

QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs

Stefan Karlsson
ABB AB, Mälardalen University
Västerås, Sweden

Email: stefan.l.karlsson@{se.abb.com, mdh.se}

Adnan Čaušević
Mälardalen University
Västerås, Sweden

Email: adnan.causevic@mdh.se

Daniel Sundmark
Mälardalen University
Västerås, Sweden

Email: daniel.sundmark@mdh.se

Abstract—RESTful APIs are an increasingly common way to expose software systems functionality and it is therefore of high interest to find methods to automatically test and verify such APIs. To lower the barrier for industry adoption, such methods need to be straightforward to use with a low effort. This paper introduces a method to explore the behaviour of a RESTful API. This is done by using automatic property-based tests produced from OpenAPI documents that describe the REST API under test. We describe how this method creates artifacts that can be leveraged both as property-based test generators and as a source of validation for results (i.e., as test oracles). Experimental results, on both industrial and open source services, indicate how this approach is a low effort way of finding real faults. Furthermore, it supports building additional knowledge about the system under test by automatically exposing misalignment of specification and implementation. Since the tests are generated from the OpenAPI document this method automatically evolves test cases as the REST API evolves.

Index Terms—Property-based testing, OpenAPI, REST

I. INTRODUCTION

Representational state transfer (REST) is an architecture style, introduced by Fielding, that describes constraints on web services [1]. With REST, a system resource is exposed by a URI and created, read, updated and deleted with HTTP verbs. A service that uses the REST architecture is said to be *RESTful*. REST APIs are a common way of exposing web services on the internet. The web-site *Programmableweb*¹ contains more than 20,000 APIs in their directory². This includes well known services such as Twitter³, YouTube⁴, Facebook⁵ and cloud providers such as Microsoft Azure⁶ and Amazon Web Services⁷. REST APIs are also commonly used when exposing internal interfaces in a microservice architecture [2]. Test methods targeting REST APIs can thus be useful both on the internal and external interfaces of a software system.

The automatic exploration of RESTful APIs has the potential to save developer and tester effort, get insights from the system under test (SUT) and fill the gaps of lack of tests where such gaps exist. Recent results by Atlidakis et al. [3]

and Arcuri [4] have shown that fault finding can be done automatically for REST APIs, both as a black- or white-box approach. We think such approaches could help developers and testers in exploring functionalities of REST APIs, particularly if more properties are evaluated other than HTTP status codes.

In our experience, exploring functionalities of a SUT is mostly a manual effort in industry, and for good reasons. Some insights about a SUT need a human in the loop, such as evaluating if a usability pattern makes sense in the given domain. However, it may be valuable to automate the parts of exploration that can be done by machines, and in doing so freeing up human effort to be spent where it is of best use.

REST APIs are increasingly commonly described with OpenAPI [5], which aims to standardize how RESTful APIs are described. Several frameworks for building REST APIs also include OpenAPI support. The OpenAPI document specifying REST APIs opens up an interesting way for an automatic tool to interact with the SUT, thus serving as a possible interaction model for automatic exploration of REST APIs.

However, the interaction model is only one piece of the puzzle. Methods for generating input data to tests and some way to evaluate the results, an oracle, are also needed. In addition, to have a useful technique in industry, a meaningful and efficient test reporting is also of high importance. If analyzing the results from automated testing is associated with a high cost, potential adoption in industry might be rather low.

In this paper, we propose a method to leverage OpenAPI documents to automatically generate tests. Test inputs are generated using a two-fold mechanism: (i) randomly generated values that are agnostic to the specification, as well as (ii) randomly generated values that conform to the parameter specification in the given OpenAPI document. Test oracles, used to provide verdict on the conformance of response data, are also automatically generated from the OpenAPI document. They assert the REST API results in a property-based fashion.

The choice of property-based testing is made due to its suitability for random exploration, the availability of libraries in that domain, and the feature of shrinking. Shrinking is a feature that, when a test case fails, aims at producing the smallest failing example possible. Property-based testing also allows for formulating and checking several properties of the test results such as if the response body conforms to a given specification, resulting in a stronger oracle than only asserting

¹<https://www.programmableweb.com>

²<https://www.programmableweb.com/apis/directory>

³<https://twitter.com/>

⁴<https://www.youtube.com/>

⁵<https://www.facebook.com/>

⁶<https://azure.microsoft.com>

⁷<https://aws.amazon.com>

on the status code of an HTTP response.

In order for our method to be suitable to a wide range of REST APIs, with as little developer effort as possible, we develop the method as a black-box approach. The approach described in this paper has been evaluated in an industrial case study at ABB and on the large open-source software (OSS) GitLab⁸. Experimental results show that we can find faults and gain insights of the SUT given an OpenAPI document in an automated way.

To summarize we make the following contributions:

- We introduce a method to help developers and testers automatically explore RESTful APIs with low effort.
- We describe how the combination of available open source libraries can be used to implement such a method, as well as provide a proof-of-concept implementation, thus lowering the barrier of entry for industry.
- We investigate how the configuration of generators effect the status code coverage and fault finding probability.
- We present the results of applying the method to multiple APIs developed and used in industry as well as in OSS.

II. BACKGROUND

Before going into the details of this work, we would like to provide some context and explanations of the terms used.

A. OpenAPI

OpenAPI, also known as Swagger, is a way of describing REST APIs [5]. OpenAPI *specification* describes the format which needs to be followed by an OpenAPI *document*. OpenAPI *document* describes an instance of a REST API whose description follows the OpenAPI *specification*.

Since much of the documentation and tooling around OpenAPI refer to Swagger it is worth to mention the history of the names. Swagger was the original name prior to OpenAPI, when the ownership was given to the OpenAPI Initiative [6]. The Swagger name is still used to refer to the tooling as well as the OpenAPI specification. However, in this paper, we further refer to it as OpenAPI. We have also chosen to target the 2.0 version of the OpenAPI specification [7]. The reason to not use the later, 3.0 version, is that the 2.0 is still largely in use and it is the version used in the industrial system which is part of our evaluation.

An OpenAPI-described REST API contains a set of available HTTP verbs (GET, POST, PUT etc.), the parameters and the responses. The parameters describe where the parameter is used in the request (path, query, body etc.), its name, type, and format. Responses are indexed by an HTTP return code and provide details of the structure of the response, if it is a value, an array or a complex object.

Figure 1 shows a simple example of an OpenAPI *document* described in JSON format, which is following the 2.0 version of the OpenAPI *specification*. We can see examples of the available paths and its parameters and expected results. In addition, there is a small definition that specifies the structure of a data type, called *object* in this example.

```

1 {"swagger": "2.0",
2  "info": {
3    "version": "1.0.0",
4    "title": "Example Service API",
5    "basePath": "/theservice",
6    "paths": {
7      "/api/v1/objects/{objectId}": {
8        "get": {
9          "tags": [],
10         "summary": "Object model lookup",
11         "parameters": [
12           { "name": "objectId",
13             "in": "path",
14             "required": true,
15             "type": "string",
16             "format": "uuid"}],
17         "responses": {
18           "200": {
19             "schema": {
20               "$ref": "#/definitions/ObjectInfo"}}}},
21       "definitions": {
22         "ObjectInfo": {
23           "type": "object",
24           "properties": {
25             "objectId": {
26               "format": "uuid",
27               "type": "string"},
28             "name": {
29               "type": "string"}}}}}}

```

Fig. 1. OpenAPI document in JSON-format

B. Property-based testing

The main idea of property-based testing (PBT) is to generate input data and to check if defined properties hold when exercising the SUT with that input. To get a better understanding of the basic principle, let us look at an example of using PBT to test a *sort* function, which sorts a given list of numbers. The first step would be to generate a random list of input numbers. Such a generator is most likely included in a PBT library, but if not, the means to create custom generators are typically there. When we know that we can generate input data we can formulate a property of the function. An invariant of the sort function is that the number at position n in the resulting list should be less than or equal to the number at position $n+1$. To make this into a test we will *check* the property. Checking will exercise the SUT for a given number of iterations. Each iteration is given input data from our generator and verifies that our defined property holds for each such input.

The feature of *shrinking* is also common in PBT. This means that whenever random data has been generated to which a property has failed, the library will try to find the smallest input that fails the property in the same way by *shrinking* it. An example of shrinking is provided later on in Section VII-B.

Formulating properties can be challenging. One such challenge is how to make a model of the SUT without re-implementing the functionality of the property to be tested. This challenge can be avoided by instead of trying to model the implementation, formulating invariants or symmetries. Examples of invariants of a sorting algorithm are that the input and the output should be of equal length and that all input values should exist in the output. These invariants are true given any randomly generated input. Further, depending on the problem at hand we might be able to find symmetries that

⁸<https://about.gitlab.com/>

should hold. A symmetry is when the input data is invariant given a set of operations. As an example, consider a lossless compression algorithm. If we start with generated data and then compress and decompress the data, the symmetry of the compression should ensure that we end up with the same data that we started with.

QuickCheck [8] is known as the first tool for property-based testing. Today there exist re-implementations of QuickCheck in a number of languages, reaching multiple platforms. A few such examples are PropEr [9] for Erlang, ScalaCheck⁹ for Scala on the JVM, and FSCheck¹⁰ for F# on .NET. Our implementation uses the Clojure variant called TestCheck¹¹. TestCheck contains basic generators and also the components needed to define own more complex generators. A reason to choose TestCheck is that Clojure.spec can produce TestCheck generators out of data specifications.

C. Clojure.spec

Clojure.spec¹² is a library for the Clojure¹³ functional programming language. Clojure.spec provides functionality to define specifications of data and to validate if given data conforms to such a specification. These specifications are referred to as *specs*. Clojure.spec also provides means to produce random data generators given a spec. As we shall see, this gives us leverage in our approach when we can use specs to both generate random input data and to validate responses.

As a small example to give an intuition of the basics of a spec, here is an example of a spec called `::age` that defines that to be a valid age the given data must be a natural integer¹⁴ and less than 150.

```
(s/def ::age (s/and nat-int? #(< % 150)))
```

The simplest form of validation is to use the `valid?` function. The result will be a boolean representing if the data is valid checked against the spec.

```
(s/valid? ::age 15) => true
```

To use this spec as a generator we first create a generator from the spec with `gen`. This returns a generator for that spec that can be used in any place where a generator is needed. Here we use it to produce 10 sample values.

```
(sample (s/gen ::age)) => (0 0 2 1 1 10 16 1 26 8)
```

The benefits of using Clojure.spec is both that we can use it as a random data generator and that we can leverage the same specs for validation.

III. PROPOSED METHOD

We propose a method that for a given specification, OpenAPI in our case, produces input generators that are used in property-based tests as well as produce automatic oracles. The properties checked in the tests, with previously produced input generators, are a combination of predefined

static properties and properties automatically derived as valid responses from the specification. Static properties are provided as status codes since success/failure codes are built into HTTP, while valid responses are specified in the OpenAPI document. This produces an automatic oracle that is able to validate if response from the SUT conforms to what is specified, as well as able to identify discrepancy in the given OpenAPI document and response codes received from the SUT.

We aim for a black-box method for test generation of REST APIs, that is as automated as possible. The reason for choosing an automatic black-box approach is to make it easy for developers and testers to use, and for the method to apply to any system using an OpenAPI documented API. We are thus not constrained by implementation details such as languages and platforms.

The proposed approach also makes it straightforward to guide input when it is needed, by having custom generators. This allows testers and developers to guide the input in a way that makes sense for a given SUT. For example, the distribution of string generation between any kind of strings and pattern-based strings, like alpha-numeric strings or UUID (Universally Unique Identifier), can be changed. By doing so, we can ensure that the SUT receives both valid and invalid inputs. A high frequency of any kind of strings would test the SUT in a more fuzzy approach while generating valid input expose a higher number of functionalities in the SUT to be tested.

In addition, our method proposes the *bidirectional* use of the specification. By doing so, a specification is used both to generate valid input as well as to generate a verdict in the form of an automated oracle. The oracle is used to validate that responses conform to the given specification. Since this method automatically derives tests and oracles from the specification of the API under test, both the tests and oracles will automatically evolve with any changes to the API and its specification.

To make results approachable and useful, we propose to leverage the already available method of shrinking, where any property that fails will produce a result that is shrunken to its smallest reproducible case.

Furthermore, we seek to support additional test objectives to fault finding during the testing of the API. Finding faults and failures while fulfilling coverage criteria are considered traditional goals of a testing process and we would like to extend this goal with our method. The properties of an API that are validated when running our generated tests include not only correctness measures but can also check, for example, that all received responses are part of the specified behaviour. Such properties give the ability to draw insights of how the SUT behaves. If a response code is received that is not part of the published OpenAPI document, then the behaviour of the SUT and the specification have diverged. While this may not be a fault in the system's behaviour, it is nevertheless an important insight.

In essence, running tests that only result in a *pass* or *fail* result is a missed opportunity to gain new knowledge. If a test passes 100% of the time, over time the probability of a failure

⁹<https://www.scalacheck.org/>

¹⁰<https://fscheck.github.io/FSCheck/>

¹¹<https://github.com/clojure/test.check>

¹²<https://clojure.org/about/spec>

¹³<https://clojure.org/>

¹⁴Inclusive of 0, according to ISO 80000-2

will be lowered for each test execution, a point will be reached where no new knowledge is gained about the SUT with the computing effort given. As stated by Feldt, such a test case, still active but not effective for finding failures, has grown old [10]. We try to solve this lack of knowledge gathering by generating new test cases that evolve with the selected interaction model, when the OpenAPI document describing the service evolves so does the tests generated, and to not only focus on fault finding properties.

IV. IMPLEMENTATION

To evaluate our proposed method, we implemented a proof-of-concept tool, called QuickREST [11]. QuickREST leverages our method and applies it to REST APIs that are specified with an OpenAPI document.

The overarching process of QuickREST when applying our method to REST APIs, which will be described in detail in following subsections, is as follows:

- 1) Acquire the OpenAPI document, via HTTP or a file.
- 2) Parse the JSON document to an internal format.
 - a) Attach specifications for the parameters.
 - b) Attach specifications for the responses.
- 3) Generate specifications for data definitions.
- 4) Make test generators based on the specifications.
- 5) Check the properties for each of the HTTP verbs defined.
 - a) If in a stateful sequence of operations, store the response.
 - b) Select next input from the collected responses.
- 6) Report the test result.

Figure 2 is an example of how a REST API, as described in Figure 1, would be processed and tested.

A. Parse the JSON document

The first step of parsing is to represent the JSON data of the OpenAPI document in a format that is more suitable to work with in the language of choice. The JSON data is translated into extensible data notation¹⁵ (EDN), which is a subset of Clojure, making the data of the OpenAPI document very accessible in a Clojure program.

The second step of parsing is to attach *Clojure.spec* specifications to the defined parameters and responses. The attached specs will be used both for input generation and for result validation. Figure 2 shows an example of this where "ObjectId" (12 - 16 in Figure 1) and "ObjectInfo" (22 - 29 in Figure 1) are attached and used.

The OpenAPI *specification* for parameters and responses state that the type can be a basic value-type such as number, boolean etc. or a schema reference to a complex object.

For basic value types, predefined specifications were made in the implementation. During parsing, the parameter/response type is then mapped to the spec for that specific type and format. For example, the OpenAPI parameter type of `string` with the type of `UUID` will map to the predefined spec shown

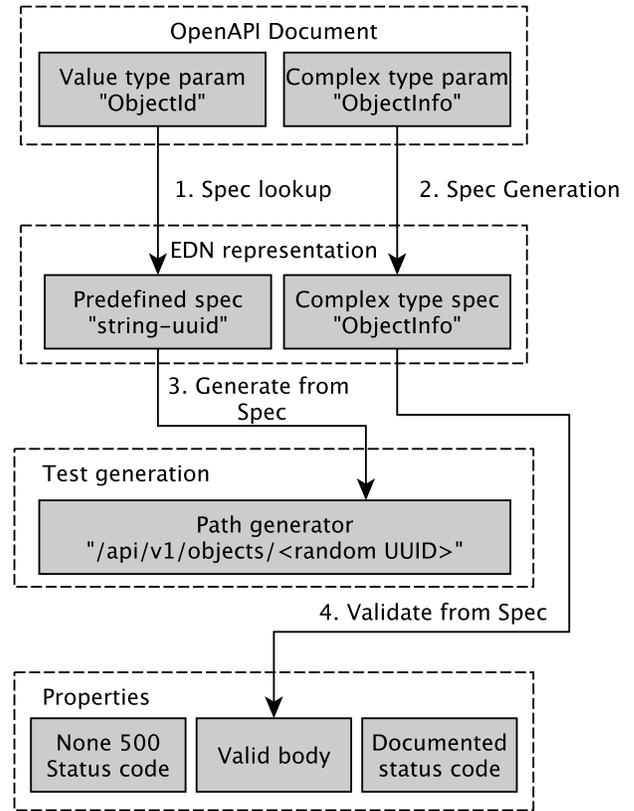


Fig. 2. Example of creation and usage of Specs

in Figure 4. This *spec* specifies (line 4) that the input should be a string and match the regular expression of a UUID. The `:string-uuid` spec has a custom generator that generates a UUID and then makes a `string` from that UUID. We need a custom generator in this case since the default one would be a string generator. The likelihood that the default generator would generate a random string that actually matches the UUID regex is very low and generation would fail.

The types in the OpenAPI document that are references to complex objects will be parsed and a reference to a spec is attached. The definition in Figure 3 that references an array of `ObjectInfo` will be attached as a reference to the spec `:array/ObjectInfo`. In the case where the reference is to a single object and not to an array, as the 200 response in Figure 1, the result would be `:definitions/ObjectInfo`.

In summary, the parsing step of our process starts with the OpenAPI document in JSON format and results in the OpenAPI document represented in a format suitable for further processing, specifications attached to basic value types and reference types, for both parameters and responses.

B. Generating specifications of the definitions

To be able to use the spec references of parameters and responses of complex objects, for input generation and response validation, the actual specs need to be created.

¹⁵<https://github.com/edn-format/edn>

```

1 {"200":
2  {"description":"The query request was successful.",
3   "schema":
4   {"uniqueItems":false,
5    "type":"array",
6    "items":{"$ref":"#/definitions/ObjectInfo"}}}}

```

Fig. 3. Definition of complex object reference

```

1 (s/def
2  ::string-uuid
3  (s/with-gen
4   (s/and string? #(re-matches uuid-regex %))
5   #(gen/fmap str (s/gen uuid?))))

```

Fig. 4. A string uuid spec

```

1 (definitions-to-specs
2  "prefix"
3  {:ObjectInfo
4   {:type "object",
5    :properties
6     {:objectId {:format "uuid",
7                :type "string"},
8            :name   {:type "string"}}}})
9 =>
10 ((s/def :prefix/objectId
11      :openapi.specs/string-uuid)
12  (s/def :prefix/name :openapi.specs/string)
13  (s/def
14   :definitions/ObjectInfo
15   (s/keys :req-un [:prefix/objectId
16                 :prefix/name])))

```

Fig. 5. Edn representation to specs

The specs are created based on the *definitions* part of the OpenAPI document. The definitions specify the types used by other parts of the document, such as parameters to HTTP verbs or expected responses. The definitions are specified using the JSON Schema specification [12]. Given that the definitions are defined in JSON Schema, implementation is needed to translate JSON Schema to Clojure.spec specifications.

Figure 5 shows an example of how the EDN representation of the definitions part of the document in Figure 1 is used to produce specs. Lines 1 - 8 show the call to the `definitions-to-spec` function with two arguments. The first argument is a string naming a namespace for the resulting specs, used to avoid naming conflicts. The second argument is the EDN-data produced by parsing the JSON Schema.

The result of this function call are three specs: One for each property of the object and one that ties these properties to the object. The specs for the properties, lines 10 - 12 in figure 5, reference the predefined specs for that type and format. The spec defining the object, starting in line 13, states the name of the spec, `:definitions/ObjectInfo`, and that the object is a collection of *keys* followed by the expected name of the keys, which also is a reference to its spec.

When the specs have been created for all definitions in the document, the pieces needed to be able to generate input data and to validate responses are in place.

C. Make test generators

To automatically produce examples of the request parameters in the document, generators are required. According to the OpenAPI specification, parameters can be of five different kinds. Those are *Path*, *Query*, *Header*, *Body*, and *Form* [7]. The tool's request generator must take those different kinds into consideration since the HTTP request will look different for different kinds of parameters.

For example, if the type of the parameter is *query*, the generated parameter value should be included in the URL as `/objects?id=value` where *id* is the parameter. In the case of a *body* parameter nothing changes in the URL but the generated parameter value should be included in the *body* value of the HTTP request.

```

1 (gen/sample (s/gen :definitions/ObjectInfo))
2 =>
3 ({:objectId "2acf532d-c60e-472c-8530-c3b5d138327b",
4  :name ""}
5  {:objectId "3cb0fcbe-2f6f-47c6-83de-bccca7c3fdc3",
6  :name "Ax"}
7  {:objectId "148aab47-61a8-4132-8d79-2572c015b822",
8  :name "WNP0"})

```

Fig. 6. Generation of data from spec

When making the generators we use the specs that have been attached to each parameter. We can leverage that Clojure.spec has the ability of producing TestCheck compatible generators from specs. Figure 6 shows an example of how straight forward it is to generate examples when we can use the specs created in previous steps.

The URL generators will iterate over each parameter and insert the generated value in the URL (*Path*, *Query*) or as attached data (*Header*, *Body*, *Form*). To increase the likelihood of generating edge-cases the generators can generate data that is outside the specification. In the OpenAPI document, parameters can be marked as *required*. This will be reflected in the specs produced and such values will always be included in the generated value. However, the generators can be configured to not include required parameters. This allows the generators to produce test cases that test the SUTs ability to handle missing input. Further, generators can also be configured to produce values out of range.

The result of this step are URL generators that can randomly produce valid URLs to call the API with random input.

D. Check properties

With all the specifications in place for the documents parameters, responses and definitions, as well as the generators for the URLs, the last step is to build the test properties and check them. The form of the properties are that *for all* generated URLs for an endpoint, the responses should:

- have a non 500 status code
- have a body payload that conforms to the defined spec
- have a status code that is defined in the document

To be able to use previously gained results, stateful properties can be used. Two ways of implementing stateful properties are to either feed the current state to the generators or to use the current state as a guard.

If the state is given to the generators, the next generated input will be randomly selected from the current state. As an example, a stateful generator could be given a list of valid objects to perform a lookup operation on and return a random item from that list. In the second case a model such as an FSM can be used to verify if the generated data is valid given the current state in the model. If it is not valid it will be thrown away and a new value will be generated.

To be able to generate requests that contain valid input we have used the approach where we give previous results as input to some generators. This is explained further in Section IV-E.

When using a property-based approach, testing the SUT is the process of *checking* that our defined properties hold for the generated URLs and input data. The process of one test run is then to map the *check* function on all our URL generators. Given the generated URL the API will then be called. The received result will be checked for the properties defined. The number of tests generated for each end-point is easily configured.

E. Stateful sequences

It quickly becomes apparent that to be effective for stateful systems and to be able to have status code coverage our approach needs to be more intelligent than what only randomly generated input will give. This is also a finding of Atlidakis et al. [3]. For example, if looking up an object is done via a path parameter with the URL template `/objects/{id}` and the parameter is specified as a *UUID*, then the probability of randomly generating a *UUID* that is identical to an object in the system is very low. Most likely, one would get only *404* response codes (object not found), and never cover the case of looking up an actual object in the system (status code 200).

Our approach is to start out with a pure random input and if we get any result, creating new resources or finding existing ones, store that as a source for future input generation. This means that if we generate a search string that gives a result that was not empty, we store the objects returned. At the next step when generating data for the endpoint that requires an *id* we search the stored results for entities that contain the attribute *id*, if we find any, those entities are then used as a source for our generator. In our implementation we simplified to only support objects with *id* as identifier but the method can be generalized with more engineering effort.

We then start out with pure random input, which is also useful as fuzzy testing, but as we are successful in searches, we learn valid objects that are stored in the system.

F. Report the test results

To be useful to practitioners the presentation of testing results is important. To save time it should be easy to see the overall result of the test run, but at the same time enough details should be provided to debug any failing test. The

```

1 ;; Successful result
2 {:result true,
3  :pass? true,
4  :num-tests 100,
5  :time-elapsed-ms 8607,
6  :seed 1558358174968}
7
8 ;; Failed result
9 {:shrunk
10  {:total-nodes-visited 10,
11   :depth 5,
12   :pass? false,
13   :result false,
14   :result-data nil,
15   :time-shrinking-ms 791,
16   :smallest
17   [{:url
18     "http://service/api/v1/objects?query=<...>",
19     :body []}]},
20  :failed-after-ms 5666,
21  :num-tests 38,
22  :seed 1558358168511,
23  :fail
24  [{:url
25     "http://service/api/v1/objects?query=<...>"
26     :body []}],
27  :result false,
28  :result-data nil,
29  :failing-size 37,
30  :pass? false}

```

Fig. 7. Default output from TestCheck

default output of TestCheck gives a succinct view of the result. Figure 7 provides an example of a successful and failed result. When a test fails, TestCheck will report the failed input and the smallest example after shrinking.

In addition to the output from TestCheck, our implementation collects data on each test executed. Each API call is recorded, and each returned result is also stored. This information can be used by developers when debugging any failures found. In addition to serving as debug information of all calls made, it is also used to present a frequency table of each endpoint and its return codes. This table shows the URL, the verb called, the response code, and how many calls had this return code. It is thus possible to analyze if all endpoints were tested and if all expected return codes were covered.

V. EVALUATION

To evaluate the proposed approach, a multiple case study was undertaken. The research questions we investigated were:

- RQ1: How do different stateless generators compare in terms of response code coverage and fault finding?
- RQ2: How do stateful generators compare to stateless generators in terms of response code coverage and fault finding?
- RQ3: Which additional insights, supplementary to fault finding, can the approach provide?

To answer these questions, we developed a proof-of-concept tool, QuickREST, that implements the proposed approach and applied it in our experiments on several services developed in industry and one open-source project.

A. Studied Cases

The cases selected to evaluate the proposed approach were a collection of services developed as a back-end for a mobile application in industry, and the open-source development management service *GitLab*. The mobile back-end was selected since it was available at our industry partner and that it was composed of services with a REST API described with OpenAPI. The reason for selecting GitLab as a testing target is to be able to compare our approach with another approach of testing REST-APIs (i.e., using *RESTler* [3]).

1) *Mobile back-end*: The system has two main parts. The client side is a mobile application running on either Android or iOS. The main use cases for the mobile application are to search for process objects, such as motors or pumps, and to display information of these objects. The information ranges from name and type to trends of live run-time values.

The mobile application connects to a back-end that is specifically designed to serve the mobile application. The back-end consists of several services that work together to solve the mobile client use cases. These services are web-services and expose their APIs via REST. The API is also described using OpenAPI. These are the services that we have been using as our system under test.

2) *GitLab*: As stated by GitLab, "GitLab is a single application for the entire software development lifecycle. From project planning and source code management to CI/CD, monitoring, and security" [13]. GitLab can be run both as an on-premise solution or as a Software-as-a-service (SaaS) hosted by GitLab. GitLab is available both as a community edition (GitLab CE), which is open-source, and a enterprise edition (GitLab EE), which is closed-source.

GitLab has an extensive REST API [14]. However, there is no official OpenAPI document describing the API. For our evaluation, we have manually produced an OpenAPI document including the end-points selected for inclusion in the experiments. The selection was based on the API operations evaluated by *RESTler* [3]. With that selection we knew of the existence of a bug [15], requiring a stateful interaction of first creating a resource, delete the resource and then edit the deleted resource, all within a very short period of time. With the knowledge of this bug we could try to reproduce the findings of *RESTler* with our method.

We ran GitLab CE locally via a monolithic Docker¹⁶ file [16]. Being monolithic means that the different services that make up a GitLab installation run as one node. This is not a setup for high performance, compared to installing GitLab on a cluster of servers, but it is a setup that is reasonable for a developer to test on as part of their development workflow, thus making it interesting.

B. Experimental Setup

In the evaluation, we exercised all services of the described mobile back-end system as well as the selected GitLab end-points. Each evaluation run of a service consisted of each

individual end-point being tested with randomly generated input. Each end-point was exercised with 10 tests per iteration and after 30 iterations the number of tests were increased by an order of magnitude. The increase in the number of test cases per iteration allows for larger sizes of the generated input. The returned status codes were used as a coverage criterion and included in the test reports. We wanted to cover all status codes defined in the OpenAPI document.

To enable evaluation of RQ1, different frequencies for basic type generators were explored to observe if this changed the number of bugs found. In the first iteration of evaluation, alphanumeric strings were used as the basis for random string generation. The reason being that we would generate valid URLs. In later iterations any string was used to also include invalid input. To further evaluate the effect of different generator frequencies two end-points, where input validation bugs had been found in GitLab, were selected. One of these end-points required string input and one required integer input.

To evaluate stateful interactions compared to stateless interactions (RQ2) sequences of API calls were generated and executed. The result of each call, if it had a body with a payload, were stored in an in-memory database. This made previous results available for input selection of coming calls in the sequence. For example, if the first call in the sequence was a successful POST operation that returned a new resource, that result was stored. If the second operation was a DELETE that required an id, a search would be performed in the database for an id property. If found, that result was used as input. This made sure that we could evaluate valid stateful sequences. If no previous results were present in the database, input was generated in the same way as for stateless tests. To evaluate if the stateful generators could find faults we aimed to reproduce a known bug in GitLab found by *RESTler* [15], requiring a stateful interaction.

In the experimental setup each endpoint of the service described in the OpenAPI document was individually tested. Specs were derived from the OpenAPI document and then used as generators. The generator for each endpoint was then given to *TestCheck* to validate if the defined properties would hold. The properties would check for status codes that do not indicate a crash, that the body of the response was valid according to the specification, and that the returned status code in the response actually is specified in the OpenAPI document. The last two properties were used to evaluate RQ3, where any failed check on those properties indicate a misalignment between the SUT and the OpenAPI document.

C. Results

1) *RQ1: How do different stateless generators compare in terms of response code coverage and fault finding?*: In the experiments, we defined bugs as test cases that resulted in a 500 status code. We found several new bugs during the experiments using a fully automatic approach, both in the mobile back-end and in GitLab [17]–[20]. The tested APIs had no problem in handling random input of alphanumeric strings. However, when the string generator was changed to include

¹⁶<https://www.docker.com/>

TABLE I
GENERATOR EFFICIENCY FOR STRING VALIDATION BUG

T_{fail} show the percentage of *iterations* that found the bug using either a generator producing any string or a generator only producing alphanumeric strings. Each setup was iterated 30 times.

Test cases / Iteration	Generator T_{fail}	
	Strings	A/N Strings
100	0.0%	0.0%
1000	13.3%	0.0%
10000	63.3%	0.0%

TABLE II
GENERATOR EFFICIENCY FOR INT VALIDATION BUG

T_{fail} show the percentage of *iterations* that found the bug using either a generator producing any integer or a generator only producing natural integers (>0). Each setup was iterated 30 times.

Test cases / Iteration	Generator T_{fail}	
	nat-int	int
10	0.0%	0.0%
100	93.3%	60.0%
1000	100.0%	100.0%

any string¹⁷, not only alphanumeric strings, bugs were quickly found. This set of bugs was produced without any statefulness, i.e. without taking previous interaction results in consideration, or generating sequences of API calls. Since these bugs were not dependent on any sequence of calls or specific states, all bugs were categorized as input validation bugs.

As described in the setup, two known bugs in GitLab were selected to evaluate the ability to find bugs by the different type of generators. Table I shows the effectiveness in finding the input validation bug where a string was required and Table II where an integer was required. Looking at the test results of the string and integer generators, it is reasonable to think that we should always use the string generator and the nat-int generator since those were most effective in finding the input validation bugs. However, in addition to finding faults, effective generators should produce input that result in coverage of all defined response codes. Table III presents the frequencies of different response codes for a POST method in GitLabs API, when tested with generators with different probabilities of strings and alphanumeric-strings. In Table III, we can see that the *string* generator is useful for finding the input validation bug (500) but will not generate any input that is valid for the end-point, no 201 responses. To achieve full

¹⁷character values of 0-255

TABLE III
RESPONSE CODE COVERAGE OF POST METHOD

Using the POST method of an API path with a known bug coverage should include 201 (resource created), 400 (bad request) and 500 (internal server error). 100 test cases / iteration for 30 iterations.

Response Code	Generator P(Str, A/N Str)				
	{1, 0}	{0, 1}	{0.5, 0.5}	{0.3, 0.7}	{0.1, 0.9}
201	0.0%	86.5%	15.7%	30.5%	63.0%
400	83.6%	13.5%	71.0%	55.2%	31.8%
500	16.4%	0.0%	13.3%	14.3%	5.2%

coverage of the defined response codes a mix of generators had to be used.

The automatic generation of generators might then, depending on the implementation, produce generators that do not get full response code coverage. To mitigate this, it is valuable to allow the user to tweak the generators or let the implementation learn a distribution that results in full coverage. In summary, *care should be taken when implementing automatic generators to produce the intended range of input values.*

2) *RQ2: How do stateful generators compare to stateless generators in terms of response code coverage and fault finding?:* We reproduced the known bug in GitLab that required stateful interactions [15]. A stateful interaction requires stateful generators, that use data previously seen as a basis to generate new input. Therefore it was not possible to find those kinds of faults with only stateless generators.

Stateful generators greatly increased the likeliness of getting response code coverage on API operations that are dependent on some existing state. Covering a successful response code of a DELETE operation was substantially simplified by first creating the resource with a stateful generator. To get the same coverage with a stateless generator an existing UUID must be generated, and that probability will be low.

While stateful generators simplify covering some cases, they bring their own set of challenges. When random sequences of API calls are generated, random input can be used. But to be able to perform multiple operations on the same resource, the identity of the resource should be selected from a known source, as a database or model. To be effective, stateful properties can use random input for the creation of new resources and random sequences of API calls, but guided input to be able to perform multiple operations on the same resource.

In the cases where the identity of a resource is not consistent in an OpenAPI document, a mapping might be required from the user. For example, if a POST request on *api/persons* is performed and the result is a JSON object with the identity property of `personId` but the DELETE end-point is specified as *api/person/{id}*, we need some input to make that connection. Hence, stateful generators are limited to how well the identity relationships are expressed in the specification. This is also an area where the implementation could apply learning to realize implicit resource relationships in complex systems. In summary, *random input is effective for both stateless and stateful properties but need to be guided for stateful resource selection.*

3) *RQ3: Which additional insights, supplementary to fault finding, can the approach provide?:* We found that several of the service APIs were under-specified. Consequently, the tests produced a failure when a response code is received that could not be found in the given OpenAPI document. The lack of specification was both for *responses* and *parameters*. For the responses the tool could automatically detect this since it is expressed in the properties checked during testing. To find the under-specified *parameters* we had to check the log files of the SUT. An example of an under-specified parameter was one that was specified as an array of *strings* but, to be valid input

to the SUT, it had to be an array of *strings* with the format of *UUID*. From the tools perspective, it got a 400 status code (i.e., malformed input) but the tool could not infer that this is due to lack of specification rather than random input. Looking at the logs of the SUT made the problem apparent, since the input was logged as not conforming to UUIDs.

The UI of the SUT was used after running several tests. This revealed that in some places very long strings, that had been produced as random input, were not truncated and made the UI look bad.

Most of these problems were found by observation by a human. It could be argued that this then could be found without any tool. But we found that the tool is an *augmentation* to a human doing exploratory testing. The human tester does not have to manually produce test cases but can run the tool and act as an observer.

VI. RELATED WORK

Property-based testing has been used to find faults with success on real industry systems, in multiple domains, such as telecom systems [21], file synchronization services [22], automotive systems [23] and databases [24].

For web services, as we target, PBT have been used for services implementing Web Services Description Language (WSDL) specifications, translating the specification to generators and properties [25]–[27]. For REST services Seijas et al. proposed PBT, however according to the authors this was a highly manual approach [28].

Jsongen, is a library for Quviq QuickCheck introduced by Fredlund et al. [29] to generate input from JSON Schema to test web services. It was extended by Earle et al. [30] to include a service’s behaviour, to automatically explore a web service. JSON Schema is related to OpenAPI since parts of an OpenAPI document is expressed in JSON Schema. While JSON Schema is considering only data, OpenAPI also includes the service model.

Aichernig et al. proposes an approach to use business rules in the form of XML, and from those create extended finite state machines used in PBT tests [31]. This is an automatic approach and has some level of intelligence, by walking the FSM, but requires that the services behaviour is available as such an XML artifact.

OpenAPI/Swagger documents have been used to generate tests, both in black- and white-box testing. EvoMaster is a white-box approach that generate tests based on a given Swagger document [4]. Usage of EvoMaster require some developer implementation effort. To require as little developer interaction as possible and to be agnostic of the target platform and languages used, we have used a black-box approach.

RESTler introduces a similar approach to ours by using an automatic black-box approach to intelligently fuzzy-test REST APIs [3]. As stated by the authors, RESTler aims to be a security testing tool. However our goal is to not only find faults but to explore the given REST API and in doing this we test the services with a wider scope of properties. Both RESTler and our approach use the status codes from the REST API calls

as an oracle, but in addition to that we also validate that any payload received actual conforms to the specification in the OpenAPI doc, resulting in a stronger oracle. Our approaches also differ in that we use randomly generated input, not a predefined dictionary.

Ed-douibi et al. propose a model-based approach testing REST APIs specified with OpenAPI [32]. This approach is related to ours with the main difference that we use a property-based method for test generation. Our approach does not only generate static test cases but it also randomly generates parameter values and sequences of operations that leverages previously returned results to perform stateful operations.

VII. DISCUSSION AND FUTURE WORK

QuickREST turns out to be a tool that allows developers and testers to easily test and explore REST APIs. Developers will get quick and actionable feedback while developing, with the option to manually tweak the tests. Testers can both use the tool to find faults and also to exercise a system while doing other tests. However, to get more conclusive results QuickREST would need to be applied to a larger set of systems under test. Here we discuss some implications that our results might have for practitioners and for future research.

A. Developer friendly

As individual end-point tests can be generated in seconds, it is a very developer friendly approach since it can be used during development without interrupting a continuous workflow. However, as the number of end-points and parameters increase, so does the time required. Hence, larger tests may be more appropriate for a continuous integration server.

B. Shrinking

As described in Section II-B, *shrinking* of generated random input is a common feature of PBT. We experienced this firsthand during our experiments. Here we describe two examples of shrinking of the input that produced two of the bugs described. One *path* parameter and one *body* parameter.

Figure 8 shows a piece of an example of pre-shrunk generated input as a *body* to be used in a *POST* request. The result of the automatic shrinking process is shown in Figure 9. Just by comparing the generated input from before and after shrinking it is apparent that the smaller input is preferable as a reproducing case for developers.

In the case of a failing *Path* example, parameters contains unprintable characters so we do not include a verbatim example. The form of an url with path parameters is `/api/v1/objects?name=<random input here>`. In our case the shrinking process could shrink the random parameter from 20 characters down to 2 characters.

The shrinking feature of PBT is very useful for developers who want a reproducing case as small as possible. However, for a fully automatic approach you can get ”stuck” with shrinking. For example, if we have an end-point with input bugs in several of the parameters, shrinking will bring that down to the first case. This means that to get further in our

testing we have to exclude that parameter. This might require manual effort, thus breaking the full automation. However, it could be argued that this kind of shrinking problem forces an agile approach where a developer needs to fix the first found bug before continuing with further testing.

C. Mutate the specification

In an OpenAPI document, parameters can be specified as *required*. To produce valid input to the API this needs to be respected. But it is useful for an automatic approach to in some cases leave out required parameters or change the specified type, *mutating* the API specification. This will produce test cases for a proper input validation of not only the value of a parameter but if the parameter itself exists or not.

D. Iterate over parameters

The size of the input domain covered will be dependant of how many parameters are included. Given an end-point with, for example, 5 required parameters, and 100 tests. Each test case will generate input for all parameters and thus after 100 iterations we have not gone as deep in each parameter as would be the case if there were only a single parameter. Therefore, an automatic approach would get larger input coverage by producing tests for each parameter before starting to use combinations of parameters.

E. Before going stateful, make sure API handles input

Generating stateful interactions will not be effective in finding stateful bugs if the SUT does not handle generated input. If the SUT does not handle input validation any sequence of interactions will shrink to just one call, the one with bad input. We recommend starting with stateless input generation and when the API can handle that, add stateful sequences.

F. Stateful shrinking might not be accurate

When shrinking is performed the API will be called, changing the state of the system. This can result in that the shrunken output is not actually exactly what will produce the error. However, we observed that in our cases, the *sequence* was correct but not the *input parameter values*. In that case, consulting the logs from the system will show the actual input used in the sequence.

G. Augment with manually created model

An automatic approach is a good way to get a lot of testing done with a low effort. However, to get maximum leverage out of this kind of approach a model could be helpful in guiding stateful interactions. Such a model then keeps track of the expected state of the system, guiding valid operations and is a source of verification of how the system should look like.

VIII. CONCLUSION

We have introduced a method and a proof-of-concept implementation to automatically test REST-APIs described with OpenAPI by leveraging PBT. The described approach leverages existing libraries and techniques and to the best of our knowledge this is the first approach that uses automatic

```

1  {:body
2  [ ...
3  ...
4  {:variables #:c0{:T \F},
5  :description "x",
6  :objectId "077de3d9-3f50-4756-bb45-ee61be31e8a2",
7  :model "",
8  :type "",
9  :name "v7"}
10 {:properties {},
11 :variables {@ "\\"},
12 :description "8",
13 :objectId "d9bd22ce-939c-4332-8702-0430e38f04c5",
14 :model "s0",
15 :type "",
16 :name "00"}
17 ...
18 ...}]

```

Fig. 8. Abbreviated pre-shrunked input (60 lines)

```

1  {:body
2  [{:variables {1.0 0},
3  :objectId "ec9c007f-5278-45a7-8aa2-b4cb13ce9e11",
4  :model "",
5  :type "",
6  :name ""}]}

```

Fig. 9. Complete shrunked input

PBT in combination with OpenAPI documents to test REST APIs, with the intent of not only finding faults but also to learn more about the SUT. The experimental results on a real industry software system show that this approach can find real faults and help in gaining new knowledge of the system with a very low effort from the developers and testers.

PBT is a useful technique with substantial tool support that can be leveraged in industry. For example, the capabilities of Clojure's dynamic data processing in combination with Clojure.spec and TestCheck was shown to be a powerful tool-chain to automate our method. The act of shrinking is indicated as a rather useful feature for industry.

In our experience, the usage of PBT in industry seems rather limited. This is unfortunate since, as we have shown, it is an approachable technique that can find real bugs and can be extensively utilised in part of a developer workflow.

To make this approach more effective and more suitable for deeper exploration, we could augment it by a model of the call sequences a real user would perform. In addition, it would be useful to help humans by automatically analyzing the logs while running tests. Exploratory testing is an important part of ensuring the quality of software systems but automation in this area is lacking although there are opportunities for it. The results presented in this paper show that it is possible to assist humans in exploration of REST-APIs with QuickREST.

ACKNOWLEDGEMENTS

This work is supported by ABB, the industrial postgraduate school Automation Region Research Academy (ARRAY) funded by The Knowledge Foundation. Additional support is provided by ITEA3 TESTOMAT project funded by VINNOVA.

REFERENCES

- [1] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, US, 2000.
- [2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12
- [3] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758. [Online]. Available: <https://doi.org/10.1109/ICSE.2019.00083>
- [4] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 3:1–3:37, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3293455>
- [5] "Openapi initiative." [Online]. Available: <https://www.openapis.org/>
- [6] "Openapi initiative." [Online]. Available: <https://www.openapis.org/faq#OAIFAQ-History>
- [7] "Openapi specification, version 2.0." [Online]. Available: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md>
- [8] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, May 2011. [Online]. Available: <http://doi.acm.org/10.1145/1988042.1988046>
- [9] M. Papadakis and K. Sagonas, "A proper integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '11. New York, NY, USA: ACM, 2011, pp. 39–50. [Online]. Available: <http://doi.acm.org.ep.bib.mdh.se/10.1145/2034654.2034663>
- [10] R. Feldt, "Do system test cases grow old?" in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, March 2014, pp. 343–352.
- [11] "Replication package." [Online]. Available: <https://github.com/zclj/replication-packages/tree/master/ICST-2020>
- [12] "Json schema." [Online]. Available: <https://json-schema.org/>
- [13] "About gitlab." [Online]. Available: <https://about.gitlab.com/>
- [14] "Gitlab api." [Online]. Available: <https://docs.gitlab.com/ce/api/>
- [15] "Gitlab sample bug 1." [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-ce/issues/50986>
- [16] "Gitlab docker files." [Online]. Available: <https://hub.docker.com/r/gitlab/gitlab-ce/tags>
- [17] "Gitlab sample bug 2." [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-ce/issues/64573>
- [18] "Gitlab sample bug 3." [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-ce/issues/64576>
- [19] "Gitlab sample bug 4." [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-ce/issues/64580>
- [20] "Gitlab sample bug 5." [Online]. Available: <https://gitlab.com/gitlab-org/gitlab-ce/issues/64583>
- [21] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ser. ERLANG '06. New York, NY, USA: ACM, 2006, pp. 2–10. [Online]. Available: <http://doi.acm.org/10.1145/1159789.1159792>
- [22] J. Hughes, B. C. Pierce, T. Arts, and U. Norell, "Mysteries of dropbox: Property-based testing of a distributed synchronization service," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 135–145.
- [23] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing autosar software with quickcheck," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–4.
- [24] J. Hughes, *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Cham: Springer International Publishing, 2016, pp. 169–186. [Online]. Available: https://doi.org/10.1007/978-3-319-30936-1_9
- [25] M. A. Francisco, M. López, H. Ferreira, and L. M. Castro, "Turning web services descriptions into quickcheck models for automatic testing," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 79–86. [Online]. Available: <http://doi.acm.org/10.1145/2505305.2505306>
- [26] L. Lampropoulos and K. Sagonas, "Automatic wsdl-guided test case generation for proper testing of web services," in *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, WWV 2012, Stockholm, Sweden, 16th July 2012.*, 2012, pp. 3–16. [Online]. Available: <https://doi.org/10.4204/EPTCS.98.3>
- [27] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco, "Automating property-based testing of evolving web services," in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '14. New York, NY, USA: ACM, 2014, pp. 169–180. [Online]. Available: <http://doi.acm.org.ep.bib.mdh.se/10.1145/2543728.2543741>
- [28] P. Lamela Seijas, H. Li, and S. Thompson, "Towards property-based testing of restful web services," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 77–78. [Online]. Available: <http://doi.acm.org/10.1145/2505305.2505317>
- [29] L. r. Fredlund, C. B. Earle, A. Herranz, and J. Mariño, "Property-based testing of json based web services," in *Proceedings of the 2014 IEEE International Conference on Web Services*, ser. ICWS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 704–707. [Online]. Available: <http://dx.doi.org/10.1109/ICWS.2014.110>
- [30] C. Benac Earle, L.-r. Fredlund, A. Herranz, and J. Mariño, "Jsongen: A quickcheck based library for testing json web services," in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '14. New York, NY, USA: ACM, 2014, pp. 33–41. [Online]. Available: <http://doi.acm.org.ep.bib.mdh.se/10.1145/2633448.2633454>
- [31] B. K. Aichernig and R. Schumi, "Property-based testing of web services by deriving properties from business-rule models," *Software & Systems Modeling*, vol. 18, no. 2, pp. 889–911, Apr 2019. [Online]. Available: <https://doi.org/10.1007/s10270-017-0647-0>
- [32] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, "Automatic generation of test cases for rest apis: A specification-based approach," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, Oct 2018, pp. 181–190.