Literature Survey: REST API Automated Test Cases Generation Tools

Preprint · December 2023

DOI: 10.13140/RG.2.2.30083.07203

3 authors:

Imadeddin Salama
Birzeit University
4 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Duha Jarrar
Birzeit University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Mohammed Abuqare
Birzeit University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Literature Survey: REST API Automated Test Cases Generation Tools

1st Dr. Samer AL-Zain
dept. software engineering
Birzeit University
Ramallah, Palestine
szain@birzeit.edu

2nd Duha Jarrar
dept. software engineering
Birzeit University
Ramallah, Palestine
duha.jarrar@gmail.com

3rd Mohammed Abuqare
dept. software engineering
Birzeit University
Ramallah, Palestine
abuqare@gmail.com

4th Imadeddin Salama
dept. software engineering
Birzeit University
Ramallah, Palestine
ce.emad1999@gmail.com

***Abstract*—Finding the right tool that automate test case generation for RESTful APIs could facilitate and accelerate the test and robustness of the APIs. However, there is no clear criteria on how to select that tool due to the lack of direct comparison between these tools. In our survey we cover some of these tools available which includes EvoMasterWB, RestTestGen, RESTler, bBOXRT, RESTest and others. We reviewed two state-of-art empirical comparison studies and a paper focusing on RestTestGen one of the strongest tools available. Then, discussed the results of them in terms robustness and test coverage. As results EvoMasterWB seems to be considered that the most coverage tool and RESTler the most solid tool. RestTestGen seems to take the second most highest coverage.**

***Index Terms*—Test API, Automated Test Cases, Black-box testing**

I. Introduction

RESTful APIs and some refer to REST (REpresentational State Transfer) APIs which is an architectural style that provides standardization between cloud based systems in a way that makes the communication between systems easier [1]. RESTful APIs provide as set of CRUD operations to alter records, CRUD is an acronym for create, read, update and delete in which every operation is corresponding to HTTP methods (POST, GET, PUT and DELETE respectively). A record is identified by a URI. For instance, bookStore endpoint is responsible for altering the records of list of books. The endpoint URI pointing to the records could be /books. Thus, the HTTP method GET request retrieves the records of books and POST /books is used to insert a new record to the books [1]. Additional endpoint input parameters are used to specify more information to the request, for example object identifier to retrieve a specific pet (e.g., /books/{bookId}) or a structured book record to be inserted to the list of books when using the POST method [1].

Despite the advancement of testing practices in software development, writing test cases for REST APIs is time consuming and expensive. Thus, the need for automated test case generators (tools) became essential to facilitate and accelerate the testing effort. Several Automated techniques and strategies have been employed to automate test cases generation to serve different testing purposes. For instance some techniques are based on data dependencies, heuristic request sequences or input data generation [1]. The input data generation techniques can be either based on written samples, previous values, obtaining dictionaries values, trying mutations, or examining constraints in input parameters [1].

The aim of this literature survey is to explore the industrial Rest API testing tools empirical studies and recommend which tools show best performance in terms of robustness and coverage metrics.

According to [1], the fairest comparison could be is against the interface itself as most automated REST API testing tools adopted black-box testing approach based on API interface specifications. API interface specification usually follows OpenAPI specifications standards. The OpenAPI specification is a structured document that standardizes the REST APIs endpoints and defines its specifications which includes how to reach the API using endpoint URI, authentication schema and all operations available including the input parameters and the response structure [1]. The OpenAPI specification document has a list of endpoints. In the BookStore example it has two endpoints: /books and /books/{bookId}. Each API endpoint has one or more HTTP methods with corresponding parameters in the URI that are defined with curly braces in addition constraints in the input values and input data type. It also specifies the response format in nominal cases and errors format in when an error occurs.

In [1], four automated REST API testing tools have been selected based on relevant literature. These tools are RestTestGen, RESTler, bBOXRT and RESTest. These tools have been deployed and tested under the same settings, environment, and initial state on 14 REST APIs case studies. Hence, some of these tools have been excluded due to reliability issues (ex. crashes) or being unable to perform the majority of case studies like RESTest. The case studies were limited down to 8 cases which are the common successful cases among the best performing tools. RestTestGen was the only tool that was able to successfully perform on all 14 cases followed by bBOXRT that performed on 8 out the 14 cases [1]. Afterwards, the test cases resulting from the experiment were compared based on eight coverage metrics proposed by Martin-Lopez to determine the best test coverage tool using Restats as a proxy to compute those metrics.

Meanwhile, the authors of [2] selected 19 tools, 8 research tools, and 11 practitioners' tools, then limited them 10 tools

which worked in the benchmark of 20 services and were most popular in GitHub including EvoMaster, and Schemathesis. After that, the authors used the JaCoCo code-coverage library to collect coverage information and error-detection data.
In the following sections, we will discuss RestTestGen as an example of REST API automated test cases generation tools and two state of art papers of empirical comparisons of REST API automated test cases generation tools studies. In Conclusion Section, we will present the conclusion and limitation of the studies in addition to future work recommendation.

II. Discussion

Before diving deep into the discussion, in the next subsection we will explain how RESTTESTGEN works to assist us in the discussion and analysis. RESTTESTGEN is one of the strongest tools taken in the two empirical comparison studies and most of the tools are based on similar concepts.

A. RESTTESTGEN

In [3], the authors proposed a solution called RESTTESTGEN which is a tool to create test cases of REST APIs through Swagger and OpenAPI documentation. The tool is built based on different components:

1) Operation Dependency Graph: The operation dependency graph is the component where the swagger documentation enters as input and the component gives an output a graph of the REST APIs based on dependency for example to do getSchoolsById you need a valid id that you get from getSchools. The dependency graph is built on this component. The component consists of 2 sub-components.

a) Graph Construction: In this component the dependency graph is built based on the APIs, the dependency is built based on that the id or parameter needs to be passed exists on another API as the example of schools mentioned above. The graph is drawn between the nodes with an edge representing a dependency and the edge is labeled by the dependency field (e.g. schoolId).

b) Dependency Inference: This component resolves the issue of naming the fields of dependency for example the schoolId can be called id, SchoolID, school_id . . . etc. To solve the issue the writer used insensitive case for field name, Id completion to remove the verbs, names and take the last name would be id and add it to the object and Stemming which is a way to tolerate some differences for example school and schools are the same [3].

2) Nominal Tester: This component is used to generate correct test cases and execute them based on ODG. They generate the values randomly based on boundaries specified in swagger. The tester executes the test leaves first in the graph which contains the dependency and saves the values inside the ResponseDictionary to be used in the dependent test cases. The APIs are being tested based on 2 oracles: response code 2xx: success, 4xx: wrong values applied, 5xx: means bug and the response structure should be the same as swagger with possible different naming [3].

3) Error Tester: This component is used to generate correct test cases and execute them based on ODG. The test cases are mutated several times on each field by removing required fields, out of range values and wrong input type and then the values are tested using oracles based on response codes 2xx means there is a bug, 4xx means pass, 5xx which is defect [3].

B. Methodologies

In order to test the tools or compare them first, you need to select case studies that support open specifications, so in the next subsection we will discuss the selection procedure of those cases studies.

1) Case Studies Selection Procedure: According to [3], the authors collected the case studies of REST APIs from a website called: API.guru accessed on 18 June 2019. Then eliminated the case studies that were not responding, and the ones that needed authentication to avoid needing time for sign up and sign in per request. In addition of selecting the case studies that support swagger documentation. The final cases studies were 87 REST APIs containing 2612 operations (an average of 30 operations per REST API).
On the other hand the authors of [1] decided to control the environment settings to avoid internal state dependencies of the case studies to have a fair comparison between tools which is not the case in [3]. Thus, the authors searched for REST API open-source projects on GitHub that can be deployed locally which also enables them to prepare fresh start database for each case study (project). Which also provides the ability to rest to the fresh start before each testing iteration. The methodology used to search for the tools were based on search for keywords, “REST”, “RESTful API”, “OpenAPI” and “Swagger” then subsequently, searched for common keywords represent frameworks that support REST APIs, for example “swagger-ui”, “SpringFox”, “swagger-jsdoc”, “flask-swagger”. Then filtered out those case studies based on whether it supports OpenAPI specification or not as a prerequisite input for black-box testing. In addition to filtering out the failed API cases.
The authors claimed that those APIs represent the real world APIs since the cases studies were written in different programming languages, frameworks and DBMS. Most importantly the case studies have good variety of complexity levels of number of operations, and dependencies in addition to comprehensive business cases.
The same methodology followed by [2], the only difference is that instead of running those case studies locally using dockers, the authors of [2] used a set Google Cloud machines as one machine per case study to avoid interference between them.

2) Tools Selection Procedure: To collect the tools according to [2], the author searched for related papers published since 2011 using the following keywords ”rest”, ”API”, and ”test”. Then selected the tools that follow REST API standards and produced actual test cases. As a result of this process 19 tools, part of which 8 research tools, and 11 practitioners’ tools, then eliminated these tools to 10 tools that preformed well over

the benchmark of 20 services (case studies) and were most popular in GitHub. Similar approach followed by [1] but only considered black-box testing and 4 tools were selected and tested on 14 case studies.

3) Experiment and Analysis: The first step authors of [1] took is to build an initial database for each case study by manually interacting with each case APIs to support all REST operations, for example DELETE operation needs a record to be created first in order to not fail. Then the authors took a snapshot of each database to support restoring the fresh start state of each case study. For that purpose the authors used "Docker" to guarantee full-reset. The second step is to make sure that the tools are in the default settings or best performance recommended by the owners. Afterwards, the authors ran each tool over each case study for 10 iterations. Each iteration starts with initial state restored and with a 10 minutes budget in order to guarantee testing strategy of random variation of non-deterministic algorithms. The results of execution were captured using Restats as a proxy to compute the coverage metrics.

The authors of [2] ran each tool for an one hour over 10 iterations to achieve randomized results. After that, he used the JaCoCo code-coverage library to collect coverage information and error-detection data. Additionally, to avoid any interference between runs, each tool was ran once for 24 hours using a fresh machine. The authors of both empirical studies seems to follow the same approach but using different tools to maintain the fresh start and compute and monitor the results of the experiments. The author of [3] tested the 87 REST APIs on nominal tester and error tester setting time budget to only 30 minutes. However, the time taken for the nominal tests is only 10 minutes proving that the time taken only for the response time needed but the purpose of the experiment was to test the tools against the case studies without comparison in mind.

Restats has been used to computed metrics in [1] computes the metrics by only monitoring the HTTP requests and responses without having access to the internal source code of the REST API [4]. But, JaCoCo code-coverage library that has been used by [2] requires access to source code in order to compute coverage metrics [5].

In terms of results, the author of [1] analyzed the result based on robustness and coverage, meanwhile the author of [2] analyzed the results based on coverage achieved, failure points, and implications.

The failure analysis was captured during the execution of the experiment as shown in Table I which shows the failures or success of each tool against each case study.

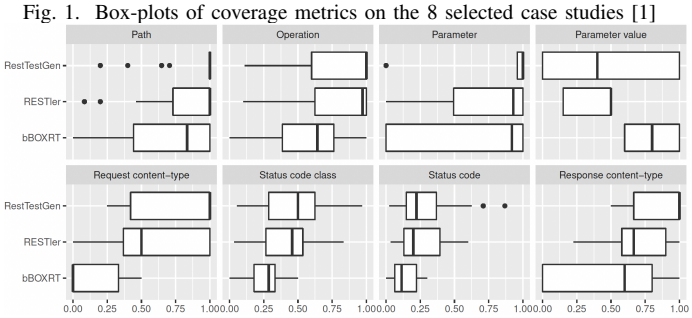
When analyzing the results of Table I, RESTler scored the most robust automated testing tool, as it was able to perform on all the 14 case studies without failure. RestTest-Gen scores second most robust tool as it performs 11 cases out of 14, and bybBOXRT comes next and performs 8 out of 14 cases. RESTest scored the least robust automated testing tool which caused it to be excluded from coverage metric analysis [1]. The author of [2] did not conduct robustness analysis and considered all case studies and tools are valid.

TABLE I
SUMMARY OF TEST CASES FOR RESTTESTGEN, RESTLER, BBOXRT, AND RESTEST [1]

Case Study	RestTestGen	RESTler	bBOXRT	RESTest
01-Slim	✓	✓	×	✓
02-Airline	×	✓	×	×
03-Streaming	×	✓	×	×
04-Petclinic	✓	✓	×	×
05-Toggle	✓	✓	✓	×
06-Problems	✓	✓	×	×
07-Products	✓	✓	✓	×
08-Widgets	✓	✓	✓	✓
09-Safrs	✓	✓	✓	×
10-Realworld	✓	✓	✓	×
11-Crud	✓	✓	✓	×
12-Order	✓	✓	✓	×
13-Users	✓	✓	✓	×
14-Scheduler	×	✓	×	×
Total	11	14	8	2

In order to answer the coverage questions in [1], the authors used two approaches, the first approach by plotting the experimental data distributed of each coverage metric in a box-plot where each box-plot presents the result of a coverage metric over 3 selected tools as shown in Figure 1. For instance, the box-plot Operation coverage metric in the first row of the second column shows RestTestGen and RESTler have very high and close values, and bBOXRT scores lower values. RestTestGen appears to score the highest values over other tools in Parameter, Request content-type, Response content-type, Status code class and Status code coverage metrics. RestTestGen and RESTler have close values of Path coverage. bBOXRT scored the highest value of Parameter value coverage. RESTler did not score any of the highest values of any coverage metric.

Fig. 1. Box-plots of coverage metrics on the 8 selected case studies [1]



The second approach the authors used a technique to record the "win" of each tool against each coverage metric. The "win" is hit each time the tool scores the highest value over other tools for each coverage metric as shown in Table II. The Draw represents a number of cases where no tool wins.

RestTestGen scored the higher Operation coverage than RESTler and bBOXRT on 1 case study. On the other hand, RESTler scored the highest value of Operation coverage on 3 case studies. The remaining 4 case studies of operation coverage have no winners. Accordingly, RestTestGen is the automated testing tool producing test suites with higher cover-

TABLE II
COVERAGE METRICS FOR RESTTESTGEN, RESTLER, BBOXRT, AND DRAW [1]

Coverage Metric	RestTestGen	RESTler	bBOXRT	Draw
Path	1	0	0	7
Operation	1	3	0	4
Parameter	1	0	0	4
Parameter value	1	0	2	0
Req. content-type	2	1	0	4
Status code class	4	4	0	0
Status code	5	3	0	0
Resp. content-type	3	2	0	2

age, because it performed better than other tools for 5 coverage metrics over the eight case studies, meanwhile RESTler and bBOXRT were the better performer over other tools only on one of coverage metrics each.

TABLE III
AVERAGE LINE, BRANCH, AND METHOD COVERAGE ACHIEVED AND 500 ERRORS FOUND IN ONE HOUR (E/UFP/ULFP) [2]

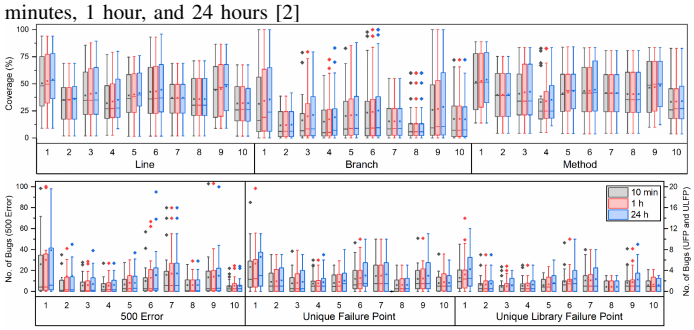
Tool	Line Coverage (%)	Branch Coverage (%)	Method Coverage (%)	#500 (E/UFP/ULFP)
EvoMasterWB	52.76	36.08	52.86	33.3 / 6.4 / 3.2
RESTler	35.44	12.52	40.03	15.1 / 2.1 / 1.3
RestTestGen	40.86	21.15	42.31	7.7 / 2 / 1
RESTest	33.86	18.26	33.99	7.2 / 1.9 / 1.1
bBOXRT	40.23	22.20	42.48	9.5 / 2.1 / 1.3
Schemathesis	43.01	25.29	43.65	14.2 / 2.8 / 2
Tcases	37.16	16.29	41.47	18.5 / 3.5 / 2.1
Dredd	36.04	13.80	40.59	6.9 / 1.5 / 0.9
EvoMasterBB	45.41	28.21	47.17	16.4 / 3.3 / 1.8
APIFuzzer	32.19	18.63	33.77	6.9 / 2.2 / 1.3

According to [2] analysis, as shown in Table III and Figure 2, the results of the experiments depicted for each tool with its coverage of line, branch, and method and the errors, as it shows "EvoMasterWB" is the best tool. Never the less, EvoMasterWB's coverage generally is not very high, the author identified the factors of that are due to that the generated parameter values are invalid, operations dependency issues, and the mismatch between APIs and their specifications. Another note, that the author mentioned is that when the coverage grows over time budget when there is a simple parameters with no specifications. Such issues didn't happen in [1], since the tools were filtered based on robustness before proceeding with coverage test. The finding-bug is a performance measure of the automated testing tool, so the author measured the tools with 500 unique errors, based on column 5 in Table III, the best tool detected 33.3 on average of several trials which is by a white-box tool "EvoMasterWB" but regarding the author, there are cases in which white box tools cannot cover and black-box tools can. The reasons for the failures could be unchecked parameter values or outside the service (network issue). When the author tried to investigate the correlation coefficient of coverage and failures, they found that there is a strong positive correlation between the failures and the tools. The tools that had high coverage, had more failures. Also, the author mentioned that different input types and using different input combinations can lead to more faults.

In our opinion the methodology of filtering out the tools first before starting the coverage testing made the comparison between tool more fair.

According to [2], there are some implications due to techniques and tools development. for example better input parameter generation for improvement. The author mentioned that we can improve the tool's performance using better input generation, especially with white-box tools such as EvoMasterWB which achieves higher coverage and could find 500 errors more than black-box tools.

Fig. 2. Code coverage achieved over all services by the ten tools in 10 minutes, 1 hour, and 24 hours [2]



In highlight of [3] results, the time taken for the nominal tests is only 10 minutes proving that the time taken only for the response time needed. They achieved 625 operations with 2xx status code, and 151 operations with 5xx status code showing faults and bugs in the REST APIs.

They were able to generate 1285 mutants of the nominal test cases of missing required fields, wrong types and wrong boundaries. They were able to detect 23 internal server error bugs, and the majority 864 cases are wrong data that were handled as correct data. Based on the authors, the tool created was a great tool for black box testing offering a way to generate test cases and automatically executing them within a budget. Responses with 4xx code were not checked correctly as it is not all of it are bad request checks. Oracles used using the status code of response is not enough to view a point that it causes an error. The algorithm of fuzzing input to generate the test cases is a random algorithm which needs to be improved.

From our point of view, we support that the tool is magnificent in checking the cases of REST APIs, but it does not check the most important core logic by purpose; it just fuzzes the values randomly, enhancing the logic of generated test cases would be useful for this point. Many developers maintain or change the code without changing the swagger documentation which makes it a limitation. Most of REST APIs used are authenticated APIs which makes a need to do some automation or authentication component and removing the authenticated APIs make the APIs in the dataset very less to test such an important tool.

III. CONCLUSION

At the conclusion and based on the results presented from the mentioned comparisons and results, EvoMasterWB gave the best coverage preference, given it out performed RestTestGen that took the best place in Black-box Testing comparison.

There are several recommendations to enhance the performance of the automated test cases tools for instance using Natural Language Process. The author of [2] introduced a technique to extracting the values from the descriptions of the specification by using the hints of the response messages of the servers. So, the author implemented a proof-of-concept text-processing tool to check the suggested values which parses the descriptions and collect nouns, pronouns, and their dependencies. Using those techniques helps detect useful values for the parameters.

Several techniques could be applied to improve the tools and coverage cases by using latest implementation of large language models for example ChatGPT. That would help in analyzing proving more accurate values for API.

REFERENCES

[1] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "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, 2021, pp. 226–236.

[2] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for rest apis: No time to rest yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 289–301.

[3] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 142–152.

[4] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Restats: A test coverage tool for restful apis," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 594–598.

[5] "Jacoco - java code coverage library," https://www.eclemma.org/jacoco/, accessed: June 3, 2023.