Assessing Black-box Test Case Generation Techniques for Microservices

Luca Giamattei(✉), Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo

DIETI, Università Degli Studi di Napoli Federico II, Napoli, Italy
{luca.giamattei,antonio.guerriero, roberto.pietrantuono,stefano.russo}@unina.it

Abstract. Testing of microservices architectures (MSA) – today a popular software architectural style - demands for automation in its several tasks, like tests generation, prioritization and execution. Automated black-box generation of test cases for MSA currently borrows techniques and tools from the testing of RESTful Web Services.

This paper: *i)* proposes the uTest stateless pairwise combinatorial technique (and its automation tool) for test cases generation for functional and robustness microservices testing, and *ii)* experimentally compares - with three open-source MSA used as subjects - four state-of-the-art black-box tools conceived for Web Services, adopting evolutionary-, dependencies- and mutation-based generation techniques, and the proposed uTest combinatorial tool.

The comparison shows little differences in coverage values; uTest pairwise testing achieves better average failure rate with a considerably lower number of tests. Web Services tools do not perform for MSA as well as a tester might expect, highlighting the need for MSA-specific techniques.

Keywords: Microservices · Black-box testing · Robustness testing

1 Introduction

Microservice Architectures (MSA) are a service-oriented software architectural style where services are loosely coupled, run in their own processes, and interact via lightweight mechanisms [1]. These characteristics favours services' development by different teams and possibly in various programming languages, and their independent deployment. MSA are often engineered with agile practices, enabling rapid and frequent software releases (even many per day).

Testing automation is essential to fully benefit from the MSA architectural paradigm and related practices. Techniques for black-box (or specification-based) automated generation of test cases for microservices are mainly borrowed from testing of RESTful web services, enabled by documentation of their interfaces [2]. The most notable open format for specifying web services and MSA Application

A. Vallecillo et al. (Eds.): QUATIC 2022, CCIS 1621, pp. 46–60, 2022.
https://doi.org/10.1007/978-3-031-14179-9_4

Programming Interfaces (API) is OpenAPI/Swagger [3].[1] Specifications include service Uniform Resource Identifier (URI), HTTP method, type and name of every parameter, and HTTP body. OpenAPI allows to automatically retrieve the interface of a microservice of interest from its IP address and port number.

Testing RESTful services may be challenging, due to dependencies of tests from the state of internal resources (e.g., a database) or from external services [2]. Thus, many test generation techniques are stateful [4–6]. They aim at maximizing coverage values, like those defined by Martin-Lopez *et al.* [7].

With respect to generic RESTful web services, microservices are expected to have finer granularity and to be self-contained (being designed with a single business responsibility), polyglot and independent. An MSA typically includes many RESTful services, whose complex dependencies are challenging to cover by stateful techniques when microservices are tested independently.

This paper provides an empirical comparison of five techniques for test case generation for microservices. Four of them are state-of-the-art techniques for RESTful web services, claimed to be applicable to microservices, namely: EvoMaster [2], RestTestGen [4], RESTler [5] and bBOXRT [8]. The fifth technique (uTest) is a pairwise combinatorial strategy that we propose here with its support tool, which automatically retrieves OpenAPI specifications of microservices to test, generates test cases, executes them and gathers results.

The experimental comparison uses as subjects three well-known open-source MSA (Train Ticket, SockShop, FTGO), with reference to two scenarios:

i) generation of a suite of tests with only *valid* inputs, i.e., adhering to the service API specification; this is a typical functional testing scenario;
ii) generation of a test suite with both *valid* and *invalid* inputs (thus including tests violating the specification); this may serve to test the service for robustness against unexpected inputs, or for coverage of return codes.

The results of experiments show that tools reach comparable values for eight coverage metrics (five input and three output coverage metrics), but exhibit different average failure rate and test generation/execution cost. The proposed combinatorial approach shows to be more cost-effective as it generates a lower number of test cases - thus exhibiting the best average failure rate.

The rest of the paper is so organized: Sect. 2 discusses related work. Section 3 describes uTest. Sections 4 and 5 present experiments and results, respectively. Section 6 discusses threats to validity. Section 7 contains final remarks.

2 Related Work

Automated black-box generation of test cases specifically for microservices is a research topic not much investigated so far, yet techniques/tools for Web Services may by applied to microservices as well.

A state-of-the-art tool for automated testing of RESTful web services is EvoMaster [2]; initially conceived for white-box testing, it has been extended for

[1] https://www.openapis.org.

black-box testing. This is performed by random testing, adding heuristics to maximize the HTTP response code coverage. During the evolutionary search, EvoMaster runs tests as HTTP service requests and evaluates the fitness of every generation (in the evolutionary meaning) of test cases. The test suite is produced in several formats (e.g., for JUnit).

RestTestGen is a stateful test generator proposed by Corradini *et al.* [4], that infers operation dependencies, first statically from OpenAPI specifications, and then dynamically, using feedback from executed tests. Input values are generated from a dictionary, from documentation examples, randomly, or re-using past observed values. It generates nominal as well as invalid test cases; these are obtained by mutating successful test cases (those that returned 2xx or 4xx HTTP codes), e.g. by removing values of mandatory parameters.

RESTler has been proposed by Atlidakis *et al.* at Microsoft Research [5], as a tool for stateful input generation via fuzzing, aiming to find security vulnerabilities. It generates sequences of requests based on data dependencies among operations. These are detected by first statically inferring producer-consumer relations from the OpenAPI specification, and then - like RestTestGen - dynamically analyzing responses of executed tests. Input values are selected from a user-configurable dictionary, or from previously observed values.

bBOXRT is a tool for robustness testing of REST services proposed by Laranjeiro *et al.* [8]. The authors designed a method for injecting faults in requests, attempting to trigger erroneous behaviors. The tool generates and executes valid requests with random values compliant with the OpenAPI specification, and then mutates inputs observing the system behaviour under a faulty workload. bBOXRT supports a large number of mutations of input parameters values.

Martin-Lopez *et al.* [6] propose a black-box technique/tool RESTest for RESTful APIs, based on dependencies among parameters, expressed in an Inter-parameter Dependency Language (IDL). Results may benefit from available additional information on dependencies, that however testers need to write in IDL; this is time-consuming and requires a deep knowledge of the system under test.

Three further OpenAPI-based techniques are proposed by Ed-douibi *et al.* [9], Karlsson *et al.* [10], and Baniaș *et al.* [11]. The first generates (JUnit) tests inferring both valid and invalid parameter values. The second (QuickREST) includes property-based stateless and stateful generators, that are compared in response codes coverage and fault finding ability. The third performs combinatorial generation, adding human intervention to augment the quality of the generation.

An important empirical comparison of black-box techniques for RESTful services, based on the coverage metrics of Martin-Lopez *et al.* [7], has been presented by Corradini *et al.* [12]. They analyzed existing tools and selected a number of them for comparison based on "robustness", meant as the ability to run on different case studies. The compared tools include RestTestGen [4], RESTler [5] and bBOXRT [8], but not EvoMaster, whose black-box version was not available yet, and RESTest, that did not pass the robustness filtering.

With respect to the analyzed literature, our contribution is twofold: *i)* we propose the uTest automated stateless combinatorial test generation technique for microservices; *ii)* we analyze experimentally the performance of the main existing tools for Web Services when used for testing MSA, and compare uTest to them. Table 1 summarizes compared tools.[2]

Table 1. Compared tools/techniques

Tool	Test specification	Test case generation
EvoMaster [2]	State-based	Evolutionary
RestTestGen [4]	State-based	Data/Operation dependencies random dictionary mutation
RESTler [5]	State-based	Data/Operation dependencies dictionary
bBOXRT [8]	Classes from API specification	Random mutation
uTest	Classes from API specification combinatorial	Random

3 The uTest combinatorial testing strategy

3.1 Background

Combinatorial design is a consolidated strategy for automatic test generation [13], extensively studied in the literature [14]. It aims to detect multi-factor faults with the use of combinatorial methods, that demonstrated good fault detection ability [15]. The definition of an Input Space Model, to identify factors (and their values) that might affect the output, is crucial in this strategy. Exhaustive enumeration of all combinations of factor's values can be impractical, as these rapidly explode in number [16]. A solution is to generate a so-called *t-way* test suite, composed of (a subset of all) combinations of t factors.

3.2 Combinatorial Test Case Generation Strategy

We adopt a *pairwise* strategy to generate a *2-way* test suite covering combinations of pairs of input classes. Tests are derived from the specification of the microservice; factors are the parameters of HTTP requests, and their values are generated in compliance to the specification. For any pair of classes c_i, c_j of parameters p_i, p_j, a test case is generated with values $v_i \in c_i$ and $v_j \in c_j$, respectively.

The generation of a *test suite* is composed of three main steps:

[2] Our comparison does not include the techniques in references [6,9,10], and [11], whose tools are probably still at proof-of-concept stage. For instance, like Corradini *et al.* [12] we did not manage to run RESTest on our case studies. However, differently from [12], our comparison includes Evomaster, besides RestTestGen, Restler, bBOXRT.

1. Input space partitioning. The OpenAPI specification of all microservices in the MSA is parsed to extract an Input Space Model consisting of HTTP methods, URIs and body templates, HTTP status codes and parameters' details (type, bounds, default value, etc.); equivalence classes for all parameters are then categorized into valid and invalid.
2. Test cases specification. Based on equivalence classes, test cases specifications are produced according to a pairwise combinatorial strategy.
3. Test cases generation. Actual test cases are generated, randomly choosing values from equivalence classes based on the test cases specifications.

Listing 1. A sample microservice OpenAPI specification

```
host:    exampleHost:8080
paths:   '/carts/{customerId}/items':
             post:
             parameters:
                 - name: customerId
                 in: path
                 required: true
                 type: string
                 example: 579f21ae98684924944651bf
                 - name: body
                 in: body
                 required: true
                 schema: '$ref': '#/definitions/CartItem'
             responses:
                 '201': description: 'Created'
                 '400': description: 'Bad Request'
definitions:     CartItem:
                     type: object
                     properties:
                       itemId:
                             type: Integer
                       discount:
                             type: boolean
                     required:
                       - itemId
```

Listing 1 shows a snippet of the OpenAPI specification of a microservice with three parameters - one *in path* (`customerId`, required) and two *in body* (`itemId`, required, and `discount`, optional). It returns 201 or 400 HTTP status codes.

At step 1, the domain of values of each parameter is partitioned into *equivalence classes*. We define them like Bertolino *et al.* [17], based on the parameter type and, when specified, value bounds, example value, default value, and obligatoriness. Then, we categorize classes into *valid* or *invalid*: valid classes (invalid classes) contain for input parameters only values compliant to (violating) the microservice specification. An example of input space partitioning for the microservice of Listing 1 is shown in Table 2.

At step 2, *test case specifications* are derived. Table 3 shows an example for the microservice of Listing 1, derived from the partitioning of Table 2. The URI and body templates include for each parameter the equivalence classes, from which a value shall be chosen for a test case. For instance, a test case generated from the specification in Table 3 shall have for p_1 (*customerId*) a value chosen from class $c_{1,2}$ (the *example* value in Listing 1); for p_2 (*itemId*) a value from class $c_{2,2}$ (negative value in range), and for p_3 (*discount*) the value *true* or *false*.

Table 2. Input space partitioning for the microservice of Listing 1

Parameter	Name	Type	Equivalence classes	Category
p_1 *(required, in path)*	*customerId*	*string*	$c_{1,1}$: string in range	*valid*
			$c_{1,2}$: specified example value(s)	*valid*
			$c_{1,3}$: empty string	*invalid*
			$c_{1,4}$: no string	*invalid*
p_2 *(required, in body)*	*itemId*	*integer*	$c_{2,1}$: positive value in range	*valid*
			$c_{2,2}$: negative value in range	*valid*
			$c_{2,3}$: alphanumeric string	*invalid*
			$c_{2,4}$: no value	*invalid*
p_3 *(optional, in body)*	*discount*	*boolean*	$c_{3,1}$: {true,false}	*valid*
			$c_{3,2}$: no value	*valid*
			$c_{3,3}$: empty string	*invalid*
			$c_{3,4}$: alphanumeric string	*invalid*

Table 3. A test case specification for the microservice of Listing 1

URI template	http://examplehost:8080/carts/ {$c_{1,2}$}/items
HTTP method	POST
body template	{"*itemId*":{$c_{2,2}$},"*discount*":{$c_{3,1}$}}
HTTP status code	201, 400

A *test suite* shall entail test cases combining values from input classes according to a pairwise strategy. To this aim, uTest uses a recursive algorithm. Two *valid* equivalence classes per parameter are selected to generate a nominal test suite (when available, examples and default values are preferred as valid classes). Then, for each method of each path uTest builds a binary tree, whose leaves represent all combinations of classes. The tree for the example of Listing 1 is shown in Fig. 1, having selected the (only) two valid classes for each parameter in Table 2; leaves represent all combinations of pairs of selected classes. Test case specifications like the one in Table 3 shall be generated only for the subset of four combinations in the red boxes in Fig. 1 (output combinations), which includes all possible pairs of classes selected for the three parameters; the example of Table 3 corresponds to the combination in the green box. For functional and robustness testing, one *valid* and one *invalid* class are selected per parameter.

At step 3, actual test cases are finally generated, by randomly picking values from equivalence classes defined in the produced test cases specifications. This is done statically: no test is generated depending on the result of the execution of some previous tests. We call *valid test cases* those containing for all parameters values belonging to valid equivalence classes. We call *invalid test cases* those where the value of at least one parameter belongs to an invalid class.

3.3 The uTest tool

The proposed pairwise strategy is prototyped in the uTest tool, whose architecture is in Fig. 2. It is designed as a microservice deployable in a Docker container along with the MSA under test; it retrieves the API of microservices in

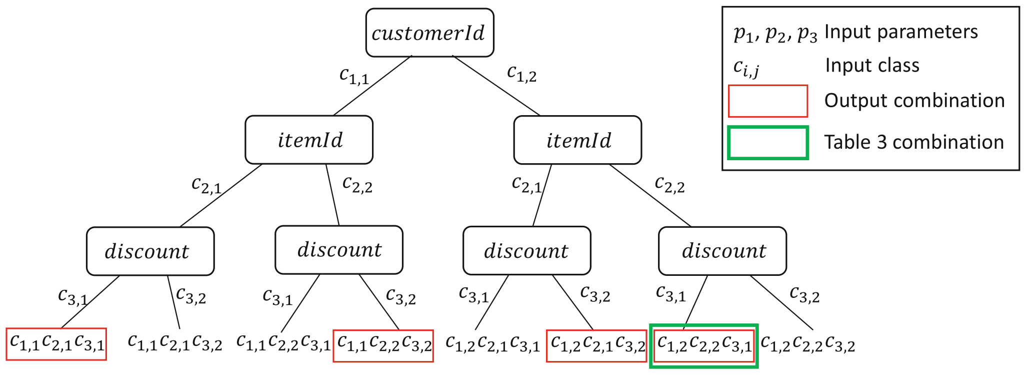


Fig. 1. Generation of valid test cases specifications for microservice of Listing 1

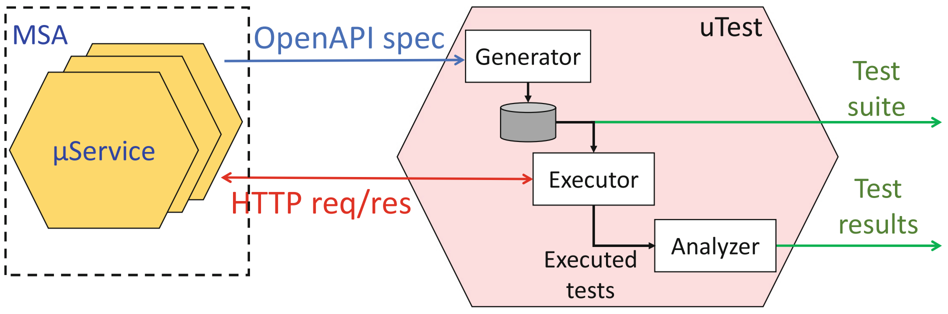


Fig. 2. uTest architecture

the MSA, including those not directly accessible by the user perspective (e.g., hidden behind a gateway or so-called edge microservices). uTest includes a *Generator* of the test suite, a tests' *Executor*, and an *Analyzer*, for the computation of coverage metrics.

The generator is fed with a specification (currently, in OpenAPI version 2.0) of the entry points of the microservice under test. uTest automatically retrieves the specifications of application's microservices, exposed through an OpenAPI interface. Otherwise, it is possible to feed specifications to uTest manually. uTest pairwise generation produces test cases specifications for all combinations of two selected equivalence classes of any pair of input parameters. The generated test suite is stored inside the Docker container and can be executed (or exported) by the tester.

Tests are run by the *Executor* which sends HTTP requests to microservices. It is possible to execute requests also in case of authentication, credentials or tokens must be specified in the configuration file. This component provides, as output, all the request-response pairs. The responses are automatically evaluated based on the HTTP response code. We consider failures the *5xx* codes, as they point out the inability of the service to perform the request, due to an error condition, an unhandled exception, or in general an unexpected behaviour.

The *Analyzer* takes a set of request-response couples as input from the *Executor*, and provides a set of basic statistics as output. It provides the results of the test process as output to console and/or to file.

Table 4. Experimental subjects

MSA	Microservices	URIs	Methods	Lines of code
TrainTicket	34	1,152	1,442	20,015
SockShop	5	24	29	5,287
FTGO	7	16	16	14,976

Further details on the implementation and on usage can be found in the GitHub repository[3].

4 Experimental Comparison

4.1 Subjects

For the experimentation, we consider as subjects three open-source MSA, publicly available on GitHub. They are:

- TrainTicket[4]: a benchmark MSA (a booking system for train tickets) [18];
- SockShop[5]: the user-facing part of an online shop that sells socks;
- FTGO[6]: the Richardson's book sample MSA [19].

Table 4 lists their characteristics.

4.2 Experiments

To investigate the ability of the tools in generating effective test cases, we consider the following two scenarios:

- Scenario 1 (functional testing): *valid test cases*;
- Scenario 2 (functional and robustness testing) *valid* and *invalid test cases.*

Scenario 1 is meant to test the MSA behavior with inputs complying to the API. Here we compare EvoMaster and uTest; additionally, we consider as baseline the generation of a single test per method with randomly chosen valid input values (in the example in Fig. 1, this corresponds to the leaf at extreme left).

Scenario 2 mixes both valid and invalid inputs; the latter emulate a robustness testing scenario, where input specifications are intentionally violated for instance to verify return of proper status codes, or to check how the MSA reacts to unexpected inputs. For this scenario we compare RestTestGen, bBOXRT, RESTler and uTest (EvoMaster does not generate invalid inputs). The *baseline* in this Scenario is the generation of two tests per method, one with all parameter values chosen from valid classes and one with all values from invalid classes (this is a sort of *1-way* testing).

The compared tools were configured, when possible, with the values that were shown in the literature to yield the best performance, otherwise with default values. We ran tests 10 times for each microservice of the three subjects.

[3] https://github.com/uDEVOPS2020/uTest.
[4] https://github.com/FudanSELab/train-ticket.
[5] https://github.com/microservices-demo/microservices-demo.
[6] https://github.com/microservices-patterns/ftgo-application.

Table 5. Coverage metrics (as defined in ref. [12])

Coverage metric	Description
Path	Ratio of the number of tested paths to the total number of paths documented in the OpenAPI specification. 100% path coverage if its tests send at least one request directed to each path of the API
Operation	Ratio of the number of tested operations to the total number of operations described in the OpenAPI specification. 100% operation coverage if there exists at least one request directed to each path for all documented HTTP methods.
Parameter	Ratio of the number of input parameters used by test cases to the total number of parameters documented in the OpenAPI specification. 100% parameter coverage if all input parameters of all operations are included in requests at least once.
Parameter value	Ratio of the number of the exercised parameter values to the total number of values that parameters can assume according to the OpenAPI specification. Applicable only to domain-limited parameters (es. boolean, enum)
Request content-type	Ratio of the number of tested content-types to the total number of accepted content-types as per the OpenAPI specification. 100% request content-type coverage if there exists at least a test request for each accepted content-type.
Status code class	A test suite reaches 100% status code class coverage when it is able to trigger both correct and erroneous status codes. If it triggers only status codes belonging to the same class (either correct or erroneous), coverage equals 50%. 2XX class represents a correct execution and 4XX and 5XX classes represent an erroneous execution.
Status code	Ratio of the number of obtained status codes to the total number of status codes documented in the OpenAPI specification, for each operation. 100% status code coverage if, for each operation, all status codes are tested.
Response content-type	Ratio of the number of obtained content-types to the total number of response content-types as per the OpenAPI specification. 100% response content-type coverage if there exists at least one test response whose body matches each documented content-type, for each operation

4.3 Metrics

We adopt the coverage metrics defined by Martin-Lopez *et al.* [7], used also in ref. [12]. The definitions are provided in Table 5. The first five metrics concern the goodness of the generation with respect to the input specification (input metrics), while the last three are computed on response codes (output metrics). Coverage metrics are computed with the tools `Burp Suite` [20] and `Restats` [21]. `Burp Suite` logs each request-response pair; we export logs and compute metrics with `Restats`. This is the same process adopted in ref. [12].

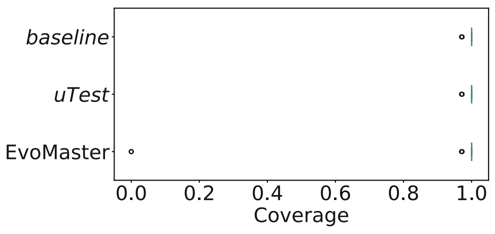
In addition with respect to [12], we compare tools in terms of cost, namely the *average number of executed tests*, and *average failure rate* (average number of failures exposed by executed tests over all microservices).

5 Results

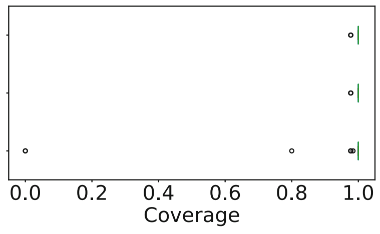
5.1 Scenario 1: Tests with Valid Input

We compare the effectiveness of uTest and EvoMaster, when testing MSA microservices as independent services, with inputs complying to their API.

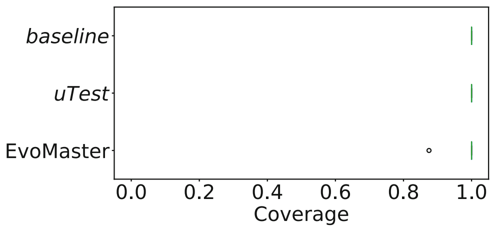
Figure 3 shows boxplots for the eight metrics. Values are averaged over repetitions for all microservices of all subjects. (For all metrics, the standard deviation over repetitions is less than 1.2% of the mean.) The results point out comparable values of uTest and EvoMaster with respect to the *baseline*, outperforming it only in *parameter value* coverage. uTest achieves slightly better results for all metrics, except for *response content-type*.



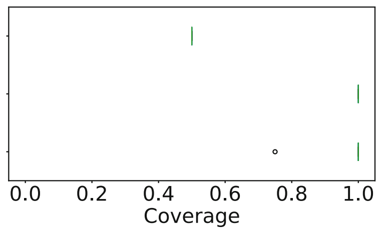
(a) Path



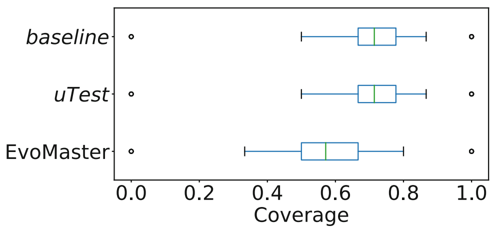
(b) Operation



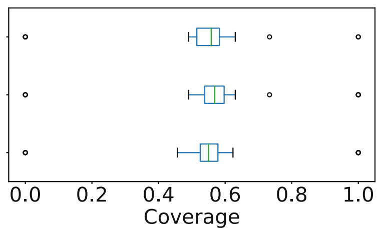
(c) Parameter



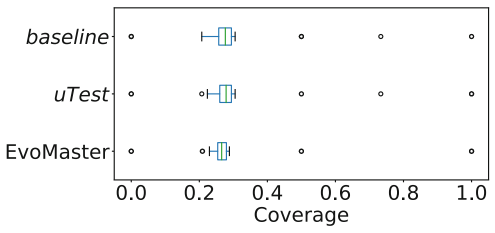
(d) Parameter value



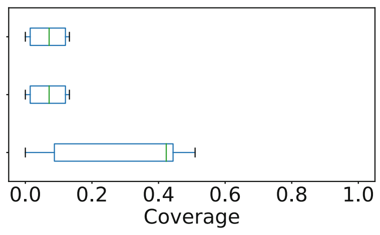
(e) Request content-type



(f) Status code class



(g) Status code



(h) Response content-type

Fig. 3. Scenario 1 (valid test cases): coverage

Figure 4a shows the average failure rate. uTest achieves a slightly better rate than EvoMaster. As for cost, Fig. 4b shows that the average number of tests executed by EvoMaster is one order of magnitude greater than the other two. Among these, uTest generates a test suite approximately three times bigger than the baseline, but detecting almost six times more failures.

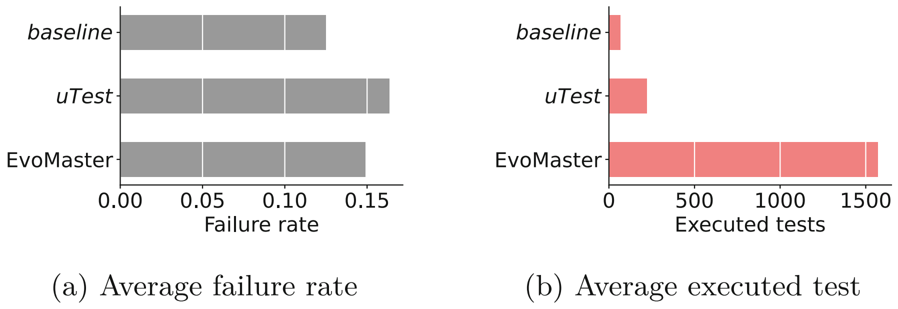


Fig. 4. Scenario 1 (valid test cases): average failure rate and executed tests

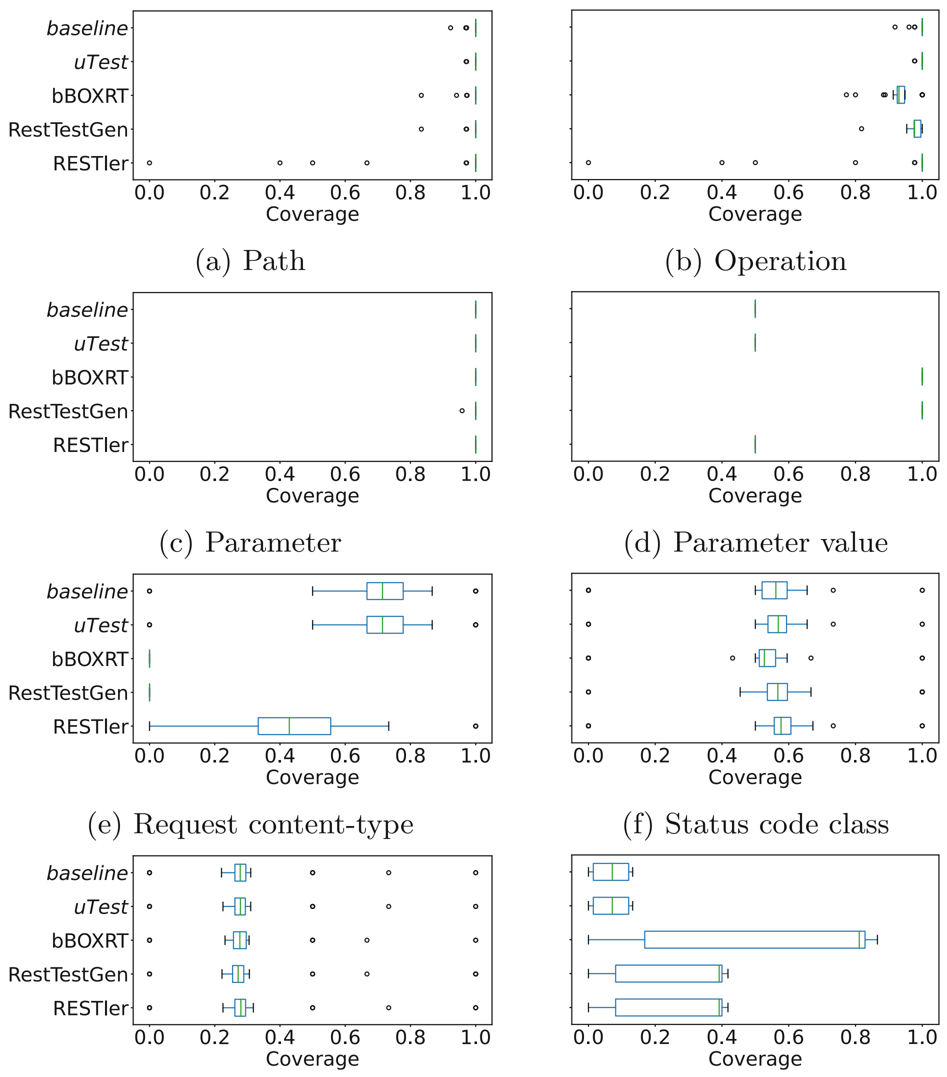
5.2 Scenario 2: Tests with Valid and Invalid Input

We evaluate coverage, average failure rate, and cost of test suites containing both *valid* and *invalid test cases* genereted by RestTestGen, bBOXRT, RESTler, uTest, and compare them to the baseline.

Figure 5 shows boxplots of the average coverage. The tools reveal comparable performance, except for *parameter value coverage* (Fig. 3d): in this case, RestTestGen and bBOXRT show full coverage, whereas RESTler, uTest and *baseline* test boolean values with either true or false, and coverage equals 0.5.

As for failure rate, Fig. 6a shows that uTest and *baseline* achieve higher values. As for cost, Fig. 6b shows that the average number of tests needed by bBOXRT, RestTestGen and RESTler is an order of magnitude higher than uTest and the baseline. Similarly to Scenario 1, the pairwise strategy generates a test suite approximately three times bigger than the baseline, but detecting almost five times more failures. In addition, the higher average failure rate and the comparable values of coverage metrics of the combinatorial approach with respect to the other tools are confirmed. The results achieved by the baseline are particularly interesting, as it generates only a single test case for each method in API (greater cost/benefit ratio compared to stateful approaches).

Failures reported by tests execution need to be investigated by debuggers. Microservices in an MSA may originate more complex invocation paths than single RESTful services, making the analysis of failure causes challenging. As for most distributed systems, observability is key to debug and troubleshooting MSA [22,23]. Solutions exist to monitor systems at the microservice level [24], or to mock dependencies in the MSA under test [2].



(a) Path (b) Operation (c) Parameter (d) Parameter value (e) Request content-type (f) Status code class (g) Status code (h) Response content-type

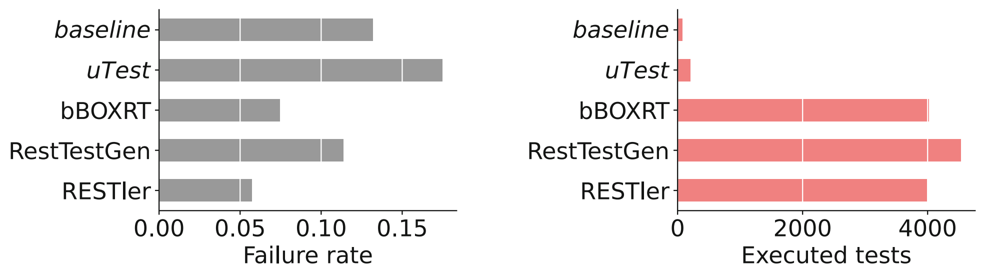
Fig. 5. Scenario 2 (valid and invalid test cases): coverage

6 Threats to Validity

Concerning internal validity, despite our best efforts, the presence of defects in the uTest prototype cannot be ruled out and might skew the results. The prototype is made anonymously available for reproducibility and repeatability.

External validity is threatened by the case studies adopted. While the three MSA used for experiments are the most used ones in the literature, they are open-source projects far from realistic MSA. However, finding real-world MSA for scientific experiments is a recognized problem [25].

An additional remark in respect to output coverage for the case studies is in order. Stateful tools generate tests trying to cover operation and/or parameter dependencies. When more dependencies are covered, higher HTTP *status code coverage* values can be achieved, because of $2xx$ codes that may not been covered



(a) Average failure rate (b) Average executed test

Fig. 6. Scenario 2 (valid and invalid test cases): average failure rate and tests

otherwise. In our experiments, output coverage values are close to the minimum (e.g., 50% *status code class*, meaning that on average all tests generated per method were able to return only codes of $2xx$ or $4-5xx$ classes). This points out the difficulties of tools in finding realistic *valid* values and/or operation sequences to cover all successful responses – the majority of codes returned belong to the 4xx class. For an MSA, dependencies are challenging to cover with a black-box approach that tests microservices independently. We noted that most tools reach the maximum execution time configured, generating plenty of worthless tests. A contributing cause might be the poor specifications for the case studies, which might affect the trustworthiness of results.

7 Conclusions

We presented an experimental comparison of techniques/tools, borrowed from black-box RESTful Web Services testing, for automatic test case generation for microservices in an MSA, and compared them to a newly proposed own combinatorial strategy. This is a more comprehensive comparison of black-box tools than past studies, as it includes the state-of-the-art tool EvoMaster, and for the first time it includes a stateless pairwise technique.

The experiments show that specification-based techniques can support MSA testers both in functional testing (e.g., for system and acceptance testing) and in robustness testing (e.g., testing fault tolerance means, error handling), indeed alleviating the burden of manually writing tests. Although results may be threatened by actual representativeness of the three case studies, the proposed combinatorial approach demonstrated to achieve coverage comparable to stateful techniques, while requiring an order of magnitude lower number of test cases. However, uTest like existing tools reach low values of output coverage.

It might be argued that applying specification-based test generation techniques (often, conceived for RESTful Web Services) to the many individual microservices of an MSA may be insufficient to cover complex interaction patterns among them. Microservices have probably better not be tested independently, but considering the entire architecture they are part of. In future work, we will investigate grey-box strategies to test microservice architectures.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 871342.

References

1. Lewis, J., Fowler, M.: Microservices - a definition of this new architectural term. http://martinfowler.com/articles/microservices.html (2014)
2. Arcuri, A.: Automated black- and white-box testing of RESTful APIs with EvoMaster. IEEE Softw. **38**(3), 72–78 (2021)
3. Ma, S., Fan, C., Chuang, Y., Lee, W., Lee, S., Hsueh, N.: Using service dependency graph to analyze and test microservices. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), vol. 02, pp. 81–86 (2018)
4. Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., Ceccato, M.: Automated black-box testing of nominal and error scenarios in RESTful APIs. Softw. Test. Verification Reliab. **32**, e1808 (2022)
5. Atlidakis, V., Godefroid, P., Polishchuk, M.: RESTler: Stateful rest API fuzzing. In: IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 748–758. IEEE (2019)
6. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: black-box constraint-based testing of RESTful web APIs. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) ICSOC 2020. LNCS, vol. 12571, pp. 459–475. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65310-1_33
7. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: Test coverage criteria for RESTful web APIs. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST), pp. 15–21. ACM (2019)
8. Laranjeiro, N., Agnelo, J., Bernardino, J.: A black box tool for robustness testing of rest services. IEEE Access **9**, 24738–24754 (2021)
9. Ed-douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic generation of test cases for REST APIs: a specification-based approach. In: IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), pp. 181–190. IEEE (2018)
10. Karlsson, S., Čaušević, A., Sundmark. D.: QuickREST: property-based test generation of OpenAPI-described RESTful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 131–141. IEEE (2020)
11. Bania, O., Florea, D., Gyalai, R., Curiac, D.: Automated specification-based testing of REST APIs. Sensors **21**(16), 5375 (2021)
12. Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M.: 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), pp. 226–236. IEEE (2021)
13. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Software **13**(5), 83–88 (1996)
14. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 1–29 (2011)

15. Hu, L., Wong, W.E., Kuhn, D.R., Kacker, R.N.: How does combinatorial testing perform in the real world: an empirical study. Empirical Software Eng. **25**(4), 2661–2693 (2020). https://doi.org/10.1007/s10664-019-09799-2
16. Pezzè, M., Young, M.: Software Testing and Analysis - Process, Principles and Techniques. Wiley, Hoboken (2007)
17. Bertolino, A., De Angelis, G., Guerriero, A., Miranda, B., Pietrantuono, R.. Russo, S.: DevOpRET: continuous reliability testing in DevOps. J. Softw. Evol. Process. e2298 (2020). smr.2298
18. Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D.: Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. IEEE Trans. Softw. Eng. **47**(2), 243–260 (2021)
19. Richardson, C.: Microservices Patterns. Manning Publications, Shelter Island (2018)
20. Portswigger: burp suite. https://portswigger.net/burp
21. Corradini, D., Zampieri, A.. Pasqua, M., Ceccato, M.: Restats: a test coverage tool for RESTful APIs. CoRR, abs/2108.08209 (2021)
22. Indrasiri, K., Siriwardena, P.: Microservices for the Enterprise: Designing, Developing, and Deploying, 1st edn. Apress, USA (2018)
23. Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G.: Design, monitoring, and testing of microservices systems: the practitioners' perspective. J. Syst. Softw. **182**, 111061 (2021)
24. Cinque, M., Della Corte, R., Pecchia, A.: Microservices monitoring with event logs and black box execution tracing. IEEE Trans. Serv. Comput. **15**(1), 294–307 (2022)
25. Zhou, X., et al.: Poster: benchmarking microservice systems for software engineering research. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 323–324. IEEE (2018)