing and verification in service-oriented architecture: A survey. *Software Testing, Verification and Reliability* 23, 4 (2013), 261–313.

[18] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 55–66.

[19] Gerardo Canfora and Massimiliano Di Penta. 2006. Testing services and service-centric systems: Challenges and opportunities. *IT Professional* 8, 2 (2006), 10–17.

[20] Gerardo Canfora and Massimiliano Di Penta. 2009. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, 78–105.

[21] Sujit Kumar Chakrabarti and Prashant Kumar. 2009. Test-the-rest: An approach to testing restful web-services. In *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (COMPUTATIONWORLD'09)*. IEEE, 302–308.

[22] Sujit Kumar Chakrabarti and Reswin Rodriquez. 2010. Connectedness testing of restful web-services. In *Proceedings of the 3rd India Software Engineering Conference*. ACM, 143–152.

[23] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. 2002. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing* 6, 2 (2002), 86–93.

[24] Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito, Valentina Mazza, and Marcello Bruno. 2007. Search-based testing of service level agreements. In *Genetic and Evolutionary Computation Conference (GECCO'07)*. ACM, 1090–1097.

[25] Tobias Fertig and Peter Braun. 2015. Model-driven testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1497–1502.

[26] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures.* Ph.D. Dissertation. University of California, Irvine, Irvine, CA.

[27] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE'11)*. 416–419.

[28] G. Fraser and A. Arcuri. 2012. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE'12)*. 178–188.

[29] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[30] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with Evosuite. *Empirical Software Engineering* 20, 3 (2015), 611–639.

[31] Samer Hanna and Malcolm Munro. 2008. Fault-based web services testing. In *5th International Conference on Information Technology: New Generations (ITNG'08)*. IEEE, 471–476.

[32] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* 45, 1 (2012), 11.

[33] Seema Jehan, Ingo Pill, and Franz Wotawa. 2014. SOA testing via random paths in BPEL models. In *IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14)*. IEEE, 260–263.

[34] B. Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[35] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In *Proceedings of the 12th ACM SIGPLAN Workshop on Erlang*. ACM, 77–78.

[36] Yin Li, Zhi-an Sun, and Jian-Yong Fang. 2016. Generating an automated test suite by variable strength combinatorial testing for web services. *CIT. Journal of Computing and Information Technology* 24, 3 (2016), 271–282.

[37] Chunyan Ma, Chenglie Du, Tao Zhang, Fei Hu, and Xiaobin Cai. 2008. WSDL-based automated test data generation for web service. In *International Conference on Computer Science and Software Engineering,* Vol. 2. IEEE, 731–737.

[38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *ACM International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 94–105.

[39] Evan Martin, Suranjana Basu, and Tao Xie. 2006. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA and Web Services Best Practices (SOAWS'06)*.

[40] P. McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[41] Sam Newman. 2015. *Building Microservices.* O'Reilly Media, Inc.

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

[43] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2018. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.

[44] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-based testing of RESTful web services using UML protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*.

[45] R. V. Rajesh. 2016. *Spring Microservices*. Packt Publishing Ltd.

[46] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: A large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering*. Springer, 21–39.

[47] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099.

[48] Harry M. Sneed and Shihong Huang. 2006. WSDLTest—A tool for testing web services. In *8th IEEE International Symposium on Web Site Evolution (WSE'06)*. IEEE, 14–21.

[49] Wei-Tek Tsai, Ray Paul, Weiwei Song, and Zhibin Cao. 2002. Coyote: An XML-based framework for web services testing. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 173–174.

[50] Franz Wotawa, Marco Schulz, Ingo Pill, Seema Jehan, Philipp Leitner, Waldemar Hummer, Stefan Schulte, Philipp Hoenisch, and Schahram Dustdar. 2013. Fifty shades of grey in SOA testing. In *6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13)*. IEEE, 154–157.

[51] Wuzhi Xu, Jeff Offutt, and Juan Luo. 2005. Testing web services by XML perturbation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE, 10–pp.

[52] Chunyang Ye and Hans-Arno Jacobsen. 2013. Whitening SOA testing via event exposure. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1444–1465.

Received April 2018; revised August 2018; accepted October 2018