A Survey of Test Case Generation Strategies for RESTful APIs

Călin Georgescu
Delft University of Technology
Delft, The Netherlands
c.a.georgescu@student.tudelft.nl

ABSTRACT

REST API testing is a field whose popularity has increased in the aftermath of the de facto standardization of the REST architectural principles in many areas of the World Wide Web. This paper offers a comprehensive survey of available tools for testing RESTful APIs. The survey covers 32 papers in the field, including approaches for black- and white-box testing, extensions that address current challenges, empirical analyses of published strategies, and industrial applications. We outline the most imposing challenges currently facing the field and highlight potential research opportunities for future work to tackle.

1 INTRODUCTION

With the tremendous increase in popularity of component-centric architectural styles such as microservices, exposing modules' functionality through a systematic Application Programming Interface (API) has become a standard practice of paramount importance due to the indispensible requirement of inter-service web-based communication. The Representational State Transfer (REST) [18] architectural pattern has emerged as the cornerstone of API design thanks to its effective decoupling of the underlying application's state and the API's exposed interface. Because of their ubiquity and positioning as the application's entry point, REST APIs require substantial testing. Effectively verifying the functionality of the communication layer of services is an essential step in the overarching system's quality assurance process. However, API testing poses a number of challenges that make manual testing challenging, complex, and time-consuming for developers [9, 28, 29]. Because of the laborious nature of generating API test cases, researchers have been exploring ways to of automating this task using different approaches.

REST API testing poses nuanced challenges that intertwine many complex attributes of the system, from the underlying state, to the structure of the exposed interface, and to the available information. The need for automated ways to reason about and provide solutions to these problems has led to emergence of an active and developing research field into various ways of generating meaningful test cases for such systems. In a broad sense, the work carried out in this field has concentrated around two epicentric paradigms. Black-box (BB) testing treats the API as an opaque process, assuming no access to the source code of the system and leveraging the API as a simple mapping between input and output. White-box (WB) testing, by contrast, requires access to the system's source code and exploits its structure to develop more informed test case generation strategies.

This paper surveys the research conducted in the domain of REST API testing, seeking to provide a comprehensive description of the landscape of this field. We compile a broad list of testing approaches and applications, dividing the conceptually into three categories: black-box methods, white-box methods, and empirical applications.

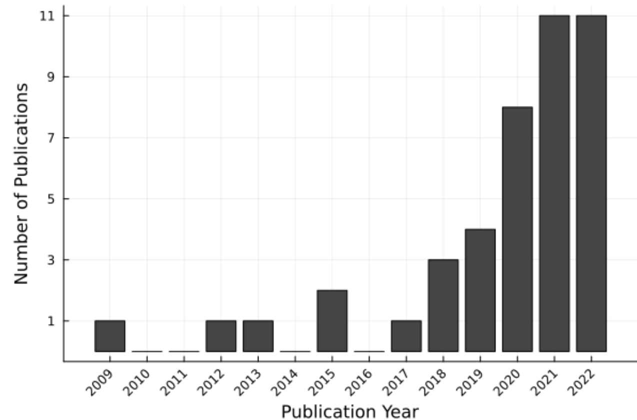

Figure 1: Cumulative distribution of REST API testing publication between 2009 and 2022.

The distribution of publications over time is displayed in Figure 1. In total, we surveyed 43 papers related to automated REST API test case generation. We provide a more detailed breakdown of the distribution in Section 3.2

We are only aware of one other survey regarding the testing of RESTful APIs, carried out in Ehsan et al. [16]. In their work, the authors focus on extrapolating the main challenges that the REST API testing faces, as well as providing a high-level view of the approaches present in literature, based on only 16 publications. This paper seeks to provide not only a more comprehensive view of the landscape, but also a detailed description of how state-of-the-art tools address concrete challenges in practice. In summary, the main contributions of this paper are the following:

(1) The paper provides a comprehensive introduction to the task of RESTful API testing, accompanied by unifying definitions that seek to provide common context to the numerous approaches in literature.
(2) The paper provides a detailed survey 33 REST API testing papers split between black-box approaches, white-box approaches, and applications.
(3) The paper extrapolates current challenges to open problems and identifies possible avenues of future research.

The remainder of the paper is structured as follows. Section 2 provides the necessary background information, definitions, and terminology, that are required to understand existing literature. Sections 4 and 5 provide overviews of techniques following the black-box and white-box testing paradigms, respectively. Section 6 presents the available empirical analyses an comparisons of available tools. Finally, section 7 outlines current challenges and opportunities within the field, and Section 8 concludes the paper.

2 BACKGROUND

This section reviews the common terminology used in the field and provides accompanying examples to help build intuition for readers unfamiliar with the field.

RESTful API. Any Web API (or service) that conforms to the REST architectural style rules proposed by Fielding [18]. The cornerstone of RESTful API functionality resides in their ability to provide a stateless, uniform interface that allows for the creation, reading, update, or deletion (CRUD) of a resource.

Resources. The primary abstraction of REST for data representation is the resource. In general, a resource can be any data handled by the API, such as strings, documents, and images. Such data can be manipulated through the usage of CRUD operations, which in practice usually map to HTTP requests. Resources are mapped to their corresponding Unique Resource Identifiers (URIs).

HTTP Requests. The Hypertext Transfer Protocol is the cornerstone protocol of the Web and defines a standardized set of rules for transferring data over a network. Communication over HTTP is carried out in terms of messages. Messages can either be requests (from a client to a server) or responses (from the server back to the client) that correspond to previous requests. Requests are composed of the following elements: a *path* specifying the location of the resource to action, a *method* that defines the type of operation to perform on the resource, a *header* that carries information about the request, and an optional *body* which carries the payload of the message.

HTTP Methods. A total of 9 HTTP methods (also called *verbs* or *actions*) exist, but we mainly focus on 4 essential methods that best correspond to the CRUD paradigm. `GET` requests the retrieval of a specific resource without any modification. `POST` requests that a new resource be created from the message's data. `PUT` requests the update of an existing resource by replacing its attributes with the data included in the payload. `DELETE` requests that an existing method be deleted.

OpenAPI Specification. An OpenAPI Specification (OAS), also referred to as a Swagger Specification in literature (due to a recent name change), is a document that describes the schema of an API in terms of the capabilities and constraints of the service[1]. Almost all available tools use such a schema to extract information about the API under test. An excerpt of such a schema in JSON format is provided in Figure 2. This is an adapted version of the specification of Swagger's pet store API[2]. Most information included in OAS documents consists of formal descriptions of the structure of the API, such supported paths (line 2) and methods (line 3), details about parameters (lines 9-15), and possible responses (lines 16-18). Natural language information is also given in terms of summaries and descriptions (lines 4 and 5), that describes informal properties, semantics, and relations between parameters.

RESTful API Test Cases. A RESTful API test cases is an ordered sequence of fully-specified HTTP requests. A test case is *valid* if all of its requests abide by the OAS. The test case *succeeds* if all of its requests receive responses that align with the OAS in terms of response code and payload - otherwise the test *fails*. An example test case is given in Figure 3. In this example, the first request (line 1) creates a `Pet` resource with an `id` of 1 via a `POST` requst, before uploading its corresponding image (line 2) and subsequently modifying it (line 3) through a `PUT` request.

[1]https://spec.openapis.org/oas/latest.html
[2]https://petstore.swagger.io/v2/swagger.json

```
1  "paths": {
2   "/pet/{petId}/image": {
3    "post": {
4      "summary": "uploads an image",
5      "description": "",
6      "operationId": "uploadFile",
7      "consumes": ["multipart/form-data" ],
8      "produces": ["application/json"],
9      parameters": [{
10       "name": "petId",
11        "in": "path",
12        "description": "ID of pet to update",
13        "required": true,
14        "type": "integer",
15        "format": "int64"}],
16     "responses": {
17      "200": { "description": "Success" },
18      "404": { "description": "Not found" } }
19 ...
```

Figure 2: Sample OAS schema excerpt for a Pet Store API.

```
1    POST     /pet             {}
2    POST     /pet/1/image     {img:img.png}
3    PATCH    /pet/1/image     {img:new_img.png}
4    GET      /pet/1/image     {}
```

Figure 3: Pseudocode of a sequence of HTTP requests comprising a REST API test case for a Pet Store API.

3 PUBLICATION SELECTION STRATEGY

This section reviews the scope of the survey, explains the paper collection and selection procedures, and provides an analysis of the surveyed papers.

3.1 Paper Collection Strategy

To identify relevant publications in the field, we first surveyed relevant conferences [3] and journals from the last 7 years. We also queried large scientific database search engines including Google Scholar, IEEE Xplore, and arXiv for combinations of keywords created by combining the tuples ⟨`REST API`, `RESTful API`, `REST`⟩ and ⟨`testing`, `test case generation`, `software testing`⟩ based on common keywords identified in previously discovered papers. We additionally used Google Scholar for the purpose of forward snowballing and consulted Related Work sections and bibliographies of selected publications.

[3]ICSE, ESEC/FSE, CCS, ISSTA, ICST, AST, EDOC, ICIT, and ICCSDET

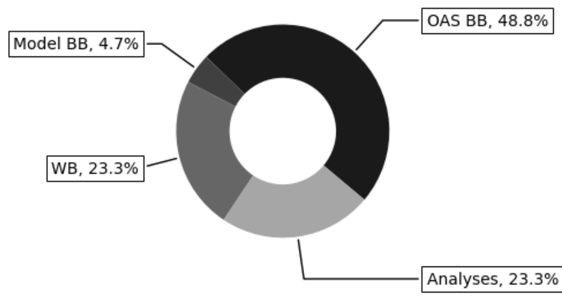

Figure 4: Distribution of surveyed work by the domain it adheres to.

We define our inclusion criteria in terms of four requirements with regard to the contributions of the paper. We include a paper in this survey if at least one of the following conditions is met:

(1) The paper introduces a novel approach or proposes an enhancement for an existing tool that generates RESTful API test cases.
(2) The paper compares existing RESTful API test case generation approaches either conceptually or empirically.
(3) The paper applies a tool from a previous publication to a real-life problem and reports meaningful results with practical implications.

3.2 Paper Distribution

To provide a better understanding of the research landscape, we provide a visual representation of the distribution of papers in Figure 4, classifying research by the categories that shape the structure of this survey. This categorisation shows that just over half of the work (53.3%) has focused on black-box approaches, with the vast majority of this work (48.8% of all papers) relying on the OAS alone as the guiding input for test case generation. Just over 23% of papers focused on white-box testing of REST APIs, which entirely consists of `EvoMaster` and evolutionary algorithms within this framework. An equal proportion of papers are analyses, empirical studies, or industry application reports. This distribution highlights the popularity of black-box approaches, which we attribute to their high degree of flexibility and lack of strict requirements imposed on the underlying software.

4 BLACK-BOX REST API TESTING

The black-box testing paradigm has been the focal point of the majority of RESTful API test case generation research, thanks to its flexibility and weak assumptions about the system under test (SUT). We conceptually identify two main categories of tools following this paradigm: specification-based approaches and model-based approaches. The former heavily rely on the OAS to provide derive the structure of REST API, while model-based tools additionally leverage more sophisticated, fine-grained models that convey more information than the OAS. The latter approach is the less common of the two, due to the strict requirements that a formal model imposes on the system and the development process.

4.1 Specification-based approaches

Chakrabarti and Kumar [10] propose one of the early attempts at automating the testing process of REST APIs. Their tool, called `Test-the-REST` introduces a uniform test-specification language based on XML, that automates the tasks of validating and running the test cases, agnostic to the underlying API's implementation. This approach is shown to be applicable to both manually written and automatically generated test cases, although the authors do not mention the underlying approach used for automated generation. Despite it only covering part of the testing pipeline, the authors show that there are significant productivity benefits to this task to warrant involved algorithms tailored to REST APIs.

Ed-douibi et al. [15] propose a technique that extracts system-specific representations from an overarching OAS metamodel, which enables the automated generation and reusability of test cases and input data. Their pipeline follows four steps: first, the API's specification is parsed, extracting a model that defines input-output behavior. This model is then enriched by embedding inferred parameters that guide the data generation process. Individual test case models emerge from the previous step, which are finally instantiated and converted into executable code. The generation process aims to trigger both nominal and faulty execution traces in the SUT, with tailored sampling processes for either scenario. The analysis of the schema offers a crucial advantage in that default and example values found in the specification are effective samples for required input data. Further, the output of some actions can be reused as input for others, providing an additional channel of information. Later work formalizes this latter technique and shows that dependency analysis between commands is an effective sampling mechanism for input data [8, 20].

Segura et al. [43] adapt the framework of metamorphic testing to RESTful APIs by defining an abstraction hierarchy based on metamorphic relation output patterns (MROPs). In general, metamorphic testing provides a foundation for testing programs whose output may be difficult to determine. Metamorphic relations are properties that input-output pairs that exercise the SUT must uphold. Such relations enable powerful testing techniques by defining rules that the system's behavior must invariantly respect under certain conditions. These relations implicitly specify how one can transform input objects to produce additional tests based on previously generated data. MROPs are domain-agnostic higher-level abstractions that define general output behavior typically found in web APIs, and consist of manipulating (i.e., adding, removing, mutating) input resources. The paper identifies 6 such abstractions, each of which can instantiate several metamorphic relations for an API under test, which can in turn be used to generate concrete test cases. These abstraction levels are helpful in guiding testers toward creating structurally meaningful metamorphic tests, and auxiliary data generation strategies can automate the implementation of concrete test cases stemming from the inferred hierarchies.

`RESTler` [8] is a tool that enhances lightweight analysis of the system's OAS with two heuristic additions that increase the efficiency of fuzzing the target API. The tool proceeds in iterations that produce increasingly lengthier sequences of requests used to fuzz the SUT. Each round of the algorithm produces new sequences

by appending one HTTP request to a previous test case. This technique can be either stochastic or exhaustive. Consumer-producer relations are the base model for dependencies between requests. This model is implemented as follows: when attempting to extend a sequence with an additional request that requires the existence of a resource (the consumer), `RESTler` only "accepts" this extension if a previous request (the producer) in the sequence creates such a resource. By having producers precede consumers in all sequences, it is less likely that the test would fail due to mishandling of the system's state. The second novel contribution is dynamic feedback, a strategy that discards sequences that fail (`5XX` return codes) from future iterations of the algorithm, as to avoid spending the computational budget on test cases that are guaranteed to crash the system. With its fuzzing-centered approach, `RESTler` does not exploit any model knowledge of the SUT, but it does allow for domain-specific parameters, such as pre-defined dictionaries for sampling variables.

`Pythia` [7] uses a statistical model to infer recurring usage patterns of REST API input data with the goal of generating more powerful mutation operators for fuzzing the SUT. To realize its framework, `Pythia` employs a three-stage pipeline. First, test cases are provided as input, either by capturing network traffic or by using the output of a standalone fuzzer, such as `RESTler` [8]. The tool then utilizes a user-provided grammar to parse the input test cases into abstract syntax tree (AST) representations. The second phase exploits the derived ASTs by training a `seq2seq` [46] autoencoder to learn the common structure of the underlying test cases. A mutation engine utilizes the trained autoencoder by injecting minimal noise between the encoding and the decoding phases to perturb the output, while still producing valid test cases. The location of perturbation in the output (if any) determines the mutation strategy applied to the target test case. An empirical evaluation of 7 real-world REST APIs finds that the novel implementations of `Pythia` outperform `RESTler` both in terms of structural coverage and in the number of defects uncovered.

Karlsson et al. [24] propose a property-based testing (PBT) approach called `QuickREST`. The aim of PBT is to generate input data in a fuzzer-like fashion to check the validity of pre-defined SUT's properties (also called invariants), with the additional goal of finding the simplest possible input that causes the program to violate its invariants, if any. For REST APIs, OAS serves as the basis for deriving such properties. In particular, the invariant for each path in the documentation requires that all generated test cases (i) produce return codes different than `5XX`, (ii) produce return codes in accordance with the specification, and (iii) produce responses whose payloads conform to the specification. Randomly generated input that either finds or successfully creates resources in the underlying system is cached and reused for requests that require prior state alterations, as to amortize the search procedure. Initial test data generation is entirely random, though `QuickREST` supports domain-appropriate alternatives, which can significantly improve performance compared to fuzzing.

`RestTestGen` [13, 48] is a tool that heuristically enhances the analysis of the OAS by enabling the reuse shared request data. An Operation Dependency Graph (ODG) serves as the model which captures a hierarchical dependency structure between the available operations of the SUT. Nodes in the graph represent operations specified by the OAS, and directed edges containing variable name annotations determine the order in which to target operations. An edge is drawn between two nodes when the output of one operation contains an identifier whose name is similar to the input to a different node. As the API outputs values during testing, they may be cached in a response dictionary and reused for testing upstream nodes, thus circumventing the effort required to randomly sample those values. `RestTestGen` generates test cases in a two-phase process. It first creates nominal test cases according to the ODG heuristic, testing the good-weather behavior of the API (using response code `2XX` as an oracle). Value generation for this stage is driven by random samples for both strings and integers, with a special bias for the empty string and `0`, respectively. Second, the nominal test cases are mutated in violation of the OAS schema, aiming to test the API's handling of invalid input, thus expecting `4XX` codes.

Godefroid et al. [20] focus on the problem of intelligently generating input data for REST API fuzzing. They process the OAS into several tree-formatted schemas, which encode the hierarchical representation of request bodies. Each schema (tree) corresponds to a specific request. Each node in the tree representation correlates with a property field, which is subject to several mutation operators during fuzzing. Mutations are applied both to individual nodes, as well as to entire trees and paths within trees. This approach is implemented in the `RESTler` tool [8] and experimentally validated on several cloud-based RESTful APIs. Their empirical analysis shows that a combination of mutation operators is beneficial, as is a search technique that stochastically balances the number of operators applied to each schema. The authors also notice an imbalance in the number of errors triggered in SUT (i.e., "easy" to find defects are triggered more often). Value rendering, a technique which caches and tags the payload body of API responses and uses a form of pattern matching to re-use the stored values as inputs, alleviates this problem. This technique closely resembles that employed in `RestTestGen` [48] to avoid re-sampling the same input values.

`RESTTest` [36] is a modular tool that introduces an extension to the OAS, that allows to better model the dependency structure of API operation parameters. The inter-parameter dependency language (IDL) [33] is based on the manual analysis of 2,500 operations from 40 real-world applications [34] and enables automatic reasoning about the relations between input parameters of a REST API via a reduction to the constraint satisfaction problem (CSP). `RESTTest` integrates an IDL reasoning module that allows it to leverage a lightweight interface for validating requests, and obtaining new (both valid and invalid) requests based on its analysis of the OAS. To generate test cases, `RESTTest` first constructs abstract test models based on the OAS, which are later transformed into abstract test cases through user-defined generation strategies that may include auxiliary information from the IDL. The tool implements support for both nominal and faulty test case generation. Oracles derive from response codes and the OAS, and are defined in accordance to the type of test case the algorithm seeks to generate. Finally, test cases are instantiated into executable code in either online or offline settings.

`Leif` [11] is a REST API fuzzer that attempts to reduce the large search space induced by weakly typed input parameters, while simultaneously bootstrapping the test case generation process. The conceptual cornerstone of `Leif` is the Format-encoded Type (FET)

lattice, a structure inspired by type lattices used in programming language design to determine type casting validity. In a REST API testing framework, FET lattices help determine the "true" types of input parameter values by either accepting or rejecting proposed type assignments. The lattice materializes under a tree-based model that undergoes a disambiguation merging process that iteratively alters the tree's topology based on local structure. Test case generation can exploit FET lattices by using their regular expression-encoded type definitions as dictionaries to exhaustively query for input data. `Leif` additionally accelerates the test generation process by capturing HTTP traffic concerning the SUT, which also provides for additional data for the lattice inference task.

`bBOXRT` [27] is a tool aimed at revealing robustness issues and security vulnerabilities in RESTful API-based systems. It employs a two-step approach that first exploits the OAS to generate tests that abide by the specified input constraints before attempting to alter these tests to trigger robustness-threatening behavior in the API. For the first step, `bBOXRT` uses a randomly-driven generation procedure called the workload generator that seeks to create input data that leads to specification-correct behavior in the SUT. Subsequently, the faultload generator applies type-specific mutations to the data generated by the first process. This two-phase approach is conceptually very similar to that of `RestTestGen` [48], however, the fault injection strategy of `bBOXRT` is more involved. In particular, type-appropriate mutations are defined (such as boundary manipulation for numeric inputs) and used in a fuzzing manner to inject faults in the previously generated data. Finally, faults triggered by executing the tests with the generated data are cached for both generation techniques, and the robustness weaknesses of the system are manually analyzed.

Mirabella et al. [39] introduce a deep learning-based framework for validating REST API input data based on the system's past behavior. This approach seeks to provide an oracle that can predict whether generated test input can successfully exercise the SUT without actually running the system. To do this, a neural network is trained on an encoded tabular representation of input data and its corresponding validity. Raw input data is enriched with (i) auxiliary variables that signal whether the input data is empty, (ii) normalization of numerical values that reduce the influence of high variation between different input parameters, and (iii) one-hot encoding of enumerator-based input parameters. The empirical analysis carried out on 8 industry APIs demonstrates the viability of the results, with the approach reaching an average test set accuracy of 97.3%.

Wu et al. [50] apply combinatorial testing (CT) in tandem with approximate constraint inference to increase the cost-effectiveness of test case generation for REST APIs, in a tool called `RestCT`. Their approach draws from the intuition that failures in REST APIs can often only be triggered if the associated input data satisfies a set of non-trivial constraints, commonly regarding input-parameter value combinations. To this end, `RestCT` seeks to identify hierarchical relations that emerge from OAS, as well as CRUD semantic properties. The tool uses natural language processing (NLP) techniques to exploit the NL channel of the OAS, seeking to identify members of a set of 23 dependency patterns specified by the authors. This catalog was extracted from the manual analysis of 40 real-world test APIs. The tool generates test input data using one of four procedures, either utilizing previously encountered values and specification information, or sampling values at random to increase diversity. To generate semantically sound tests, the sequence of requests in a test case must ensure that a resource is not accessed before creation or after deletion. This constraint enforcement technique is similar to the consumer-producer model employed in `RESTler` [8], however, `RestCT` imposes stronger conditions concerning the ordering of constraints, thus significantly truncating the search space. Also similar to `RESTler` is the generation mechanism, which appends constraint-satisfying commands to the end of sequences created in previous iterations. `RestCT` empirically outperforms `RESTler` on several open-source APIs.

`ARTE` [1] is a tool integrated in `RESTest` [36] that aims to reduce the size of the search space of input data generation by automatically producing realistic input data for REST API test cases through the use of knowledge bases. This technique is motivated by the limitations of dictionary- and stochasticity-based approaches. To address this issue, the authors implement a four stage pipeline. The input generation strategy begins with processing the OAS description of target parameters through standard NLP techniques to heuristically extract nouns that best match the name of the target parameter. Second, `ARTE` queries a provided knowledge base for predicates that best match the extracted nouns in the first step, according to six hierarchically ordered rules. The approach only uses predicates whose support, defined as the number of Resource Description Framework (RDF) triplets that contain it, exceeds a fixed threshold. Once extracted, the predicates can be used to generate the final input data for the test cases. To this end, `ARTE` builds queries that contain as many of the selected predicates as possible. However, because results for individual predicates may repeat between different query results, a relatively high fixed threshold of 100 is enforced for the number of total matching objects. The the constraints imposed on the query are iteratively eased until the number of obtained results exceeds this threshold. Finally, `ARTE` offers the optional feature of generating regular expressions that encapsulate the obtained input data. The results show that the generated data is both semantically and syntactically meaningful, with around 57% of input generated using this strategy resulting in valid API calls, and around 65% being semantically appropriate.

`Schemathesis` [23] is an adaptable tool that derives structure- and semantic- aware fuzzers for API-based systems. The tool exploits the API's OAS through the usage of `Hypothesis` [31], a state-of-the-art tool for PBT. Incorporating an external tool into the test case generation process allows `Schemathesis` to leverage advanced techniques while circumventing the implementation effort. `Hypothesis` seamlessly integrates into this workflow, as it allows the JSON representation of the OAS to serve as input for the creation of sophisticated data generation strategies without requiring a complex conversion process. To generate negative test cases, `Schemathesis` applies mutations to the OAS and again relies on `Hypothesis` to infer input generators for invalid excerpts of the OAS. The tool allows for extensive customization, such as modifiable test oracles and format specifiers for complex input data. The authors empirically compare `Schemathesis` against 8 alternative tools on 16 real-world APIs and find that their tool is the most effective at bug discovery, finding between 1.4× and 4.5× more defects than the second-best performing tool.

`Morest` [30] expands on `RestTestGen`'s ODG implementation by formulating a novel restful-service property graph (RPG). The RPG establishes the same consumer-producer relation between entities, however, it simultaneously considers both schema and operation nodes, and includes additional labels on its directed edges. The algorithm first builds an RPG from the OAS, and iteratively refines it as test execution progresses. `Morest` alters the graph's topology by exploiting misalignments between the OAS and the system's observed behavior. To this end, the graph receives additional edges if the system returns undocumented objects in responses, which could be leveraged in the subsequent requests. Graph edges may also be rendered infeasible if, after a pre-set number of tries, input obtained from other operations fails to produce a valid response. To generate test cases, argument-free request sequences are first created by recursively traversing the RPG's schema nodes. Heuristic rules are in place to reduce the number of misplaced requests, such as deleting resources before accessing them or accessing resources before creating them. Input data is generated for test cases either by (i) reusing previously valid input data, (ii) reusing the return values of previous requests as per the RPG, or (iii) sampling random values from the possible value range.

4.2 Model-based approaches

Pinheiro et al. [42] propone one of the first model-based approaches for REST API testing. Their tool employs a UML-based representation of Protocol State Machine models to derive coverage criteria in terms of states and transitions. The state machine model is appropriate for this task because its detail granularity suffices to accurately describe the behavior of the SUT, while simultaneously providing a measurable qualitative assessment for generated test suites. The model's states are enriched with additional information, including state invariants and composite states, which include nested state machines. Transitions between states are modeled by elementary `POST`, `PUT`, and `DELETE` requests that are only equipped with an adjacent resource. Crucially, this means that the task of finding appropriate additional parameters for these transitional request is left to the tester. To leverage the state machine model, the authors propose an approach that heuristically expands a directed acyclic graph (DAG) based on the model, where state invariants of the behavioral model are inherited in the generated graph. The DAG is then used as the basis for two coverage criteria: state coverage of nodes in the graph and transition coverage by means of traversing the graph in a recursive, node-invariant abiding manner, such that all possible transitions are greedily expanded for each node.

Fertig and Braun [17] formulate a test case generation tool that specifically targets systems built using Model Driven Software Engineering (MDSE). Their tool makes heavy use of formal models to derive test case specifications that are then filled using data generators. The framework additionally specifies one template for each possible HTTP action, such that requests that are heavily reliant on data being already available at the endpoint prior to invocation (such as `POST` and `PUT`) can be tested appropriately. The formal model of the SUT is exploited for generating data for the HTTP requests by, for instance, defining truncated domains for path variables adjacent to an endpoint. This property is exceptionally useful because removes large portions of the search space, thus making the test case generation more efficient. However, a drawback of this approach is that it is only viable when detailed and accurate formal models are available for the SUT, making generalizability a challenge.

5 WHITE-BOX REST API TESTING

EvoMaster [2] is a tool for generating system-level tests for REST API applications implemented in JVM-based languages. Originally designed for white-box scenarios, EvoMaster exploits search-based techniques to iteratively improve a set of individual test cases. The modular architecture of the framework allows for the implementation of different evolutionary algorithms that serve as the engine for the test case generation process. In addition to exploiting the OAS of the system, the key difference in comparison to black-box counterparts is that EvoMaster assumes the entire source code base of the SUT is available, and as such can be instrumented and manipulated to provide more concrete testing targets. By default, EvoMaster defines coverage targets of three types: statement coverage, branch coverage, and HTTP status code coverage in accordance to the OAS. In addition to standard techniques of "smoothing" the search space, EvoMaster introduces a measure of distance that disincentivizes tests that result in exceptions on the path to a specific target. `EvoMaster` uses smart sampling, a technique tjat, once triggered for a request, prepends that request with a sequence of other requests sampled according to pre-defined rules, which aim to bring the system to an appropriate state.

Much the research effort in the domain of white-box REST API testing has been centered around the EvoMaster framework, namely by proposing increasingly performant algorithms that can be inserted as modules for test case generation. Section 5.1 surveys such algorithms, while Section 5.2 focuses on futher refinements of EvoMaster.

5.1 Evolutionary Algorithms for White-Box REST API testing

One of the first works to extend the notions of search-based software testing to the object-oriented (OO) domain and introduce the notion of dynamic targets (i.e., selecting different objective functions from the SUT) is that of Tonella [47]. Though Genetic Algorithms had been used for the test case generation problem previously (i.e., Sthamer [45] demonstrated the advantages of GAs in comparison to random testing; Michael and McGraw [38] investigated the challenges of scaling up search-based test case generation to complex systems; Pargas et. al [41] introduced dynamic target selection to GA-based test generation), these solutions were built for the procedural programming paradigm. The adaptations proposed by Tonella enabled GA-based algorithms to be applied to object-oriented software, an application that grew tremendously in importance as the OO paradigm gained popularity in the following decades.

Regarding its application to REST APIs, the relevance of the approach resides in the way it targets objectives. The algorithm, representative of a class called *single-target approaches*, first randomly selects an unreached target (i.e., a branch) before evolving the pool of test cases (the population) using a heuristic (i.e., branch distance) to approach it. A new target is selected either when a test

in the current population reaches the previous one or after a predetermined number of generations. This method suffers from two key drawbacks: (1) the computational budget is uniformly distributed among the targets, meaning that infeasible targets may significantly slow down the algorithm, and (2) inadvertently reached targets are not effectively exploited. For instance, a test A in the population reaches an uncovered target X but not the current objective, and another test B from the same population reaches the objective target but not X, only test B is kept in the archive, and target X will be optimized for in a future generation. Pargas et al. [41] previously noted this drawback, which was later addressed by different formulations.

To mitigate the shortcomings of the single-target strategy, the Whole Test Suite (WTS) [19] algorithm institutes an approach where instead of test cases, the search space is composed of test suites. Accordingly, the optimization is single-objective and tracks a scalarized version of the many-objective approach. This transformation from the many-objective space to the single-objective scalarization can be thought of as a combination of two techniques used in evolutionary computation: weighted Tchebycheff utility functions and hypervolume indicators [56]. The WTS approach uses a weighted aggregation-based objective function formulation but behaves similarly to hypervolume indicators in that the function is used to evolve the collection of underlying tests all at once. The advantages of this formulation are (1) that infeasible targets are no longer individually optimized for, and (2) that once covered, a target will not be re-optimized for in a separate, independent test that seeks to reach, for instance, a nested branch. In its implementation, WTS also includes a penalty for uncovered methods to increase selection pressure.

WTS employs a population of sets of test cases. Each set may contain up to N tests, which can be up to L statements long. Both recombination and mutation are supported at the set level. Recombination follows a one-point crossover pattern, where subsets of tests are "swapped" between parents to produce offspring. Mutation is performed by stochastically altering the tests in a member of the population, by either removing, changing, or inserting a new statement. Random test cases are sampled to generate the initial population, by repeatedly using the insertion-based mutation probabilistically.

The Many Independent Objective (MIO) algorithm [3] is a tailored evolutionary algorithm that aims to improve the scalability of automated test case generation for software with very high number of individual test targets. MIO introduces a novel population modelling concept which works in tandem a tailored sampling strategy to alleviate the weaknesses of the single-target approach. In MIO, each target (assumed to be indepedently optimizable) is granted a unique population of a bounded size. Whenever a new test is sampled, it is assigned a heuristic value for each of the (yet unreached) test targets, based on how close it is to reaching that target. If the sampled test is better than that population's worst member, it replaces the weaker counterpart in that target's population. If the sampled test reaches a previously unreached target, that objective is dropped from the optimization function.

The sampling procedure used in MIO functions through two channels: random generation or mutation of existing members. To complement the dynamic population system, MIO introduces Feedback-Directed Sampling (FDS) for the population sampling channel. FDS is a mechanism that seeks to bias the exploratory process toward targets with a higher chance of being reachable as opposed to spreading the exploration budget uniformly. In FDS, each population is assigned a counter that is incremented each time it is sampled for mutation purposes, and reset whenever a better member is added to it. Populations with lower counters (i.e., that have recently been improved) are favored.

The Many-Objective Sorting Algorithm (MOSA) [40] provides an alternative formulation of the test case generation problem. It treats all (not yet covered) targets in the SUT as distinct and independently targetable objectives, to be optimized simultaneously. This method is motivated by the inefficiency of scalarization-based methods for some common classes of problems [14], and the effectiveness of many-objective solutions in complex problem settings [22, 26]. In MOSA, each branch in the target program underlies an objective to be minimized, composed of the sum of the minimum normalized branch distance (computed in the same way as in WTS [19]) and the approach level. The search space is composed of individual test cases. In addition, the authors introduce the notion of *preference concerning a target*: a test is preferred over another if the former's objective evaluation (i.e., normalized branch distance + approach level) for that objective is smaller than the latter's.

In the test case generation setting, certain general concepts from numerical many-objective optimization, such as thew goal of generating a set of non-dominated solutions that capture trade-offs, do not align well. In addition, since the number of targets in real-world systems may reach thousands, and the number of solutions in Pareto optimal sets increases exponentially in the number of objectives [49], a mechanism that selects individuals that reach optimality for as many objectives as possible is necessary. To do this, MOSA uses *preference sorting* to distinguish between pairs of non-dominated individuals. In particular, this mechanism separates the test(s) with the best objective function for each uncovered branch. MOSA then inserts these tests in the next generation's population before selecting from the remainder of the non-dominated tests using the standard non-domination sorting routine. This increases the selection pressure by preferring the best test cases with respect to at least one target. Finally, the selection procedure uses crowding distance as a metric to increase the probability of selecting a diverse set of test cases.

LT-MOSA [44] is an extension of MOSA [40] that targets REST API test case generation by leveraging Linkage Structure learning. More specifically, groups of genes commonly present in "good" members of the population are statistically identified in an above-average subset of individuals and used in the creation of subsequent populations. Instead of representing test cases in the standard chromosomal way used for the core search procedure, a novel constrained, fixed-length representation linkage encoding is used solely for the learning task. This encoding uses an ordered list of binary variables that each represent the presence of a particular HTTP action in a given individual.

To build the linkage tree model, a greedy algorithm successively merges groups of variables based on the unweighted pair group method with arithmetic mean (UPGMA). The model is built using the individuals in the first non-dominated front. At each iteration, the algorithm computes the distance between every possible group

of genes based on the mutual information computation (i.e., the difference between the sum of individual information entropies and the joint information entropy). The hierarchical aspect of this solution refers to how variables are clustered together: the greedy algorithm starts from a set of singleton groups of genes before merging two groups at each iteration. The information captured by the LT model is leveraged through MOSA's recombination operator. Two individuals, a parent and a donor are sampled from the population by means of tournament selection. A single offspring emerges by first copying all the genes from the parent and then possibly injecting additional genes from the donor. Subsets that are part of the learned model and which manifest in the donor's chromosomal representation are sampled at random and injected into the generated offspring at a random point. If no such subset is present in the donor, the standard one-point crossover operator is employed instead. Finally, LT-MOSA performs mutation stochastically, with a linearly increasing probability that is less perturbative in the early generations and more so in later ones.

5.2 Heuristics Enhancements for White-Box REST API testing

Arcuri and Galeotti [5] focus on the interaction between APIs and SUT's database to improve test case generation. To provide more informed guidance, they introduce a driver-level SQL monitor that captures the system's outgoing SQL commands. To complement this information, the notion of SQL distance is introduced for SELECT queries, as an analogous alternative to the branch distance heuristic used at the code level. This metric heuristically computes how "close" a query is to matching data out of a sampled subset of rows in the database. The SQL distance is implemented as a secondary dimension of the fitness function used in the evolutionary algorithm of `EvoMaster`, as a way of discriminating between individuals with equal values along the first (structural coverage) dimension. To compute this value, 3 formulas are established, that use different heuristic biases to determine a fitness value for a given test case. The authors further allow for the direct insertion of data into SQL databases, which circumvents the necessity of sampling an unlikely sequence of requests for this purpose. Additional focus is placed on automatically handling constraints imposed by SQL's CHECK semantics, and regular expressions required by LIKE and SIMILAR queries. An empirical study carried out on six REST APIs shows that the novel techniques can improve the coverage of `EvoMaster` by up to 16.5% and discover previously unfound bugs, but they can also adversely impact test bloat by generating larger tests.

Zhang et al. [54, 55] focus on improving resource-centric knowledge exploitation in `EvoMaster`. The backbone of this approach consists of 10 semantically meaningful resource-based templates used to generate new test cases during search. These templates enable resource-based sampling, a technique that relies on four sampling methods equipped with necessary preconditions. Both the methods themselves and the preconditions account for dependency relations between resources and templates, respectively. To select which method to use, the proposed approach utilizes five sampling strategies, that each focus on different factors, including the structure of the SUT's endpoints, the remaining search budget, and previously sampled methods. An adjacent resource-based mutation operator is established to operate within the confines of the template sampling conditions. To further enhance sampling and mutation, a heuristic dependency handler is defined. Dependencies are identified statically from the OAS based on the names that developers assign to resources and parameters, and from SQL tables. Runtime fitness feedback is also considered, and dependencies are inferred based on the effect of past resource-based mutations. A graphical model represents the dependency structure of the SUT, and individual dependencies exhibit an additional confidence parameter that is adapted during search, and represents the approximate likelihood of the dependency being "true". Complementary sampling and mutation operators emerge from dependency analysis, which materializes in the implementation of a parameter that controls the probability of sampling linked resources, when appropriate. An empirical analysis of 7 open-source REST APIs and 12 synthetic APIs reveals that the novel additions improve the overall performance of the tool in 17 of the 19 total benchmark instances and that both resource-based sampling and dependency analysis are individually valuable extensions.

Arcuri and Galeotti [6] establish a list of testability transformations aimed at improving the guidance of fitness functions used in `EvoMaster`. Testability transformations are implemented bytecode manipulation techniques that replace existing methods with alternatives that provide more fine-grained guidance. The authors introduce four categories of such manipulations, to address different ways in which traditional fitness functions may reach a so-called plateau. The flag problem, a phenomenon that often occurs when fitness objectives are binary, is addressed by introducing heuristic distance metrics for 11 boolean class methods. This information is further exploited through input tracking, a technique that observes when input variables are directly used by replaced methods. This technique enables a more effective, context-appropriate, mutation operator to be leveraged on such input. An additional transformation is defined to tackle likely causes of incompleteness in OAS schemas. This transformation tracks specific objects that automatic schema generation tools may fail to exploit, in such a way as to derive optional genes that target the newly uncovered query parameters or their types. Finally, a TCP-specific transformation is applied to keep connections to the SUT alive for longer. The authors evaluate their additions to 3 toy systems, 9 real-world REST API services, and 1 industrial web service. The empirical study demonstrates the effectiveness of the testability transformations, with an increase of up to 10× better line coverage for the industrial API, and a statistically significant improvement in complex open-source systems.

Zhang and Arcuri [51] introduce a weight-based adaptation mechanism for mutation rates during evolutionary search. Weights represent a balancing device that aims to proportionately adjust the mutation probability of genes with respect to the size of the represented underlying object. In practice, genes that correspond to larger objects (such as arrays) receive a high probability of undergoing mutation at each iteration than smaller ones (like booleans). To complement this mechanism, the rate of mutation (i.e., the expected number of mutations suffered by an individual in each iteration) can also be appropriately adapted. To this end, the authors propose a formula that adjusts the mutation rate by a factor proportional

to the weight of a gene in relation to a selected gene superset. The proposed mutation framework additionally takes into account the impact of each performed mutation. The impact of a gene is flexibly defined in terms of the number of times a mutation of that gene changed the fitness of an individual for a testing target and the number of times in which it has not done so. The derived weights can be implemented in the standard hypermutation formula to derive adaptable probabilities. Adaptive gene selection and adaptive gene value mutation are derived using impact-based weights and married together in a mechanism called adaptive weight-based hypermutation (AWBH). The overarching AWBH framework enables the definition of all of the operators required to define general-purpose mutation functionality, which can be modularly connected to existing EAs. The performance of the novel mutation operator is assessed in an empirical analysis over 7 open-source REST APIs. Adaptive mutation is found to improve coverage in 18 considered assessment criteria and decrease performance in 9. Weight-based mutation contrastingly improves coverage in 19 cases and only worsens performance in 2. The combination of both techniques is shown to significantly improve performance over the baseline in all but two assessed criteria.

6 APPLICATIONS AND ANALYSES

Martin-Lopez et al. [35] present a list of 10 test coverage criteria that aim at offering a comprehensive assessment of the quality of a given test suite. The proposed criteria cover both the API's request and response behavior, and referred to as input and output coverage, respectively. The authors categorise the coverage measures into a Test Coverage Model (TCM) composed of 7 Test Coverage Levels (TCLs), with the goal of establishing a unifying framework for test suite assessment. TCLs are hierarchically ranked on the basis of the strength of the criteria they correspond to. In this hierarchy, all weaker TCLs' requirements (where level 0 is the weakest) must be fulfilled for a particular test suite to "progress" in the hierarchy. This distinction is, however, different from the subsumption-based ranking of traditional coverage criteria, meaning that the criteria corresponding to higher level TCLs do not necessarily subsume those of lower level TCLs. The authors evaluate the potential of the TCM on test suites generated using `EvoMaster` [3] on 2 REST APIs and find that the hierarchical structure of TCLs suitably represents the quality of test suites. Concretely, test suites that minimally meet the requirements of lower TCLs generally achieve lower coverage and discover fewer errors than those that fulfill higher TCLs.

Arcuri [4] compares the performance of black- and white-box testing approaches by running `EvoMaster` [2] on 8 industrial APIs, to assess whether the additional overhead incurred by white-box approaches translates into higher coverage. MIO [3] is used as the driving algorithm for the white-box scenarios, while a random testing approach drives the black-box strategy. The quality of the resulting test suites is compared in terms of size, return code and code coverage, as well as the number of faults found. The results show that the additional information leveraged by white-box approaches is in fact effective in comparison to random testing, as the white-box mode outperforms its counterpart in almost every coverage criterion. There are only 3 instances of return code coverage where the black-box strategy equals its counterpart. In contrast, the black-box approach produces significantly smaller test suites, with an average length an order of magnitude smaller than the white-box approach in 6 out of 8 cases.

Godefroid et al. [21] apply automated REST API test case generation to the task of differential regression testing, which aims at finding regressions, or breaking changes, between different versions of REST APIs, both in the API specification, as well as in the underlying service itself. For this task, `RESTler` [8] to generate test cases based on the version-specific API specification of the services, and process the results using differential testing to detect potential descrepencies across versions. The authors carry out an empirical analysis on 17 versions of Azure networking APIs and detect a total 14 previously undiscovered breaking changes that were later fixed in subsequent deployments.

Corradini et al. [12] compare `RestTestGen` [48], `RESTler` [8], `bBXORT` [27], and `RESTest` [36] in terms of robustness and test coverage on 14 real-world REST services. Their results suggest `RESTler` is the most robust tool, and the only on that is able to generate tests for all benchmark instances. Test coverage analysis indicates `RestTestGen` is performs the best in terms of 5 coverag criteria, while `bBOXRT` and `RESTler` each perform best in one criterion. `RESTest` is the weakest performer in both categories, only supporting 2 of the 14 REST APIs and never providing the highest coverage.

Martin-Lopez et al. [32] investigate the differences between black-box and white-box testing in an empirical analysis on 4 real-world systems. RESTest [36] is used as the BB example and EvoMaster [2] using the default MIO [3] search algorithm is the WB counterpart. To measure the performance of the tools, the authors measure both the achieved branch coverage of each resulting test suite, as well as the number of faults detected (i.e., the number of endpoints returning at least one `5XX` status code). Their results show that the BB approach performed better than its WB counterpart with statistical evidence in 2 out of the 4 instances. The authors suggest that the complex and rule-based nature of the underlying system is the reason for this discrepancy. In particular, some of the tested systems require highly structured input for requests to be valid, such as small subsets of strings (i.e., `en-US` for a language parameter). Random sampling-based input generation tools such as EvoMaster are unlikely to match such strings. In contrast, RESTest employs test data generators for this task, which greatly improves the probability of sampling acceptable requests. The authors additionally propose a novel approach that builds a pipeline based on both the BB and WB approaches. This system works as follows: first, `RESTest` generates a test suite in its standard configuration, with only a fraction of the total time budget. The test suite is then written to an intermediary format before being parsed and passed on to `EvoMaster` in the appropriate embedding. Finally, `EvoMaster` uses the received test cases as the initial population, which it evolves using a GA, for the remainder of the time budget. The final solution is the elitist archive which the underlying GA of `EvoMaster` returns. Analysis shows that the hybrid approach outperforms `EvoMaster` alone by up to 60% in all instances, and `RESTest` alone by between 1.1% and 4.8% for 3 out of the 4 systems, suggesting that the combination of approaches is more powerful

than either of them in isolation. The analysis of the achieved coverage over time also shows that the BB approach converges much faster than the WB counterpart.

Martin-Lopez et al. [37] use RESTest [36] to empirically test the effectiveness of automatic test case generation on 13 industrial REST APIs. To assess the practicality of using automatic test cases generation for production-ready and web-scaled systems, the authors measure several metrics regarding failure and fault detection rates and API coverage in relation to the specification. After generating over 1 million test cases over a 15-day period, the results revealed a total of 147 faults in 11 APIs, with specification disconformities causing around half, and accepting invalid API inputs causing around one third. The experiments also show that different data generation techniques (i.e., fuzzing, data perturbation, and random inputs) are complementary and capable of detecting faults. In total, 248 bugs were detected throughout the experiment, 65 of which were fixed by developers of the respective systems.

Kim et al. [25] carry out an analysis of 8 research tools and 2 open-source practitioner's tools for REST API testing over 20 real-world systems with the goal of analyzing the code coverage and error discovery capabilities. The analyzed tools are bBOXRT [27], EvoMaster [2], RESTest [36], RESTler [8], RestTestGen [48], Schemathesis [23], Dredd, APIFuzzer, and Tcases [4]. In a black-box setting, EvoMaster obtains the best coverage by all three considered criteria, and none of the available tools exceed 50% coverage. Tcases, EvoMaster, and Schemathesis are on average able to trigger the largest number of 5XX response codes, with Tcases being the highest. In a white-box environment, EvoMaster drastically outperforms its counterparts both in terms of coverage and the number of errors uncovered. The authors attribute this significant gap to the coverage-driven guidance that white-box scenarios enable, that allows EvoMaster to generate better quality input parameters. Another avenue for improvement is the ability of tools to generate so-called stateful tests by better taking into account the producer-consumer relationships between operations of the SUT.

Zhang and Arcuri [52] empirically compare 7 state-of-the-art fuzzers in both black- and white-box settings on a total of 18 open-source and 1 industrial API. The analysis includes bBOXRT [27], EvoMaster [2], RESTest [36], RestCT [50], RESTler [8], RestTestGen [48], and Schemathesis [23]. The black-box experiments reveal that EvoMaster and Schemathesis provide the best coverage in all APIs but 1, with EvoMaster performing best in 11 cases, and Schemathesis in 7. bBOXRT and RestTestGen are comparable, but fall behind, while the remainder tools produce lower coverage. White-box experiments are conducted using EvoMaster, and a statistically significant increase in code coverage is obtained in 15 out of the 19 instances. An accompanying increase in the number of detected faults is also observed, however, in many cases, the coverage does not exceed 50%. Both sets of results are consistent with those of Kim et al. [25]. The authors conclude by identifying several aspects that require attention when designing a REST API test case generation tool, including robustness, schema fault tolerance, database effect integration, and mocking external services.

Zhang et al. [53] apply EvoMaster at Meituan, with the goal of assessing the quality of integrating research-oriented tools on scale-sensitive tools in industry. The results show that EvoMaster is able to generate both meaningful coverage of between 10.9 and 79.8%, as well as reveal previously unknown bugs. Overall, the tool achieved an average of 51.7% and 71.3% coverage of business logic and endpoint components of the system, respectively, while also uncovering 21 real faults. The authors reflect on 10 challenges that the experiment revealed, which they recommend future research efforts focus on.

[4]TODO: citations

7 CHALLENGES AND OPPORTUNITIES

Despite a significant increase research efforts in recent years, the area of REST API testing is still in its early stages. This section summarizes the challenges facing future research in this area and the opportunities that arise as a consequence.

7.1 Challenges

Several hard challenges remain unsolved in REST API testing, both domain-specific, as well as more general in nature.

Generating meaningful stateful sequences. For test cases to exercise complex functionality within the API, resources need to be adjusted accordingly before the invocation of certain methods. We refer to the process of creating, modifying or otherwise altering the state of resources with the goal of invoking further requests with specific semantics as the *stateful sequence problem*. Intuitively, this materializes in a the necessity of creating a resource before performing a GET request on it, or, by contrast, not accessing a resource anymore after its deletion. In practice, tools rely on heuristic approaches to address this problem, by either stochastically inserting requests that seek to bring the system into the right state, or shaping the sampling space to increase the probability of meaningful stateful sequences being instantiated.

We hypothesize that relatively simple solutions like the ones implemented in current state-of-the-art tools might fail to scale in complex scenarios, where requests trigger complex state transitions in SUT. Complex input dependencies and high independent traffic in online testing might further aggravate the problem, which necessitates more sophisticated solutions. However, no available research directly analyze such strategies, so the assessing their strengths and weaknesses can only be done analytically.

Reasoning about parameter interdependence. Automatically generating semantically meaningful test cases requires a logical flow of information between the sequence of HTTP requests. In practice, this means that often, requests have to share identical parameters to trigger complex behavior in the SUT. Relying on random generators to solve this problem is a suboptimal strategy due to the low likelihood of generating identical inputs multiple times in large search spaces. To mitigate this, several approaches have been proposed, which base themselves on one of three principles: either (i) reusing old values which have resulted in valid test cases, (ii) utilizing predefined dictionaries or examples provided in the OAS, or (iii) taking advantage of values returned by, or used in other operations. The latter approach shows the most promise, as it is able to leverage not only static OAS information, but also dynamic snippets that are either randomly generated or produced by the system.

Recent solutions based on NLP in RestCT [39] and property graphs in Morest [30] have shown promising results in this area, demonstrating the potential for improvement over more standard approaches like RestTestGen's consumer-producer model. Such approaches are complementary: while NLP-based techniques may be able to better leverage the information exhibited in the OAS, graph-based solutions are more flexible and can be adjusted on-the-fly according to the system's feedback. We note that interdependence problems can be easily reasoned about in scenarios where formal models are available for the API under test. However, the lack of popularity of model-based REST API testing suggests that the cost of this requirement is likely too high for practitioners, and that further pursuing current avenues of research might prove more fruitful.

Generating input data. Stringent requirements on input data formats and the lack of formal specifications in the OAS makes generating input data for certain fields highly unlikely if done randomly. Search-based methods may break down in such situations due to extremely large search spaces for which very few candidates are valid (for instance, the search space of strings that are also country codes). Recent advances have emerged from both machine learning (ML) techniques like ARTE [1] and constraint satisfaction strategies like Leif [11]. Such approaches exceed the performance of standard approaches like pre-defined dictionaries, motivating further research in this area. Though results are promising, these techniques have not been validated on large numbers of systems and their generalizability is unproven.

Reliable and accurate oracles. The oracle problem is well-established in most branches of software testing, and is especially evident in the case of RESTful APIs. Most approaches define oracles in terms of return codes and adherence to the OAS specification. However, such approaches fail to take into account situations in which the OAS may be incomplete or misaligned, and there may be potential for deriving online oracles from the API's behavior over time.

7.2 Opportunities

In spite of facing numerous challenges, REST API testing also presents opportunities that may significantly enhance the performance of state-of-the-art tools.

Intersectional opportunities between black- and white-box approaches. Previous work has shown that the code-level structural guidance that white-box approaches benefit from makes them more generally more capable than black-box approaches [25, 53]. However, literature still contains cases in which specific black-box approaches are able to outperform their white-box counterparts [32], and that a combination of the two may provide better coverage than either could individually. To the best of our knowledge, combining the two techniques has only been explored in one study [32], and many potential combinations of the two strategies remain unexplored. Intertwining black-box techniques with white-box guidance may provide better insight into how to effectively improve the state of the art.

Opportunities in data-driven approaches. Data-driven ML approaches have been proven effective for both input data generation [1, 7], input validation [39], and interdependency discovery [50]. However, the scope of the on which the models are based is relatively modest. For instance, RestCT's catalog of dependency patterns is composed from only 40 real-world OAS schemas. Exploring data mining solutions that would allow to collect, process, and leverage larger amounts of data could improve both the performance and the generalizability of existing techniques.

Additional fields for data-driven exploitation could also be studied. Chen et al. [11] fuzz reused API traffic to amortize test case generation time. Moreover, algorithms like MIO [3] re-sample existing test cases before mutating them. However, there are no studies in the current literature that assess the relative amortization that results from reusing captured traffic, and no qualitative comparison of generated and reused data exists. Because tools that generate both input data and request sequences from scratch simultaneously perform two search tasks, analyzing the benefit of reusing previous (input-free) request sequences may boost the performance of current approaches by enabling them to better focus on the input data generation problem.

Providing more detailed guidance for black-box approaches. Black-box approaches trade off the guidance available through access to the system's source code for the versatility of not being subject to language-specific requirements like instrumentation. Model-based approaches provide better guidance through the inclusion of formal models, which severely limits their use cases in the real world. The drawback of lacking guidance may be partly ameliorated by constructing an approximate and coarse-grained model driven by the interaction with the API. The level of this model may range depending on the application, and driving factors could vary from API responses that help determine the underlying system's state, to logs the system produces during execution. Such approaches could provide an appealing trade-off between the rigidity and accuracy of formal model algorithms and the flexibility of OAS-based solutions.

Standardized data sets. Previous empirical analyses of REST API testing tools have focused on relatively small data sets. To the best of our knowledge, the empirical experiments of Corradini et al. [12], Kim et al. [25], and Zhang and Arcuri [53] contain the largest datasets assembled for assessing REST API testing tools. These papers utilize data sets consisting of 14, 19, and 20 projects, respectively. Since data sets differ between evaluations, proposing a common corpus of projects to use for future evaluations would facilitate comparisons. Assembling larger data sets would also increase the robustness and validity of results, while potentially providing more insightful analysis.

8 CONCLUSION

This paper provided a comprehensive overview of research efforts focusing on automated test case generation for REST APIs. We introduced the necessary background and definitions used throughout literature and analysed the distribution of papers over time and by category. We surveyed available REST API testing tools that operate in black- and white-box approaches and highlighted real world

applications and analytical comparisons of recent work. We also analyzed the current state of the field, outlining its biggest challenges and suggesting appropriate opportunities for future work.

REFERENCES

[1] Juan Carlos Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* (2022).

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

[3] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.

[4] Andrea Arcuri. 2020. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software* 38, 3 (2020), 72–78.

[5] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31.

[6] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–34.

[7] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).

[8] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

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

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

[11] Yixiong Chen, Yang Yang, Zhanyao Lei, Mingyuan Xia, and Zhengwei Qi. 2021. Bootstrapping Automated Testing for RESTful Web Services. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 46–66.

[12] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for restful apis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 226–236.

[13] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability* (2022), e1808.

[14] Kalyanmoy Deb. 2014. Multi-objective optimization. In *Search methodologies*. Springer, 403–449.

[15] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In *2018 IEEE 22nd international enterprise distributed object computing conference (EDOC)*. IEEE, 181–190.

[16] Adeel Ehsan, Mohammed Ahmad ME Abuhaliqa, Cagatay Catal, and Deepti Mishra. 2022. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences* 12, 9 (2022), 4369.

[17] Tobias Fertig and Peter Braun. 2015. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*. 1497–1502.

[18] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

[19] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[20] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 725–736.

[21] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.

[22] Julia Handl, Simon C Lovell, and Joshua Knowles. 2008. Multiobjectivization by decomposition of scalar cost functions. In *International Conference on Parallel Problem Solving from Nature*. Springer, 31–40.

[23] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 345–346.

[24] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141.

[25] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. *arXiv preprint arXiv:2204.08348* (2022).

[26] Joshua D Knowles, Richard A Watson, and David W Corne. 2001. Reducing local optima in single-objective problems by multi-objectivization. In *International conference on evolutionary multi-criterion optimization*. Springer, 269–283.

[27] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738–24754.

[28] Valentina Lenarduzzi, Jeremy Daly, Antonio Martini, Sebastiano Panichella, and Damian Andrew Tamburri. 2020. Toward a technical debt conceptualization for serverless computing. *IEEE Software* 38, 1 (2020), 40–47.

[29] Valentina Lenarduzzi and Annibale Panichella. 2020. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software* 38, 1 (2020), 54–60.

[30] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. *arXiv preprint arXiv:2204.12148* (2022).

[31] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[32] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231–241.

[33] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* 15, 4 (2021), 2342–2355.

[34] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In *International Conference on Service-Oriented Computing*. Springer, 399–414.

[35] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 15–21.

[36] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In *International Conference on Service-Oriented Computing*. Springer, 459–475.

[37] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22). ACM.*

[38] Christoph Michael and Gary McGraw. 1998. Automated software test data generation for complex programs. In *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*. IEEE, 136–146.

[39] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep learning-based prediction of test input validity for restful apis. In *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, 9–16.

[40] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[41] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. 1999. Test-data generation using genetic algorithms. *Software testing, verification and reliability* 9, 4 (1999), 263–282.

[42] 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*. 1–10.

[43] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. In *Proceedings of the 40th International Conference on Software Engineering*. 882–882.

[44] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 117–128.

[45] Harmen-Hinrich Sthamer. 1995. *The automatic generation of software test data using genetic algorithms*. University of South Wales (United Kingdom).

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[47] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.

[48] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

[49] Christian Von Lücken, Benjamín Barán, and Carlos Brizuela. 2014. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational optimization and applications* 58, 3 (2014), 707–756.

[50] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[51] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–52.

[52] Man Zhang and Andrea Arcuri. 2022. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *arXiv preprint arXiv:2205.05325* (2022).

[53] Man Zhang, Andrea Arcuri, Yonggang Li, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2022. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. *arXiv preprint arXiv:2208.03988* (2022).

[54] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for restful web services. In *Proceedings of the genetic and evolutionary computation conference*. 1426–1434.

[55] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 1–61.

[56] Eckart Zitzler and Lothar Thiele. 1999. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE transactions on Evolutionary Computation* 3, 4 (1999), 257–271.