

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 paper, we consider testing from the point of view of the developers, which do 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 pre-defined 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 do show that our novel technique did automatically find 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 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: Search-based 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.* 1, 1 (January 2019), 37 pages. <https://doi.org/>-

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 for example in companies such as Netflix, Uber, Airbnb, eBay, Amazon, Twitter, Nike, etc [45].

Author's address: Andrea Arcuri, Kristiania University College, Oslo, Norway, University of Luxembourg, Luxembourg, Luxembourg, andrea.arcuri@kristiania.no.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery. 1049-331X/2019/1-ART \$15.00
<https://doi.org/>-

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 thousand 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 for example Google², Amazon³, Twitter⁴, Reddit⁵, LinkedIn⁶, etc. In the Java ecosystem, based on a survey⁷ of 1700 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 candidates⁸.

Testing web services, and in particular RESTful web services, does pose 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 nowadays are shifting to REST services, which usually employ JSON (JavaScript Object Notation) as data format for the 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 paper, 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, do 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 engineer in industry [7], writing this kind of tests 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 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 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 a HTTP endpoint X we first need to have a resource Y , this information can be derived by analyzing their URIs, and so we can create test templates in which we first create Y (by doing a 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 pre-defined templates. Note: even if the structure of a test is fixed, there is still the need to search for their right input

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/>

data, e.g. HTTP headers, URL parameters and payloads for POST/PUT/PATCH methods (typically in JSON and XML format).

We implemented a tool prototype called EvoMASTER, and carried out experiments on five open-source RESTful web services. Those systems range from 718 to 7534 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.

In particular, this paper provides the following research and engineering contributions:

- We designed a novel technique that is able to generate effective tests 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 which shows that, even if our tool is in a 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 GitHub⁹.

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

The paper 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 presents our empirical study. Threats to validity are discussed in Section 7. Finally, Section 8 concludes the paper.

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 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 for example RFC 7230¹⁰ and RFC 7231¹¹.

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

Verb/Method: the type of operation to do, like getting a specific web page.

Resource path: an identifier to specify on which resource the HTTP operation should be applied, like for example the path of an HTML document to get.

⁹<https://github.com/arcuri82/EvoMaster>

¹⁰<https://tools.ietf.org/html/rfc7230>

¹¹<https://tools.ietf.org/html/rfc7231>

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, like the HTML text of a web page that is returned as 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 only needs to check if a resource exists, or if he only needs to get its headers.

POST: send data to the server, e.g., 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 with PUT, where the resource is fully replaced with a new one.

TRACE: echo the received request. This is useful to find out if 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 a 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, like 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 for example further specify that a new resource was created (201), e.g., as a result of a POST command, or that nothing is expected in the response body (204), e.g., as a result of a DELETE command.

3xx: those codes are used for redirection, e.g., 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 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. The user will need to include such header in all of his HTTP requests.

2.2 REST

For many years, the main way to write a web service was to use SOAP (Simple Object Access Protocol), which is a communication protocol using XML enveloped messages. However, in recent years, there has been a clear shift in industry toward REST (Representational State Transfer) when developing web services. All major players are now using REST, such as for example Google¹², Amazon¹³, Twitter¹⁴, Reddit¹⁵, LinkedIn¹⁶, etc.

The concepts of REST were first introduced in a highly influential PhD thesis [26] in 2000. REST is not a protocol (like SOAP is), but rather a set of architectural guidelines on building web services on top of HTTP. Such client-server applications needs to satisfy some constraints to be considered RESTful, like 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, e.g., a resource should be deleted with a DELETE request and not a POST one.

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)

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*, i.e., they do 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 plural name. Creating a new element on such 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, i.e. “/products/{id}”. Given an empty state (e.g., an empty database), before testing a resource like “GET /products/{id}”, a tester would require first to do a “POST /products” to create the element he 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, and that should be specified in the headers of the request.

Another aspect of REST is the so called HATEOAS (Hypermedia As The Engine Of Application State), where each resource representation should also provide links to other resources (in a similar

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>

way as links in web pages). For example, when calling a *GET /products*, not only all products should be returned, but also there should be links to what other methods are available. Ideally, given the main entry point of an API, such API should be fully discoverable by using those links. However, the use of HATEOS is quite rare in practice [46], mainly due 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 entail 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 paper we use the term REST, the presented novel techniques would work as well to any Web API which 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 approach 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 naive 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, where one wants 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, where Genetic Algorithms (GAs) are perhaps 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 for example 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 SUT); (3) some testing targets can be *infeasible* to cover.

Algorithm 1 High-level pseudo-code of the Many Independent Objective (MIO) Algorithm [4]

Input: Stopping condition C , Fitness function δ , Population size limit n , Probability of random sampling P_r , Start of focused search F

Output: Archive of optimised individuals A

```

1:  $T \leftarrow \text{SETOFEMPTYPOPULATIONS}()$ 
2:  $A \leftarrow \{\}$ 
3: while  $\neg C$  do
4:   if  $P_r > \text{rand}()$  then
5:      $p \leftarrow \text{RANDOMINDIVIDUAL}()$ 
6:   else
7:      $p \leftarrow \text{SAMPLEINDIVIDUAL}(T)$ 
8:      $p \leftarrow \text{MUTATE}(p)$ 
9:   end if
10:  for all  $k \in \text{REACHEDTARGETS}(p)$  do
11:    if  $\text{ISTARGETCOVERED}(k)$  then
12:       $\text{UPDATEARCHIVE}(A, p)$ 
13:       $T \leftarrow T \setminus \{T_k\}$ 
14:    else
15:       $T_k \leftarrow T_k \cup \{p\}$ 
16:      if  $|T_k| > n$  then
17:         $\text{REMOVEWORSTTEST}(T_k, \delta)$ 
18:      end if
19:    end if
20:  end for
21:   $\text{UPDATEPARAMETERS}(F, P_r, n)$ 
22: end while
23: return  $A$ 

```

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* the populations for uncovered targets and is thus 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, initialised 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 tradeoff 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, but then, with the passing 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. Pseudocode of MIO is listed in Algorithm 1.

It is important to notice 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 (such as for example 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.

3 RELATED WORK

Canfora and Di Penta provided a discussion on the trends and challenges of Service-oriented architecture (SOA) testing [19]. Afterwards, they provided a more in detail survey [20]. There are different kinds of testing for SOA (unit, integration, regression, robustness, etc.), which also depend on which stakeholders are involved, e.g., service developers, service providers, service integrators and third-party certifiers. Also Bertolino et al. [13, 15] 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 throughout 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 WSDL (Web Services Description Language). Different strategies have been proposed, such as for example [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 BPEL (Web Services Business Process Execution Language), different techniques can be used to generate tests for those compositions [33, 50]

Black-box testing has its advantages, but also 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 paper, 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 in which “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 Rodriguez [22] defined a technique to formalize the “connectedness” of a RESTful service, and generate tests based on such 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 usage of evolutionary techniques for testing web services, Di Penta et al. [24] proposed an approach for testing Service Level Agreements (SLA). 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 paper, 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., like 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 (recall 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 which are stored in an archive. Only at the end of the search the final test suite is created. So, we need to define how to represent test cases for RESTful APIs.

In our context, a test case is one or more HTTP requests towards 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 the components of a 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 paper we will focus just on such format, which we fully support. Handling other less popular formats, like for example XML, would be just 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 Swagger¹⁷, which is currently available for more than 25 different programming languages. Another tool is RAML¹⁸, 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

¹⁷<http://swagger.io>

¹⁸<http://raml.org>

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

```

"/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" }
      }
    }
  }
}

```

service. Therefore, the first step, when testing such RESTful service, is to retrieve its Swagger JSON definition.

Figure 1 shows an extract from a Swagger definition of one of the systems we use in the empirical study. The full JSON file is more than 2000 lines of code. In that figure, there is the definition for two HTTP operations (GET and PUT) on the same resource. To execute a GET operation on such resource, there is the need of two values: 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 such 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 as well (e.g., “author”). When a test case is written for such

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

```

"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"}
  }
}

```

PUT operation, besides specifying an “id” in the path, one would also need to instantiate such “ActivityProperties” object, and marshal it as a JSON string to add in the body of the HTTP request.

To handle all 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*. These latter represent the different variable parts of the HTTP requests, such as for example the query parameters and the content of the body payload. For each gene, we explicitly specify if it is used to represent a query parameter (including the name of such parameter), a path parameter, body payload or a 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 alphabetic 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 on all of its contained genes (if any). For example, a *DateTime* is composed of two different genes (*Date* and *Time*), which in turn are composed of three *Integer* genes, for a total of $1 + 2 + 3 + 3 = 9$ genes. When such *DateTime* gene is used in a HTTP call, a well-formed string representing the date with the time will be built (using the RFC3339 format¹⁹).

¹⁹<https://www.ietf.org/rfc/rfc3339.txt>

Genes can also have specific constraints based on the Swagger schema, like minimum and maximum values for each numeric variable. The search operators will take such information into account when sampling and mutating the genes.

- *Array*: containing zero or more genes, all of the same type (e.g., all *Integer* genes). To avoid too large test cases (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*.
- *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 loops 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*, the *month* and the *date*. All these genes are constrained within valid values (e.g., months can only have values 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 such gene is controlled with a specific probability, which can be set to 0 (and so prevent all mutations on it once initialized with some specific value). This type of genes was important to handle some cases of URL path parameters which should not be modified once set (this will be discussed in more details 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: 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 for example 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*, the *minute* and the *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 (recall 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 only happens 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 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 to be treated specially, because how to best modify them strongly depends on the fitness function. In our context, either one character is randomly modified, or 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 system under test (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 like 2xx, but also user errors and server errors, regardless of the achieved coverage). When two test cases have 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 an heuristic is defined to check if it is close to be covered (e.g., values close to 1), or far from it (e.g., values close to 0), once all 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 if 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. So, 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 paper, for our prototype, we started by focusing on JVM languages, like 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). Such an approach is the same used in unit test generation tools for Java like EvoSuite [27].

²⁰<https://github.com/electronicarts/ea-agent-loader>

²¹<http://asm.ow2.org/>

Measuring coverage is not enough. Knowing that a test case cover 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 a test data is to solve a constraint. For example, given the constraint $x == 0$, although neither 5 nor 1000 does solve such constraint, the value 5 is *heuristically* closer than 1000 to solve 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). Besides primitive types, we also use customised 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 which calculate custom heuristics (e.g., based on Euclidian distance when comparing if 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 smoothing 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 the 10 statements in such 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 the test cases *A* and *B* are executed, all the targets for the lines 1 to 7 will have maximum heuristic score $h = 1$ (i.e., they are covered), where line 8 would have 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 such test to see if it can avoid throwing an exception, and so possible reach line 9. However, there is no gradient to remove the cause for which the exception is thrown, apart from penalizing tests like *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 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 gradient provided by the code coverage heuristics (e.g., the branch distance). However, not all HTTP requests lead to execute the business logic of the SUT. When applications are written with frameworks such as Spring and JEE, a HTTP request could be completed directly in such frameworks before any of developers' code that we want to test is executed. This for example could happen due to input validation when constraints are expressed with Java annotations. 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 does target 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 such values. In unit testing, a test data generation tool would run in the same process as the one where the tests are evaluated, and so such values could be directly read. It would be possible to do the same for system testing: the testing tool and the SUT could run in the same process, e.g., 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. For 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 the coverage and branch distance information in a JSON format. The testing tool, when generating and running tests, would use such API to determine the fitness of these tests. The testing tool would be just one, but, then, for each target programming language (e.g., Java, C# and JavaScript) we would just need its library implementation for the code instrumentation.

Such an 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, such 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 particular, in our novel approach we consider the two following 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, like the application servers (e.g., Jetty or Tomcat), or ORM libraries like Hibernate.
- by default, when querying the SUT for code coverage and branch distance information, not all information is retrieved: only the one of newly covered targets, or better branch distance, is returned. The reason is that, if the SUT is 100 thousand lines of code, then you do not want to un/marshal JSON data with 100 thousand 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 info 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, like 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 such 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 really 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 if the whole system under test does crash (e.g., a segmentation fault in C programs) when a test is executed. Such 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 a violation of an unspecified pre-condition (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 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 if the expected output should really be 9 (which is the actual value returned when calling *foo(x)*). A developer could look at such generated test, and then confirm if indeed 9 is the expected output. But, even if he does not check it, such 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., Jenkins²²). If a modification to the source code leads *foo* to return a different values than 9 when called with input 5, then that test case will fail. At this point, the developers would need to check if indeed the recently introduced change does break the function (i.e., it is a bug), or rather 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, a HTTP invocation in which the returned status is 403 (unauthorized) can detect regression faults in which the authorization check 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 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 such service. A typical example is thrown exceptions: the application server will not crash if an exception is thrown in the business logic of a RESTful endpoint. Such exception would be caught, and a 5xx code (e.g., 500) would be returned. Note: if the user sends invalid inputs, he should get back a 4xx code, not a 5xx one. Not doing input validation and letting the endpoint throwing an exception would have two main problems:

- the user would not know that it is his fault, and so just think it is a bug in the web service. Inside an organization, such developer might end up wasting time in filling a bug report, for example. Furthermore, the 5xx code would not give him any hint on how to fix the way he is calling 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 to be 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 a inconsistent state, making the entire system work incorrectly.

Figure 3 shows a simple example of endpoint definition which contains bugs. This code is from one of the projects used in the empirical study. In that particular case, a resource (a media file) is referenced by id in the endpoint path (i.e., *@Path("/{id}/file")*). Such id is used to load such resource from a database (i.e., *dao.read(id)*), but there is no check if it exists (e.g., if different from *null*).

²²<https://jenkins.io>

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, does lead to a 500 code due to a null pointer exception.

```
@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) {
        ...
    }
    ...
}
```

Therefore, when a test is created with an invalid id, the statement *mediaFile.getUri()* does result in a null pointer exception. Such exception is propagated to the application server (Jetty, in this case), which will create a 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 tests. 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 rather just capture the behavior of the SUT, and so asserting that the returned status code is indeed 5xx (we will see examples of this later in the paper when we will show some of the generated tests on our case study, like in Figure 7). These tests could point to faults in the SUT though, and so could be more valuable than the other tests. How to best convey such info 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. However, more research will be needed to check if other oracles could be added as well, like for example 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 paper. The tool is called EvoMASTER, and it is released under the LGPL open-source license.

For handling the SUT (start/stop it, and code instrumentation), we have developed a library for Java, which 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. Such test suites will be fully self-contained, i.e., they will also deal with the starting/stopping of the SUT. The test cases

are configured in 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 toward the SUT, we use the highly popular RestAssured²³ 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 paper.

4.6 Manual Preparation

In contrast to tools for unit testing like 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 does require some manual configuration.

The developers of the RESTful APIs need to import our library (which is published on Maven Central²⁴), and then create a class that extends the *EmbeddedSutController* class in such library. The developers will be responsible to define how the SUT should be started, where the Swagger schema can be found, which packages should be instrumented, etc. This will of course vary based on how the RESTful API is implemented, e.g., if with Spring²⁵, DropWizard²⁶, Play²⁷, Spark²⁸ or JEE.

Figure 4 shows an example of one such class we had to write for one of the SUTs in our empirical study. That SUT uses Spring. That 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 to do 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 such class.

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

- RESTful APIs are supposed to be stateless (so 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 particular case of the class in Figure 4, the only state is in the database, and so we use a support function to clean its content (but without changing the existing schema).
- if a RESTful API requires some sort of authentication, such 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, e.g., to do HTTP calls with and without authentication.

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>

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

```

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";
    }
}

```

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 of 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 for example to initialize the first population of a Genetic Algorithm. Other algorithms, like 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 Swagger²⁹ 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 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 changing the structure of the test, by adding or removing HTTP calls. This is similar to how test sequences are evolved in unit testing tools like EvoSuite [27]. However, there is a major difference, which is related on 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 just add new calls 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):

```
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. But there might be no operation (e.g., POST or PUT) to create such 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 such id could be generated dynamically by the SUT (e.g., if the same id is used to store the record of the product in a SQL database). Even if by chance we would get a “POST /products” that creates a product with id 100, running such test twice could result in a different id, with the test behaving differently (and so likely becoming *flaky*). The standard search-based heuristics like 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).

²⁹<https://swagger.io/>

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 rather use a predefined *template* to define the structure of such 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 such action on this endpoint is the main target we want to test, which will be the last HTTP call in such test. Other HTTP calls will be added before it to try to bring the state of the SUT in the right configuration. What kind of HTTP calls will be added depends on the HTTP verb of the target and its URI. Here, 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
```

Such endpoint might represent a single named resource, or a collection (whether the name is in plural form might give 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. So, we add a random number k of POST calls (if POST is possible on such 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: here for simplicity in those examples we are not showing headers, query parameters nor 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, i.e. the controller handling that endpoint will accept any request for a URIs 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 if there is any creation operation (i.e., POST or PUT) on such resource. If none is available, we look at the closest *ancestor* resources in the URI, which in this case is “/products”, and see if there is a creation operation on it. If so, we add it, and we end up with a test with two HTTP calls, e.g.:

```
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, we do not need to handle it specially. So, we will just 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. But 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, like when the `id` in the URI is the same used in a SQL database (where `ids` might be generated by the database according to some specify rules). Here, we consider these two different cases, i.e., given a certain probability we

either sample a test with a single PUT, or a test in which we first try to create such resource with a POST (if any is available on it or on an ancestor), like for example:

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

5.2.4 PATCH Template. A PATCH does a partial update on a resource. So to test it, we need to create such resource first. Note, this is different from PUT, where the PUT itself could create the resource if missing. So, 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 needed to have more than one PATCH in sequence on the same resource. So, in this template, we randomly choose if having either one or two PATCHes in the test, e.g.:

```
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, like in the other templates, we need a POST/PUT to first create such resource, e.g.:

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

5.2.6 Intermediate Resources. In the templates discussed in the previous subsections, 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. For example, 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 such endpoint. However, such GET call would likely result in a 404 *not found* error.

To avoid this kind of problems, in all the templates discussed so far, we add the pre-step of recursively generating all the needed intermediate resources. This can be identified by checking if 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”. So, 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 if we want to test an endpoint like “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 subsection, 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 such 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 to create such resources directly, then we would have a test like:

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

Here, we would choose the ids (e.g., randomly), and then just 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 such newly generated resource. The recommended approach in HTTP (RFC 7231) is to put such information in the *Location* header of the response. This could be either a relative URI to the newly create 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”. So, 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 such resources. So, 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., like 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 such response is structured data (e.g., JSON or XML), we check if 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 such value from the POST response and use it as 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 a HTTP call (e.g., the payload), or changing 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 (Section 5.2.7), or add calls on unrelated endpoints or states, which would just make the test more expensive to run. The only exception is the parameter *k* in the GET Template (Section 5.2.1), i.e., we allow to add/remove POST calls on the collection.

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

Note: 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. So, if tests sampled with our strategy lead to worse fitness, they will just die out during the search compared to the tests originally sampled at random.

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 test9() throws Exception {

    String location_categories = "";

    String id_0 = given().accept("*/*")
        .header("Authorization", "ApiKey moderator") // moderator
        .contentType("application/json")
        .body("{\"group\":\"9oRyB\", \"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);
}
```

5.2.9 Premature Stopping. The use of our templates is warranted for when we need to create resources before doing actions on them. But doing a call to create such a resource might fail. This could happen due to bugs in the SUT, or due 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 following 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 (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 location header is missing, `resolveLocation()` still tries to create a valid URL.

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

```
@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(isValidURLorEmpty(location_x));

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

    assertTrue(isValidURLorEmpty(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(isValidURLorEmpty(location_z));

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

6 EMPIRICAL STUDY

In this paper, 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 achievements 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

Table 1. Information about the five RESTful web services used in the empirical study. 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 a SQL database.

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

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 BigQuery³⁰ to analyze the content of the Java projects hosted on GitHub³¹, 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 paper, we manually chose four different RESTful web services 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 have developed in the past 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 paper, we gather together all these services in a single GitHub repository³⁶, including all the needed configuration files to run EvoMASTER on these web services.

Those five RESTful web services contain up to 7500 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, such services usually become split if growing too large, as to make them manageable by a single, small team. This, however, does also imply that enterprise applications can end up being composed of hundreds of different services. Currently, EvoMASTER does focus 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 Central³⁷). In particular, this class extends the `EmbeddedSutController` class, and starts the driver functionality with `InstrumentedSutStarter`. This latter will start a RESTful API used by the EvoMASTER main process (which by default tries to connect to port

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 2. Results of the experiments, based on 100 runs per SUT. We considered EvoMASTER with a 100 thousand 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.

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

40100) to control the SUT (e.g., to send commands to start/stop/reset it). Note the specific configurations for *features-service*, like 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 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 much, much larger than the overhead of the search algorithm code in EvoMASTER, we did not use time as 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 stopping criterion, as that is more user-friendly.

For each run of the algorithm, not only we need to run the EvoMASTER core process, but also the the EvoMASTER driver and the SUT itself (so three different processes). Due 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 1348 days of computational resources.

For these experiments, all the SUTs were configured to use an embedded SQL database, such as H2³⁸ and Derby³⁹.

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 thousand HTTP calls budget and with $P = 0.6$ for the sampling probability. Although during the search we evaluated 100 thousand 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

³⁸www.h2database.com

³⁹db.apache.org/derby

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.

```

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);
}

```

only keep tests that contribute to cover 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 one hand, an endpoint might fail in different ways due 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 paper 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 we previously showed in Figure 3. A generated test revealing such bug is shown in Figure 7. Note: 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 return 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 (like in the case of Figure 3 and Figure 7). However, defining what is *valid* and *invalid* is not necessarily obvious. Swagger definitions do provide the possibility to write some forms of *pre-conditions*, but complex constraints (e.g., involving relations among more than one input field) cannot be currently expressed. Such constraints might

Table 3. Statement coverage results for the generated tests compared to the ones obtained by the already existing, manually written tests. Coverage was measured by running the tests inside IntelliJ IDEA. For the manual tests, we distinguish on whether they do HTTP calls toward the API, or are rather lower level unit or integration tests.

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

still exist, but they might be rather 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 paper), it is not possible to state whether the found faults are critical or not. However, a preliminary analysis of some of these faults seem to point out that the majority of the found faults are due to improper handling of *invalid* inputs. This is not unexpected. A developer might concentrate on testing and fixing the basic functionalities of his APIs in “happy day” scenarios, and not considering all the different ways they can be called with invalid inputs. It is so reasonable to expect that finding this kind of faults would be more common when applying an automated tool on such kind of existing APIs. 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 due 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, such endpoint does return the 500 HTTP status code due an uncaught `IndexOutOfBoundsException`.

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

Besides finding faults, the generated test suites can also be used for regression testing. To be useful in such context, it would desirable that such test suites would 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 if 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 the running of the tests in IntelliJ, and their coverage analysis, was done manually.

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

Regarding the manual tests, besides their total coverage, we also report values separately based on two non-overlapping groups: (1) if the tests do HTTP calls toward the API (i.e., the same kind of tests that we generate with EvoMASTER), or (2) if they are rather lower level unit or integration

Table 4. Average numbers of covered test targets. Values in bold are the highest in their row.

SUT	Budget	Base	Smart Sampling Probability				
		0	0.2	0.4	0.6	0.8	1
<i>catwatch</i>	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
<i>features-service</i>	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
<i>proxyprint</i>	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
<i>rest-news</i>	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
<i>scout-api</i>	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

tests. The results are very different among the SUTs. On 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 does decrease when all 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 in general 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 in general 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 for example would ignore tests that can crash the SUT but that do not increase coverage.

Table 5 is based on the data from Table 4, but, 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 on average between 12.5% and 15.1%, 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. Whether our

Table 5. Relative improvements (in terms of average numbers of covered targets) of smart sampling technique compared with default version of EvoMASTER. Values in bold are the highest in their row.

SUT	Budget	Smart Sampling Probability				
		0.2	0.4	0.6	0.8	1
<i>catwatch</i>	10k	+0.0%	+0.1%	+0.2%	+0.2%	+0.5%
	100k	-0.0%	+0.0%	+0.0%	+0.0%	+0.0%
<i>features-service</i>	10k	+72.1%	+79.1%	+77.2%	+77.5%	+75.7%
	100k	+23.4%	+21.5%	+20.0%	+19.3%	+16.4%
<i>proxyprint</i>	10k	+3.8%	+5.2%	+2.6%	+2.7%	+2.7%
	100k	+0.1%	-0.5%	+0.3%	-1.0%	-0.8%
<i>rest-news</i>	10k	-1.8%	+0.3%	-1.2%	-0.3%	-4.7%
	100k	-0.9%	+1.0%	-0.1%	+1.3%	-2.9%
<i>scout-api</i>	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%

Table 6. In details 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
<i>catwatch</i>	10k	964.2	965.8	0.66	< 0.001
	100k	971.5	971.8	0.51	0.830
<i>features-service</i>	10k	282.5	500.5	1.00	< 0.001
	100k	446.9	536.2	0.75	< 0.001
<i>proxyprint</i>	10k	1308.2	1342.4	0.71	< 0.001
	100k	1414.9	1418.5	0.46	0.376
<i>rest-news</i>	10k	251.1	248.0	0.41	0.041
	100k	262.0	261.9	0.43	0.080
<i>scout-api</i>	10k	1328.1	1506.2	0.91	< 0.001
	100k	1414.6	1869.5	1.00	< 0.001

technique is successful depends on the schema of the RESTful APIs. If an API does not have any hierarchical resource (as it 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 out to which P is best, i.e. if either a low value or a high one.

Given the current data, it could be advisable to choose a middle value like $P = 0.6$. With such P , Table 6 shows the comparisons with the base configuration (i.e., $P = 0$). To check if indeed our experiments show that smart testing provides better results, Table 6 also reports standardised effect

sizes and p-values of statistical tests [8]. In particular, we used the Vargha-Delaney \hat{A}_{12} effect size, and the non-parametric Mann-Whitney-Wilcoxon U-test. Results in Table 6 clearly shows, 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 budgets, evolutionary algorithms would not have time to do much evolution, and so 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 paper, we considered two different search budgets: maximum 10 thousand HTTP calls, and maximum 100 thousands. 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 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 thousands and 100 thousands were rather arbitrary, based on our experience. Furthermore, an issue is that such 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 thousands, improvement is very large, around +70%. However, when a larger budget is used, although there is still improvement, such improvement is much smaller, in the order of just 16%-23%. This can be explained by the fact that, although smart sampling does give 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 round. With 10 thousand search budget, improvements are *at most* 19.5%, whereas for 100 thousands they are *at least* 23.4%, with the peak at 33.8%. A possible explanation is that the smart sampling allows to cover more endpoints, however, you still need the right input parameters (i.e., HTTP headers, body payloads and URL query parameters). With only 10 thousands as search budget, it seems that there is not enough time to evolve the right input parameters, and so the boost of smart sampling cannot express itself in its fullness.

Another interesting phenomenon that we can observe in Table 4 is that, for both *features-service* and *scout-api*, smart sampling configuration at 10 thousand search budget gives better (506.1 vs. 446.9 for *features-service* and 1587.2 vs. 1414.6 for *scout-api*) performance than not using smart sampling with a 10 times larger search budget (i.e., 100 thousands).

Table 7 shows the same kind of analysis done in Table 4, but only considering the faults due to 5xx return statuses instead of all testing targets. In such 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 involving 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 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 an unknown numbers and types of faults in them. In the future, it would be important to study how results would vary in

Table 7. Average numbers of faults detected based on 5xx status responses. Values in bold are the highest in their row (before trunking the values to 1 digit precision).

SUT	Budget	Base 0	Smart Sampling Probability				
			0.2	0.4	0.6	0.8	1
<i>catwatch</i>	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
<i>features-service</i>	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
<i>proxyprint</i>	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
<i>rest-news</i>	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
<i>scout-api</i>	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

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 even by 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 did *manually* analyze some of the cases in which only low coverage was obtained. We found out that one possible main reason for such results is the handling 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 toward generating 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 info 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 does 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 such case, the generation of data in the database should be part of the search as well: a test case would not be any more 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 such 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 such 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 analyse the results.

Threats to external validity come from the fact that only five RESTful web services were used in the empirical study. This was due to the difficulty of finding this kind of applications 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 1348 days of computational resources on a cluster of computers. Although those five services are not trivial (i.e., up to 7500 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 scope.

8 CONCLUSION

RESTful web services are popular in industry. Their ease of development, deployment and scalability make 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. This kind of tests are what often engineers write during the development of their web services [7], using, for example, the very popular library RestAssured.

In this paper, we have proposed a technique to automatically collect white-box information from the running web services, and, then, exploit such information to generate test cases using an evolutionary algorithm. We have 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 thousand lines of code.

Our technique was able to generate test cases which did find 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 also be important to extend our tool to also handle other popular languages in which RESTful web services are often written in, such as for example 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 re-implementing 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 Bjørn-Hansen for valuable feedback. 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).

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 (TSE)* 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 (STVR)* 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 (SBSE)*. 3–17.
- [5] Andrea Arcuri. 2017. RESTful API Automated Test Case Generation. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 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)*. 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 (EMSE)* (2018), 1–23.
- [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 (STVR)* 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 (TSE)* 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 *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*. 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 (TSE)* 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 (JSS)* 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 *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. 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 *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*. 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 (STVR)* 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 Int. Conference on Automated Software Engineering (ASE)*. 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 *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World*. 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)*. 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.
- [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)*. 416–419.
- [28] G. Fraser and A. Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 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 (EMSE)* 20, 3 (2015), 611–639.
- [31] Samer Hanna and Malcolm Munro. 2008. Fault-based web services testing. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*. IEEE, 471–476.
- [32] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
- [33] Seema Jehan, Ingo Pill, and Franz Wotawa. 2014. SOA testing via random paths in BPEL models. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 260–263.
- [34] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* (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 twelfth 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 *Computer Science and Software Engineering, 2008 International Conference on*, Vol. 2. IEEE, 731–737.
- [38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 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 2006)*.
- [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 (TSE)* 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] RV 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. 2017. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering (TSE)* (2017).
- [48] Harry M Sneed and Shihong Huang. 2006. WSDLTest-a tool for testing web services. In *Web Site Evolution, 2006. WSE'06. Eighth IEEE International Symposium on*. 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 *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on*. 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 *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 154–157.
- [51] Wuzhi Xu, Jeff Offutt, and Juan Luo. 2005. Testing web services by XML perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 10–pp.
- [52] Chunyang Ye and Hans-Arno Jacobsen. 2013. Whitening SOA testing via event exposure. *IEEE Transactions on Software Engineering (TSE)* 39, 10 (2013), 1444–1465.