Combinatorial Testing of RESTful APIs

Huayao Wu
State Key Laboratory for Novel Software Technology
Nanjing University, China
hywu@nju.edu.cn

Lixin Xu
State Key Laboratory for Novel Software Technology
Nanjing University, China
lxxu@smail.nju.edu.cn

Xintao Niu
State Key Laboratory for Novel Software Technology
Nanjing University, China
niuxintao@nju.edu.cn

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University, China
changhainie@nju.edu.cn

ABSTRACT

This paper presents RESTCT, a systematic and fully automatic approach that adopts Combinatorial Testing (CT) to test RESTful APIs. RESTCT is systematic in that it covers and tests not only the interactions of a certain number of operations in RESTful APIs, but also the interactions of particular input-parameters in every single operation. This is realised by a novel two-phase test case generation approach, which first generates a constrained sequence covering array to determine the execution orders of available operations, and then applies an adaptive strategy to generate and refine several constrained covering arrays to concretise input-parameters of each operation. RESTCT is also automatic in that its application relies on only a given Swagger specification of RESTful APIs. The creation of CT test models (especially, the inferring of dependency relationships in both operations and input-parameters), and the generation and execution of test cases are performed without any human intervention. Experimental results on 11 real-world RESTful APIs demonstrate the effectiveness and efficiency of RESTCT. In particular, RESTCT can find eight new bugs, where only one of them can be triggered by the state-of-the-art testing tool of RESTful APIs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

RESTful API, combinatorial testing, test case generation

ACM Reference Format:
Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510151

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510151

1 INTRODUCTION

Modern web applications are usually built as a composition of heterogeneous web services that interact with each other through web Application Programming Interfaces (APIs). The REpresentation State Transfer (REST) represents an architectural style for creating such APIs, and the APIs conforming to REST are thereby referred to as RESTful APIs [39]. The use of RESTful APIs can provide a uniform and flexible way to access and manipulate web resources via HTTP method operations, which has become the *de facto* standard for the design and implementation of modern web services [4, 5].

Given the increasing prevalence and widespread adoption of RESTful APIs, it is of great importance for these APIs to be thoroughly tested. Currently, web service developers usually use the Swagger specification (more formally, OpenAPI) to document RESTful APIs, in which the exact operations that can be performed, and the format of input and output parameters are explicitly described [9, 42]. The availability of this common specification has then motivated researchers to develop black-box testing approaches of RESTful APIs [9, 11, 14, 18, 22, 27, 33, 35, 41–43]. However, it remains challenging to design test cases that can effectively exercise service behaviours, as the available operations (which can be executed in many different orders) and their associated input-parameters (which can be concretised using many different values) typically constitute a potentially huge input space that is unlikely to be exhaustively explored.

Apart from the many possible ways to execute RESTful APIs, the dependency relationships, i.e., *constraints*, in both operations and input-parameters also contribute to the complexity of test case generation. A constraint in RESTful APIs might prohibit specific execution orders of operations (e.g., data received in the response of operation X is necessary for executing another operation Y) [9, 42]; it might also restrict the ways in which input-parameters of a single operation can be combined and concretised (e.g., if input-parameter X is set, then input-parameter Y must be set too) [30, 31]. Clearly, any test case generation approach that cannot properly handle such constraints will lead to invalid HTTP requests, and thereby, reduce testing effectiveness.

The huge and constrained input space of RESTful APIs especially asks for a smart approach to generate test cases that are as cost-effective as possible. Currently, despite that approaches like model-based [18, 22] and fuzzing [8, 9] are developed, they are not in a position that takes "combinations" of operations (and also, input-parameters) into account. In particular, a primary focus of existing approaches is on the construction of constraints-satisfying

operation sequences [9, 42]. This helps to reach and exercise every "testable" operation, but it is unlikely to effectively trigger failures due to certain execution orders of operations (and also, certain combinations of input-parameter values).

Moreover, for each given operation, the values of its input-parameters should also be carefully determined to produce concrete and valid HTTP requests. Existing studies [1, 9, 20, 22, 42] have presented a variety of value rendering strategies (e.g., using dynamic objects, static directory, examples, random values, and knowledge bases), but unfortunately, they are usually unaware of the constraints between input-parameters. In fact, the support of inter-parameter constraints is important, because such constraints are frequently observed in real-world RESTful APIs [31]. Although testers can choose to use formal languages to re-write constraints to fulfil automatic constraints handling [29, 30, 33], it could be more desirable to automatically infer such constraints from natural language descriptions of the Swagger specification.

In this study, we make use of Combinatorial Testing (CT) to devise a systematic and fully automatic approach, RestCT, to test RESTful APIs. Our aim is to cover and test the interactions of a certain number of operations (and also, a certain number of input-parameters) described in the specification, so that the service behaviours can be systematically exercised. Meanwhile, we also seek to automatically model the input space of RESTful APIs (including the inferring of constraints in operations and input-parameters), so that the complete testing process can be fully automated.

To this end, RestCT adapts the concepts of both sequence and classical covering arrays to implement a novel two-phase test case generation approach. Specifically, given a Swagger specification as the input, RestCT first generates a constrained sequence covering array to determine the execution orders of operations, in which the hierarchical relations of resources, and CRUD semantics (create, read, update, and delete) are used together to handle the constraints between operations. Next, for each operation sequence generated before, RestCT uses an adaptive strategy to generate several constrained covering arrays to concretise input-parameters of each operation (i.e., producing concrete HTTP requests). In this phase, RestCT relies on several heuristic strategies to determine value domains of input-parameters, and leverages natural language processing to implement a pattern-based approach for extracting inter-parameter constraints.

We have implemented RestCT as a testing tool[1], and carried out experiment on 11 real-world RESTful APIs of *GitLab* and *Bing Maps* to evaluate its effectiveness and efficiency. The experimental results reveal that RestCT can successfully test, on average, 56% of operations described in the Swagger specification, while this proportion is only 3% for the state-of-the-art testing tool, RESTler [9]. RestCT is also time efficient, as it spends no more than ten minutes in 55% of cases studied. In addition, we have further investigated the impact of coverage strengths on the performance of RestCT. We find that the coverage strength applied for generating operation sequences tends to have a great influence, and coverage strength two is the most cost-effective choice. Overall, RestCT detects eight new bugs of the subject APIs, where only one of them can be triggered by RESTler.

[1] https://github.com/GIST-NJU/RestCT

```yaml
paths:
  /users:
    post:
      parameters:
        - name: username
          in: body
          description: The name of the user.
          required: true
          schema:
            $ref: '#/definitions/UserName'
        - name: password
          in: body
          description: If random_password is False, password is required.
          required: false
          type: string
        - name: random_password
          in: body
          required: false
          type: boolean
      responses:
        '200':
          description: OK
definitions:
  UserName:
    type: object
    properties:
      FirstName:
        type: string
      LastName:
        type: string
```

Figure 1: An example of Swagger specification.

Summing up, the **main contributions** of this paper are two-folds: (1) We proposed RestCT, the first systematic and fully automatic approach that adopts combinatorial testing to test RESTful APIs; and (2) We presented detailed experimental results that evaluate the effectiveness and efficiency of RestCT on 11 real-world RESTful APIs. This study also extends the knowledge of both RESTful APIs and combinatorial testing literatures, as it not only explores a systematic strategy (i.e., CT) to test operations and input-parameters of RESTful APIs, but also provides an application of CT in which the test models can be automatically created without testers' manual efforts.

The rest of this paper is organised as follows: Section 2 covers the background on Swagger specification of RESTful APIs and combinatorial testing. Section 3 describes the detailed process of RestCT. Section 4 presents the experiment for evaluating RestCT. Section 5 discusses possible threats to validity. Section 6 discusses related works, and Section 7 concludes this paper.

2 BACKGROUND

We begin this study with background on specification of RESTful APIs, and basic concept of combinatorial testing.

2.1 Specification of RESTful APIs

Modern web services usually rely on RESTful APIs to expose their services to the Web. In this way, each service should provide its *resources* in a textual representation (typically, in JSON format), and allow these resources to be accessed and manipulated by a set of CRUD operations (create, read, update, and delete) [41].

In practice, a resource of RESTful APIs is generally identified by a Uniform Resource Locator (URL) over HTTP protocol. The CRUD operations are then realised by HTTP method operations (i.e., `POST`, `GET`, `PUT`, and `DELETE`). Accordingly, a client can send an HTTP

Table 1: A 2-way Constrained Covering Array

#	p_1	p_2	p_3	p_4
1	1	1	0	0
2	0	1	1	1
3	1	0	1	0
4	0	1	0	0
5	1	0	0	1

request to the service, in which a certain URL, an HTTP method, and corresponding input-parameter values should be specified. The service will then return a *response* with the required representation of the resource.

In order to document the necessary information for using RESTful APIs, a structured specification should be provided. Currently, the most widely recognised choice is the OpenAPI specification (it is also known as the Swagger specification for versions older than v3.0), which defines a standard to specify the exact syntax of all the available requests and expected responses of the APIs [9, 41, 42].

Figure 1, for example, gives a snippet of the Swagger specification (in YAML format). Here, the `paths` field describes a list of resources available (i.e., API endpoints), each of which corresponds to a set of HTTP methods that can be performed (e.g., the `POST` method is supported for the resource `/users`). For each HTTP method, the format of input and output parameters are described in the `parameters` and `responses` fields (e.g., the input-parameter `password` should take a string value; it is optional, and should be sent in the request body). In addition, some parameters might be associated with a `schema`, whose format is defined in the `definitions` field (e.g., the input-parameter `username` is associated with an object `UserName`, which consists of two parameters of the string type).

2.2 Combinatorial Testing

Combinatorial Testing (CT) is a popular testing approach to test interactions of parameters that influence the software's behaviour [24, 36]. The rational of this approach derives from the observation that many software failures are triggered by combinations of parameter values [25, 34]. Accordingly, CT employs a *covering array* as the test suite to systematically cover such combinations, hoping to achieve a good balance between test suite size and failure revelation effectiveness.

Assume that the behaviour of a system is influenced by n parameters, $P = \{p_1, p_2, \ldots, p_n\}$, and each parameter p_i can take discrete values from a finite set V_i, for $1 \leq i \leq n$. Such parameters P and their value domains $V = \{V_1, V_2, \ldots, V_n\}$ constitute a *test model* of CT, which depict the input space of the system. Then, a test case can be constructed by assigning to each parameter a specific value, and we refer to a combination of t parameter values as a *t-way combination*.

Definition 1 (Covering Array). *Given a set of parameters and their associated value domains, a t-way Covering Array (CA) is a set of test cases, in which every t-way combination is covered at least once. The value of t indicates the coverage strength of the array.*

In addition to the combination of parameter values, sometimes the behaviour of the system can be influenced by a set of n events, $E = \{o_1, o_2, \ldots, o_n\}$, that can be executed in different orders. In this case, the test model consists of a set of events E, and a test case indicates a sequence of $k \leq n$ distinct events in E (that is, the test cases can be of different lengths[2]). We refer to a permutation of t events as a *t-way sequence*.

Table 2: A 3-way Constrained Sequence Covering Array

#	Test Cases	Unique 3-way Sequences Covered
1	o_1, o_4, o_3, o_2	(o_1, o_4, o_3), (o_1, o_4, o_2), (o_1, o_3, o_2) (o_4, o_3, o_2)
2	o_3, o_1, o_4, o_2	(o_3, o_1, o_4), (o_3, o_1, o_2), (o_3, o_4, o_2)
3	o_1, o_3, o_4	(o_1, o_3, o_4)

Definition 2 (Sequence Covering Array). *Given a set of n events, a t-way Sequence Covering Array (SCA) is a set of event sequences, in which every t-way sequence is covered at lease once (the t events of a t-way sequence are not required to be adjacent in an event sequence).*

The above mentioned covering arrays assume that all possible t-way combinations, or t-way sequences, are feasible and should thus be covered. However, this might be unrealistic due to the presence of constraints [37, 45]. A *constraint* of the system depicts the dependency relationships between software parameters, or events. Any test case that violates constraints is considered invalid, and should thus be excluded from the final test suite.

To cater for the constraints in CT, the definition of a covering array, or a sequence covering array, should be extended to that of a *constrained covering array*, or a *constrained sequence covering array*. Specifically, given a set of constraints C, each test case in a constrained CA (or a constrained SCA) should be C-satisfying, and every C-satisfying t-way combination (or t-way sequence) should be covered at least once in the array [45]. Accordingly, the CT test model should specify not only the set of parameters and their value domains (or the set of events), but also the constraints in the system.

For example, Table 1 gives a 2-way constrained CA of the test model with $P = \{p_1, p_2, p_3, p_4\}$, $V_1 = V_2 = V_3 = V_4 = \{0, 1\}$, and one constraint: if $p_1 = 0$ then $p_2 = 1$. In addition, Table 2 gives a 3-way constrained SCA of the test model with $E = \{o_1, o_2, o_3, o_4\}$, and two constraints: 1) o_4 can only be executed after o_1; and 2) no event can be executed after o_2.

3 THE RESTCT APPROACH

Given a Swagger specification as the input, an effective black-box testing approach of RESTful APIs should generate not only appropriate execution orders of available operations, but also valid value assignments of each operation's input-parameters. RestCT adopts combinatorial testing to address the above two problems. Its overall workflow involves two main phases: (1) *Operation Sequence Generation*, which seeks to model the input space of available operations, and construct a sequence covering array as a representative set of operation sequences; and (2) *Input-Parameter Value Rendering*, which seeks to model the input space of input-parameters of each

[2]We note that this definition of test case is different from that of traditional sequence covering array [23], where all test cases should be of the equal length.

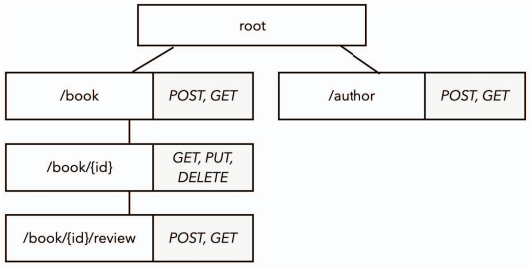

Figure 2: An example of hierarchical relations of resources.

operation, and sample representative value assignments via several covering arrays to produce concrete HTTP requests.

3.1 Operation Sequence Generation

In this study, we refer to an *operation* as a combination of a resource and an associated HTTP method (e.g., `POST /users` in Figure 1). RestCT will first parse the Swagger specification to find the set of available operations and their dependency relationships, and then generate a t_s-way constrained sequence covering array (i.e., a set of operation sequences) to determine the execution orders of these operations.

3.1.1 Handling Constraints between Operations. Given a Swagger specification, the set of all the available operations, $O = \{o_1, o_2, \ldots, o_m\}$, can be directly extracted from the `paths` field of the specification. The challenge here is to appropriately cater for the constraints between these operations [9, 42].

RestCT relies on the hierarchical relations of resources and CRUD semantics to handle constraints in operation sequence generation. In RESTful APIs, the hierarchical relations of resources are specified by the forward slash "/" in resources' URLs. Such relations can also be depicted using a tree structure, as the example shown in Figure 2 (including the HTTP methods that can be performed on each resource). In this case, `/book/{id}` is a child node of `/book`, because it is a direct sub-resource of `/book`; by contrast, there is no direct relation between `/book` and `/author`.

According to the CRUD semantics, a resource (and its all sub-resources) should not be accessed before its creation, nor after its deletion. Hence, a t-way sequence, $s = (o_{k_1}, o_{k_2}, \ldots, o_{k_t})$, is considered *constraints-satisfying* if for any resource `X` involved in s, none of the operations associated with `X` and `X`'s sub-resources appears before `POST X`, or after `DELETE X`. For example, according to Figure 2, (`GET /book/{id}`, `GET /book`) is a constraints-satisfying 2-way sequence, because no operation appears before `POST`, or after `DELETE` operations. The 2-way sequence (`GET /book/{id}`, `POST /book`) is not constraints-satisfying, because the operation associated with sub-resources of `/book`, i.e., `GET /book/{id}`, appears before `POST /book`. Similarly, (`DELETE /book/{id}`, `GET /book/{id}`) also violates the constraints.

Once a constraints-satisfying sequence can be decided, we can then determine whether an operation o_i can be used to extend an existing operation sequence of length t, $s = (o_{k_1}, o_{k_2}, \ldots, o_{k_t})$. In particular, we refer to o_i as a *feasible* operation for extending

Algorithm 1 The Generation of Operation Sequences

Input: a set of operations O, and coverage strength t_s
Output: a set of operation sequences S that satisfies t_s-way sequence coverage

```
1:  S = ∅
2:  U = set of constraints-satisfying t_s-way sequences of O
3:  while U ≠ ∅ do
4:      s = an empty sequence
5:      while True do
6:          A = candidate_operations(s, O)
7:          if A ≠ ∅ then
8:              select a ∈ A that contributes the most to t_s-way sequence coverage (ties break by random)
9:              if a cannot help cover any t_s-way sequence then
10:                 break
11:             end if
12:             a' = predecessor_operations(s, a)
13:             append a' and a to s
14:         else
15:             break
16:         end if
17:     end while
18:     add s into S, and remove t_s-way sequences that are covered by s from U
19: end while
20: return S
```

s, if the $(t+1)$-way sequence constructed by appending o_i to s, $(o_{k_1}, o_{k_2}, \ldots, o_{k_t}, o_i)$, is constraints-satisfying.

Here, an operation is feasible for extending s does not necessarily mean that this operation can be directly executed after s. For example, the operation `GET /book/{id}` is feasible for extending s = (`POST /author`), but it cannot be directly executed, because the resource `/book` is yet-to-be created. In this case, the CRUD semantics should be further accounted for, to determine the additional predecessor operations that are required to be executed earlier than executing a feasible operation (e.g., `POST /book` is the predecessor operation for executing `GET /book/{id}` in the above example).

We note that the constraints handling strategy presented above differs from previous strategies that rely on the common field in both responses and requests [9, 42]. Given a particular operation sequence, the previous strategies [9, 42] primarily focus on operations that can be directly executed (basically, every dynamic object that is required should have been produced). By contrast, our strategy seeks to find operations that have the chance to be executed in the future (even though the resource required is yet-to-be created), which is more helpful in generating operation sequences of high sequence coverage.

3.1.2 Generating Operation Sequences. With the above constraints handling strategy, RestCT will then apply a greedy algorithm, i.e., Algorithm 1, to generate a t_s-way constrained sequence covering array to cover interactions of any t_s operations. This algorithm first determines the set of constraints-satisfying t_s-way sequences, U, of the operations under test (Line 2). Then, it generates an operation sequence that covers the most previously uncovered t_s-way

sequences at each time, until all t_s-way sequences in U are covered (Lines 3-19).

To generate each operation sequence, Algorithm 1 will start with an empty sequence s, and then extend s iteratively (Lines 5-17). The *candidate_operations*(s, O) function (Line 6) will return a set of candidate operations that are not included in s (i.e, each operation in O occurs at most once in s), and at the same time, are feasible for extending s (i.e., appending the operation to s does not violate the constraints). At this step, if no candidate operation can be found (i.e., $A = \emptyset$), then the iteration terminates (Line 15), and the sequence s constructed so far is added into the final set of operation sequences S. Otherwise, the algorithm will evaluate the number of t_s-way sequences that can be potentially covered by appending every candidate operation in A to s, and select the best operation $a \in A$ for extension (Lines 8-13).

Note that the best operation a selected here might not be able to be directly executed after s (as explained in Section 3.1.1). So, the algorithm will additionally account for the CRUD semantics to determine the predecessor operations, a', that should be additionally executed before executing a (i.e., the *predecessor_operations*(s, a) function in Line 12; $a' = None$ if no predecessor is required). Finally, the operations in a' and a will be appended to the current sequence s (Line 13), and the algorithm will repeat the above process to find the next best operation for extension.

3.2 Input-Parameter Value Rendering

Once the set of operation sequences is produced, RESTCT will then generate concrete HTTP requests to execute operations in these sequences. The aim of input-parameter value rendering is to concretise input-parameters with as many valid value assignments as possible, and at the same time, cover interactions of particular input-parameters. To this end, RESTCT will use an adaptive strategy to generate constrained covering arrays, in which the CT test models are dynamically created and modified according to both specification and test execution results.

3.2.1 Identifying Input-Parameters and Their Value Domains. Given each operation, RESTCT will extract all applicable input-parameters from the `parameters` field of the Swagger specification. Such parameters can be located in `Path`, `Query`, `Header`, `Form`, and `Body`, and they can be of various types. Currently, RESTCT supports parameters of the `String`, `Integer`, `Number`, `Boolean`, `Array`, and `Object` types. Each parameter of the non-`Object` type exactly indicates a unique input-parameter of the operation (in this study, we let the length of the array be one). While for the `Object`-typed parameters, their formats are defined as `schemas`, and each non-`Object`-typed parameter involved in the schema is extracted as a unique input-parameter (e.g., for the schema `UserName` in Figure 1, two input-parameters of string type, `FirstName` and `LastName`, will be extracted).

For each input-parameter identified, RESTCT will use one of the following strategies to determine its value domains (i.e., the set of values that the input-parameter can take):

- **DYNAMIC**: using resources dynamically created in a prior operations. It takes run-time information of the service into account, which is often necessary for constructing valid HTTP requests [20]. Specifically, given an input-parameter p and a set of previously received responses R, this strategy will use a heuristic name matching approach [42] to search for output-parameters in R to find those that have the most similar names with p (due to the naming inconsistency between output and input-parameters). Then, the values of such output-parameters will be used as p's value domain.
- **SPECIFICATION**: using values described in the Swagger specification. It relies on values provided by developers, which could exploit domain-specific information of the service under test [20]. Specifically, if an input-parameter p is associated with an `enum` or `default` field, the corresponding values are directly used as its value domain. Otherwise, this strategy will search the `example` fields of all input-parameters that have the same name with p, and select at most two example values (at random) as p's value domain.
- **SUCCESS**: using values that appear in previous successful requests (i.e., returning HTTP status codes of 200 range), which relies on the test execution history to re-execute an operation. Here, if the previous value is determined by the *RANDOM* strategy, this value will be mutated to increase the diversity of test inputs (e.g., avoiding to create resources of the same name).
- **RANDOM**: generating three different values at random as the value domain. This strategy will be used if no value can be identified by the above strategies.

```
"pattern_example": {
  "pattern": [
    {"LEMMA": "if"},
    {"ENT_TYPE": "PARAM"},
    {"LEMMA": "be"},
    {"ENT_TYPE": "VALUE"},
    {"IS_PUNCT": true},
    {"ENT_TYPE": "PARAM"},
    {"LEMMA": "be"},
    {"LEMMA": {"IN": ["specify", "require", "enable",
                      "present", "include", "provide"]}}
  ],
  "constraint": ["(AA = BB) => (CC != 'None')"],
  "example": "if imagerySet is Birdseye, orientation is required"
}
```

Figure 3: A pattern of constraint between input-parameters.

Here, because input-parameters that are located in `Path` (i.e., the parameter value is part of the operation's URL) must always have a value and the value is usually dependent on previous responses, their value domains will be determined by the *DYNAMIC* strategy only. While for the other parameters, RESTCT will try all the above strategies in a decreasing priority, until the value domain is determined. In addition, according to the `required` field of the specification, each input-parameter is either *required* or *optional* for constructing an HTTP request. For the optional input-parameters, a *None* value will be additionally added into its value domain, indicating that no value is assigned to this parameter.

3.2.2 Inferring Constraints. Similar to the constraints between operations, there are also inter-parameter dependencies that might restrict the way in which specific input-parameters of an operation can be concretised [31]. Currently, the Swagger specification just

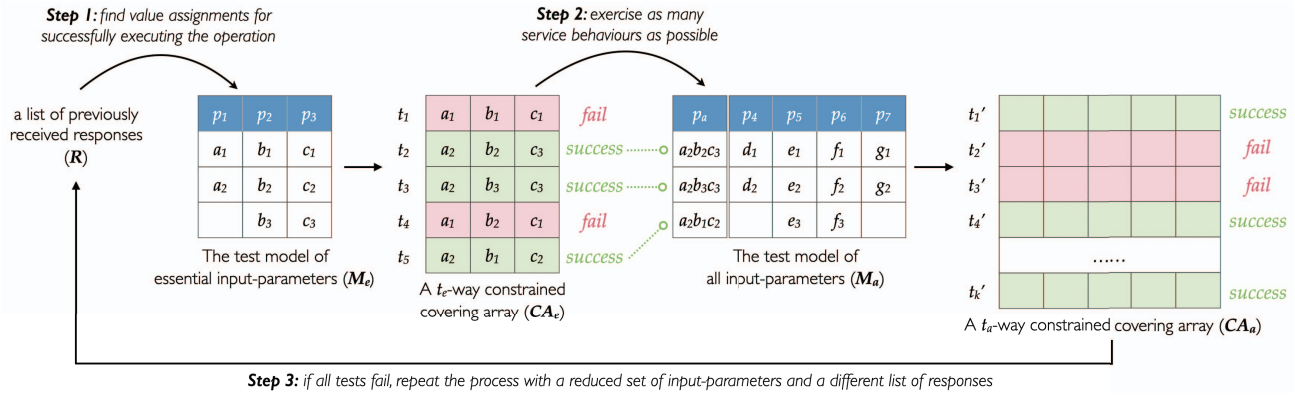

Figure 4: The adaptive strategy to generate concrete HTTP requests to execute each single operation.

encourages developers to use natural language to describe constraints between input-parameters as a part of the `description` field [30, 31] (e.g., from Figure 1, if `random_password` is `False`, then `password` must take a value). To automatically infer such constraints from the informal description, RESTCT leverages *spaCy*[3], an open-source library for natural language processing, to implement a pattern-based approach.

Figure 3, for example, gives a pattern used in RESTCT, which describes the constraint "*if PARAM_A is VALUE_B, PARAM_C is required*". Here, the `pattern` field gives the matching rule, which is encoded as a combination of tokens (similar to regular expressions, but with more flexibility in matching natural language). In this case, the first token matches the exact word *if*, the second token matches an entity (that is, an input-parameter name, or a specific input-parameter value), and the fifth token matches any punctuation[4]. This matching rule will then be used by the pattern-based matching engine of *spaCy* to extract constraint-describing sentences from a given text. In addition, the `constraint` field in Figure 3 gives the structured representation of the constraint described, where *AA*, *BB*, and *CC* indicate the first, second, and third entity involved in the sentence, respectively. Once a constraint-describing sentence is extracted, the corresponding input-parameter names and values (i.e., entities) will then be reorganised according to this representation to produce the final constraint.

In this study, we rely on an exiting catalogue of inter-parameter dependencies [31] to manually create a set of 23 patterns for RESTCT. This catalogue classifies constraints observed in 40 real-world RESTful APIs into seven types, and provides the most frequently used linguistic structures that describe constraints of each type (each constraint type can be described in different ways). We carefully studied these structures, and tried to develop patterns correspondingly. Some structures were excluded because they cannot be directly transformed into the format required by the tool that we use to generate covering arrays. Finally, 23 patterns were obtained, which cover all linguistic structures involving two parameters (about 86% of all structures available) and six of the seven constraint types.

[3] https://spacy.io
[4] A detailed explanation of token attributes can be found at https://spacy.io/usage/rule-based-matching.

3.2.3 Generating Concrete HTTP Requests. With the above approaches to identify input-parameters and infer constraints, RESTCT will then utilise an adaptive strategy, as illustrated in Figure 4, to generate concrete HTTP requests to execute operations in the given operation sequence one after another.

Suppose that the operation sequence is $s = (o_1, o_2, o_3, o_4)$, and RESTCT is going to execute the third operation o_3. As RESTCT typically generates multiple HTTP requests for each operation (in this case, o_1 and o_2), there is a need to determine which exact responses of o_1 and o_2 should be used to identify the value domains of o_3's input-parameters (that is, the *DYNAMIC* strategy as explained in Section 3.2.1). To this end, RESTCT will memorise a set of response chains, $\mathcal{R}$, in which each response chain is a list of previously received successful responses (i.e., the HTTP status code received is in 200 range). For example, assuming that three HTTP requests are generated and successfully executed for o_1, leading to three successful responses $r_{1,1}$, $r_{1,2}$, and $r_{1,3}$ (here, $r_{i,j}$ indicates the j-th response received when executing operation o_i). While for o_2, all subsequent HTTP requests generated according to $r_{1,1}$ and $r_{1,3}$ fail, and two HTTP requests generated after $r_{1,2}$ success, leading to two successful responses $r_{2,1}$ and $r_{2,2}$. At this point, the set of response chains is $\mathcal{R} = \{r_{1,1}, r_{1,2} - r_{2,1}, r_{1,2} - r_{2,2}, r_{1,3}\}$, and RESTCT will base on the longest response chain $R \in \mathcal{R}$ (ties break by random) to generate value assignments for executing the next operation. Assuming that $R = r_{1,2} - r_{2,1}$ is selected in this case. This indicates that the *DYNAMIC* strategy will use the values received in $r_{1,2}$ and $r_{2,1}$ to determine the value domains of o_3's input-parameters.

On the basis of the previously received responses R, RESTCT will then utilise several constrained covering arrays to generate concrete HTTP requests. The core idea is to ensure the successful execution of the target operation first (note that the input-parameter values identified and constraints extracted before might be incorrect), and then exercise as many service behaviours as possible. To this end, RESTCT will explore the input space governed by essential input-parameters first (Step 1 in Figure 4), and then grow this input space to include all applicable input-parameters (Step 2 in Figure 4).

Specifically, in Step 1, RESTCT creates a CT test model, M_e, that contains essential input-parameters only (including all required parameters, and those optional parameters that have dependency

relationships on required parameters). It then uses a popular covering array generation tool, ACTS[5], to generate a t_e-way constrained covering array CA_e of M_e, and construct and send concrete HTTP requests according to CA_e. After this step, a set of successful requests (i.e., returning HTTP status codes of 200 range) is obtained and recorded (e.g., t_2, t_3, and t_5 in Figure 4).

Next, in Step 2, RESTCT extends M_e to another CT test model, M_a, to further include all applicable input-parameters. This extension is based on the test execution results of Step 1, where all essential input-parameters are combined as an abstract parameter p_a, and the particular value assignments that appear in previous successful requests are used as p_a's value domain (e.g., in Figure 4, p_a can take three values, which correspond to the value assignments in t_2, t_3, and t_5, respectively). Then, RESTCT uses the ACTS tool again to generate a t_a-way constrained covering array CA_a of M_a, and construct and send concrete HTTP requests accordingly.

At this moment, if at least one successful request is produced, RESTCT will append every successful response received in CA_e and CA_a to the current response chain R to update $\mathcal{R}$ (e.g., given two successful responses $r_{3,1}$ and $r_{3,2}$, $R = r_{1,2} - r_{2,1}$ will be replaced by two new chains $r_{1,2} - r_{2,1} - r_{3,1}$ and $r_{1,2} - r_{2,1} - r_{3,2}$). Otherwise, RESTCT will analyse the message received in failed responses to find input-parameters that cannot be appropriately concretised. It will ignore such input-parameters, and repeat the above process with a different response chain R (Step 3 in Figure 4), hoping to concretise input-parameters of this operation in a different way (in this study, the maximum number of repetition is set to three).

4 EXPERIMENT

This section describes the experiment performed to evaluate the effectiveness and efficiency of the RESTCT approach. In particular, we are interested in answering the following two research questions:

- **RQ$_1$** Is RESTCT a cost-effective approach when comparing with the state-of-the-art testing tool of RESTful APIs?
- **RQ$_2$** How do the coverage strengths applied impact the performance of RESTCT?

4.1 Subject APIs

In this study, we used 11 real-world RESTful APIs as the experimental subjects. Table 3 summarises the number of operations described in each subject's Swagger specification, as well as the average number of input-parameters that can be extracted in each of these operations (using the approach described in Section 3.2).

Specifically, the first six RESTful APIs come from *GitLab*, a popular open-source web service for hosting Git repositories. This service is selected because it contains complex APIs and has been used in previous studies [9]. The subject APIs represent groups of operations that are related to branches, commits, issues, groups, projects, and repositories functionalities of *GitLab*, and we followed the previous study [9] to test these individual groups separately. We note that *GitLab* does not provide official Swagger specification files for its current API version (v4). So we manually created the required specifications based on its latest online document[6] (we excluded APIs that are only available in the paid services). In addition, we relied on an existing Docker image[7] to deploy *GitLab*, and performed testing in a local environment (on a machine with Intel Xeon CPU 2.3GHz and 128GB memory).

Table 3: The Subject APIs Used in the Experiment

	APIs	# Operations	# Input-Parameters
GitLab	Branch	9	15.2
	Commit	15	11.3
	Groups	17	13.2
	Issues	27	11.3
	Project	31	12.5
	Repository	10	11.9
Bing Maps	Elevations	4	3.5
	Imagery	10	15.1
	Locations	5	7.6
	Route	14	15.2
	TimeZone	4	5

The remaining five RESTful APIs come from *Bing Maps*, a web mapping service developed by Microsoft. This service is additionally selected because it contains a large number of inter-parameter constraints [31]. Similarly, since there is no available Swagger specification files for these APIs, we manually created the specifications based on the latest online API document[8]. We note that the online document of *Bing Maps* classifies all available operations into seven API groups, and we excluded two groups that contain one operation only. In addition, because *Bing Maps* is not an open source project, we relied on its publicly hosted service to perform testing.

We note that the website https://apis.guru offers a repository of Swagger specifications, and some previous studies [42] have used this website as the resource of subject APIs. In this study, we did not directly select specifications from that website, because such specifications might be incorrect (in fact, a specification of *GitLab* is available there, but many input-parameters are missing), or inconsistency with the current services' versions (the remotely deployed services might be updated). Instead, we chose to create specifications ourselves to ensure the correctness and completeness of the inputs taken by the testing approaches studied.

4.2 Experimental Procedures

We chose to compare RESTCT with RESTler (v8.0.0)[9], a state-of-the-art black-box fuzzing tool for testing RESTful APIs, for answering the first research question. RESTler is originally developed to explore deep service states reachable only through specific operation sequences [9]. Currently, it is still under active development and has incorporated several techniques reported in recent studies [10, 20, 21]. We note that there are other studies [15, 18, 22, 27, 33, 42] that propose black-box testing approaches of RESTful APIs. These approaches were not included, because they cannot automatically handle the dependencies between operations

[5] https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software
[6] https://docs.gitlab.com/ee/api/ (accessed in April 2021)
[7] https://hub.docker.com/r/gitlab/gitlab-ce (tag of image: 13.10.3-ce.0)
[8] https://docs.microsoft.com/en-us/bingmaps/rest-services/ (accessed in May 2021)
[9] https://github.com/microsoft/restler-fuzzer

Table 4: The Number of Operation Sequences Exercised (*Seq*) and Their Average Length (*Len*), Proportion of 1-way and 2-way Sequences that are Actually Tested ($C_{1\text{-}way}$ and $C_{2\text{-}way}$), Number of Unique Bugs Detected (*Bug*), Number of HTTP Requests Executed (*Total*), and Test Execution Cost (*Cost*, in minutes) of Different Testing Approaches

	RESTler							RESTCT						
	Seq	*Len*	$C_{1\text{-}way}$	$C_{2\text{-}way}$	*Bug*	*Total*	*Cost*	*Seq*	*Len*	$C_{1\text{-}way}$	$C_{2\text{-}way}$	*Bug*	*Total*	*Cost*
Branch	2	1	0	0	0	580	23.3	6.4	7.1	0.84	0.51	0	859.4	2.3
Commit	2	1	0	0	0	580	23.4	8.4	11.2	0.24	0.03	0	2851.8	4.4
Groups	2	1	0.06	0	1	41	0.5	10.2	11.4	0.12	0.01	1	3667.8	5.2
Issues	3	1	0	0	0	614.4	60.6	16.2	16.3	0.26	0.06	0	13287.4	27
Project	5	1	0	0	0	640	24.3	19	20	0.65	0.38	0	12225.6	25.5
Repository	2	1	0	0	0	580	23.1	6.6	7.5	0.5	0.12	0	1198.8	2.3
Elevations	4	1	0	0	0	13	0.2	3.2	3.4	0.25	0	0	112	1.3
Imagery	13.6	1.5	0.1	0	0	2929.2	60.2	6	7.4	0.9	0.74	4	3005.2	55.2
Locations	3.2	1	0.2	0	0	49	61.4	3.6	4.5	1.0	1.0	0	384.2	3.8
Route	2	1	0	0	0	31.2	60.8	9.2	9.9	0.34	0.06	3	7308.6	63.2
TimeZone	2.2	1	0	0	0	44	1	3	3.5	1.0	0.87	0	1820.8	28.6
Average	3.7	1.05	0.03	0	0.09	554.7	30.8	8.3	9.3	0.56	0.34	0.73	4247.4	19.9

(e.g., some resources should be initially created), or only a proof-of-concept tool is provided (e.g., the service authentication, which is required for testing both *GitLab* and *Bing Maps*, is not supported).

Specifically, we configured RESTler with its default settings, and used its *Fuzz* mode to perform testing. In this mode, RESTler will iteratively extend operation sequences constructed so far by a breadth-first-search strategy. While for RESTCT, the coverage strengths $t_s = 2$ (for operation sequence), $t_e = 3$ (for essential input-parameters), and $t_a = 2$ (for all input-parameters) were used. We set t_s and t_a to two because this is the most popular choice in CT applications [24, 36]; we set t_e to three because a higher coverage strength could help in testing more interactions of essential input-parameters. We note that no initialisation process was performed (e.g., creating resources that are required for executing some operations) before the execution of both RESTler and RESTCT, because these two approaches are able to resolve the dependencies between operations. We allocated one hour time budget for each of these two approaches to test each subject API, and their executions were repeated five times to account for the randomness involved in their implementations.

To analyse the testing results, we followed the common practice of RESTful APIs testing that uses the HTTP status code received in responses as test oracle [4, 9, 42]. Specifically, an HTTP request is *successful* if a code of 200 range is received, and is *invalid* if the code is in 400 range (accordingly, an HTTP request is *valid* if the code received is not 4xx). The code of 500 range represents an *Internal Server Error* exposed (i.e., the server crash is not handled gracefully), which is probably due to a defect in the system. A potential bug is thus considered to be *detected*, if such a 5xx code is observed.

With respect to the impact of coverage strengths used in RESTCT (i.e., the second research question), we performed a *base choice* experiment to investigate the relative performance of RESTCT under different coverage strength configurations. Specifically, we used $(t_s, t_e, t_a) = (2, 3, 2)$ as a *base* configuration. Then, we changed the values of each coverage strength separately (range $[1, 2, 3]$ for t_s and t_a, and range $[2, 3, 4]$ for t_e), while letting the others remain the same, to create a series of configurations. Lastly, for each subject API, we ran multiple versions of RESTCT using the above configurations, and compared the results obtained. Here, due to the use of higher coverage strengths (accordingly, higher execution time costs), the execution time budget was set to five hours, and the execution of each configuration was repeated three times.

4.3 Results

4.3.1 ***Comparison with Existing Tool*** ($\mathbf{RQ_1}$). The first research question concerns the relative performance of RESTCT to the state-of-the-art testing tool. Table 4 gives the results obtained. Here, an operation is considered *actually tested*, if at least one HTTP request of this operation is a valid request, i.e., the status code received is either 2xx (the operation is successfully executed) or 5xx (a potential bug is detected). Accordingly, a t-way sequence of operations is *actually tested*, if every t operation involved in the sequence is actually tested (note that each single operation exactly indicates a 1-way sequence).

From Table 4, we can see that RESTCT generates, on average, 8.3 operation sequences for each subject API (i.e., the size of 2-way constrained sequence covering array), and the average length of these operation sequences is 9.3. By utilising several constrained covering arrays to execute these operations, RESTCT can actually test, on average, 56% of operations described in the specification (i.e., 1-way sequences), and this proportion is higher than 80% in four subject APIs (i.e., *Branch*, *Imagery*, *Locations*, and *TimeZone*). For 2-way sequence coverage, the operation sequences generated by RESTCT (with $t_s = 2$) ensure to cover all 2-way sequences of operations (i.e., all elements in U in Algorithm 1), where 34% of them are actually tested. Nevertheless, there remain three subjects (i.e., *Imagery*, *Locations*, and *TimeZone*), where more than 70% of 2-way sequences can be actually tested.

By contrast, we can see that the existing tool, RESTler, cannot perform well in almost all subject APIs studied. It can only exercise an average number of 3.7 operation sequences, and the average length of these operation sequences is about one (i.e., each sequence contains one operation only). The average proportion of operations

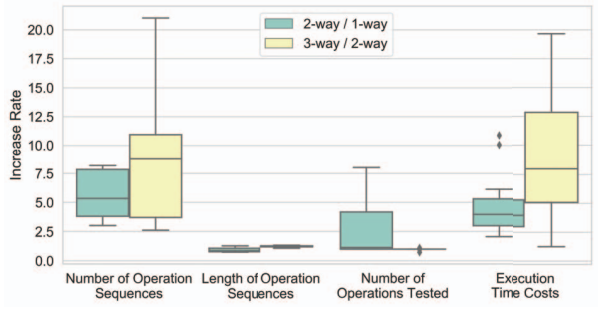

Figure 5: The impact of t_s on the performance of RESTCT.

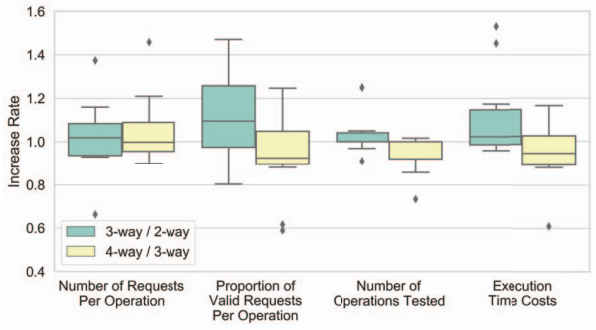

Figure 6: The impact of t_e on the performance of RESTCT.

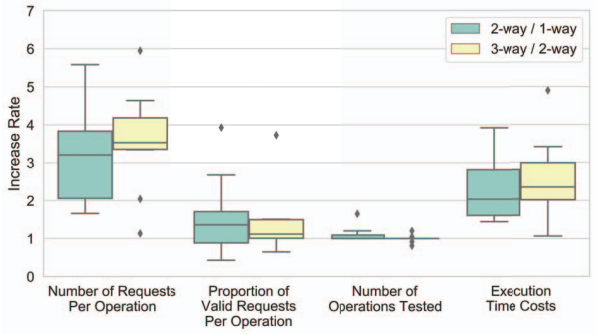

Figure 7: The impact of t_a on the performance of RESTCT.

(i.e., 1-way sequences) that are actually tested by RESTler is also low, at only 3%, and there is no 2-way sequence that can be actually tested. We note that the above results of RESTler are different from those reported in the previous study [9]. The main reason is that the previous study [9] only tries to test required input-parameters of each operation, while in this study, the RESTler tool was used to find value assignments of all input-parameters (the default setting of this tool). This indicates that RESTCT is more likely to produce valid HTTP requests to test operations that have complex (and constrained) input-parameters.

The *Bug* column of Table 4 gives the total number of unique bugs that are revealed in each subject API. In this study, we refer to a combination of an operation and a 5xx error code as a *bug*, and all bugs observed are clustered to determine the set of unique bugs. Overall, RESTCT can find a total number of eight bugs, where only one of them can be triggered by RESTler. These bugs have been manually analysed and reproduced to confirm their existence.

As far as the testing efficiency is concerned, from Table 4, we can see that RESTCT usually takes fewer time costs to generate and execute much more HTTP requests than RESTler. Specifically, RESTCT can generate and execute an average number of 4247.4 HTTP requests across all subject APIs, and this number is only 554.7 for RESTler. Meanwhile, the test execution cost of RESTCT observed (on average, 19.9 minutes) is also lower than that of RESTler (30.8 minutes). Especially, in six subject APIs, RESTCT takes no more than ten minutes to perform the test.

Answer to RQ_1: The existing testing tool, RESTler, cannot effectively test the subject APIs studied, as it tests only 3% of operations described in the specification. By contrast, RESTCT can actually test 56% of operations on average, and the test execution cost is less than ten minutes in six out of 11 subjects. Especially, RESTCT finds eight new bugs of the subject APIs, where only one of them can be triggered by RESTler.

4.3.2 ***Impact of Coverage Strengths*** (RQ_2). The second research question concerns the impact of covering strengths on the performance of RESTCT. This includes the coverage strength of operation sequences (t_s), and the coverage strengths of essential and all input-parameters in each operation (t_e and t_a). Figures 5 to 7 give the increase rates of specific metrics when changing the values of t_s, t_e, and t_a separately (each box contains 11 values obtained from the 11 subject APIs). For example, in Figure 5, the average number of operation sequences generated for *Branch* are 1.7, 6, and 25.3 when t_s takes 1, 2, and 3, respectively. The increase rates of *2-way/1-way* and *3-way/2-way* are thus 3.5 (6/1.7) and 4.2 (25.3/6).

Regarding the coverage strength of operation sequences (t_s), from Figure 5, we can see that a larger t_s tends to lead to more operation sequences (i.e., larger sequence covering arrays), and accordingly, higher test execution costs. In particular, for half of cases, the number of operation sequences and test execution costs of $t_s = 2$ are at least 5.3 and 4 times larger than those of $t_s = 1$; and these two values will increase to 8.8 and 8 when t_s increases from 2 to 3. Whereas, the increase rate of average length of operation sequences is relatively low, less than 1.4 in all cases. Generally, a higher testing cost is likely to result in a larger number of operations that can be tested. However, the increase rate of the number of operations tested of *3-way/2-way* is around 1.0, which is much lower than that of *2-way/1-way* (on average, 2.9). That is, the use of $t_s = 3$ will lead to much higher test execution costs than $t_s = 2$, but it will not test substantially more operations. So $t_s = 2$ tends to be the most cost-effective choice for the subject APIs.

Regarding the coverage strength of essential input-parameters (t_e), from Figure 6, we can see that the impact of different choices of t_e is not as great as that of t_s, as the increase rates observed are all less than 1.6. In particular, when comparing with $t_e = 2$, the use of $t_e = 3$ will slightly increase the proportion of valid requests generated per operation by a factor of 1.1 times (median value), whereas this increase rate will drop to 0.9 when t_e further increases to 4. Since a similar trend can also be observed for the other three

metrics, a larger value of t_e does not necessarily result in a higher testing performance of RestCT.

Lastly, regarding the coverage strength of all input-parameters (t_a), from Figure 7, we can see that a higher value of t_a will lead to more HTTP requests generated for each operation (i.e., larger covering arrays), and the median of increase rates observed is around 3.4 for both *2-way/1-way* and *3-way/2-way*. However, increasing t_a from 2 to 3 is not likely to result in a substantially higher proportion of valid HTTP requests per operation (and also, more operations tested). Nevertheless, the use of $t_a = 2$ can result in, on average, 1.5 times higher proportion of valid HTTP requests per operation, and test 1.1 times more operations than the use of $t_a = 1$.

Answer to RQ$_2$: the coverage strength of operation sequences (t_s) greatly influences the performance of RestCT, and $t_s = 2$ tends to be the most cost-effective choice for the subject APIs studied (the use of $t_s = 3$ will take about 8 times higher test execution costs than $t_s = 2$, but it can only test a similar number of operations). By contrast, the different choices of t_e and t_a tend to have little influence on the number of operations tested, as the increase rates observed are less than 1.7 in all cases.

5 THREATS TO VALIDITY

As far as internal threats to validity are concerned, the experimental results reported in this paper might be influenced by the concrete implementation of the RestCT approach. We have carefully inspected our codes to avoid potential mistakes, and made them publicly accessible so that others can check and extend. The experimental results can also be influenced by the Swagger specifications used. In this study, due to the unavailability of official specification files of subject APIs, we manually created the required specifications based on their online API documents. Note that RestCT relies on a pattern-based approach to extract constraints between input-parameters. We manually developed 23 patterns based on an existing catalogue of inter-parameter dependencies [31], and we followed the linguistic structures provided in the same catalogue to describe constraints when writing specifications (these two tasks were performed independently by two authors of this paper). We chose such a strategy because we want to reflect the common practice of web developers for documenting APIs (which could make these specifications more useful for future studies), rather than using our own writing styles. Consequently, RestCT (especially, the constraints extraction approach) might exhibit different performance if the specifications are written in different ways. Given the fact that the linguistic structures in the catalogue [31] indicate the most frequently used ones in practice, we believe that RestCT could work well for specifications of similar writing styles.

In addition, regarding the testing tool for comparison, we chose to use the default (recommended) configuration to run the RESTler tool in the experiment. We acknowledge that the performance of RESTler can be improved by performing several manual updates (e.g., fixing specifications, adding new dependencies of operations, providing additional fuzzing dictionaries, etc.), as suggested in its online document[10].

[10]https://github.com/microsoft/restler-fuzzer/blob/main/docs/user-guide/ImprovingCoverage.md

As far as external threats to validity are concerned, in this study, the effectiveness and efficiency of RestCT are only evaluated on 11 RESTful APIs. One reason that prevents RestCT from being evaluated more broadly is that the service under test usually requires authentication. Moreover, the publicly hosted services typically prohibit frequent requests of short periods of time, and could ban the account in an unpredictable way, which thereby introduces a substantial manual effort to apply and manage API keys (actually, we had to create several accounts to test *Bing Maps*, but such costs could be reduced if the testing is performed in a local environment). As a result, RestCT might exhibit different performance when testing different APIs (especially, APIs of different application domains). Nevertheless, both *GitLab* and *Bing Maps* are real-world, large-scale, and popular RESTful web services. We believe that they are representative subjects for the evaluation of RestCT.

6 RELATED WORK

In order to *automatically* generate test cases for RESTful APIs, both white-box [3–7, 40, 46–49] and black-box [12, 13, 16, 17, 19, 26, 28, 38, 41, 44] testing approaches have been developed. Black-box approaches attract more attentions in the current literature [32], because they require no access to the source code of the system, which are more applicable for testing complex and remote web services. Model-based testing [12, 13, 17, 19, 26, 28, 38, 44] is a representative approach in early studies of this research direction. Although test cases that satisfy certain coverage criterion can be automatically derived based on specific models (e.g., a UML state machine [38]), testers should still create such a model by hand at the beginning of testing.

Recently, the increasing popularity of the Swagger specification (more formally, OpenAPI) offers a more common and standard way for documenting RESTful APIs [39]. Accordingly, a variety of specification-based approaches [1, 2, 9, 18, 20, 22, 27, 33, 42] have been developed, which seek to automatically generate test cases based on the Swagger specification only. In this way, the operations that the service under test provides, and input-parameters that each operation requires can be easily identified. While the challenge lies in the determination of the most appropriate execution orders of different operations, as well as the value assignments of input-parameters of each operation.

With respect to the execution orders of operations, some studies [18, 27, 33, 43] only focus on the testing of each single operation (accordingly, certain resources should be created before the testing proceeds). By contrast, Viglianisi et al. [42] proposed to construct a single sequence that traverses every operation once to test nominal and exceptional scenarios of RESTful APIs. Their approach first analyses the common data field in both responses and requests to build an operation dependency graph. The next operation to be executed will then be decided based on the graph and CRUD semantics. In addition, Atlidakis et al. [9] proposed to explore all possible operation sequences to test deep service states. Their approach, RESTler, relies on the producer-consumer relationship (i.e., whether a resource received in the response of a prior operation is necessary as input of a following operation) to handle constraints between operations. Then, a search strategy, like breadth-first or random search, is applied to iteratively extend existing sequences

with all constraints-satisfying operations. Later, RESTler is further reused by several studies to develop approaches such as security testing [10] and differential testing [21] of RESTful APIs.

Regarding the determination of concrete input-parameter values, the uses of static mapping directory and random values are probably the most straightforward choices in current studies [9, 20, 42]. The example values described in the specification [42], and additional knowledge bases [1] are also used to account for the domain-specific information of the APIs. In addition, when sequences of operations are involved, there is a need to take the runtime information, i.e., resources dynamically created in prior operations, into account [9, 42]. There are also studies [20] that seek to modify the schema structures of payload to trigger as many error types as possible.

Similar to the constraints between operations, there are also inter-parameter dependencies in each single operation. The empirical study of Martin-Lopez et al. [31] has revealed that constraints between input-parameters are pervasive in real-world RESTful APIs, but unfortunately, they are barely addressed in the above mentioned approaches [9, 18, 20, 22, 27, 42]. In order to handle such constraints, Martin-Lopez et al. [29, 30] proposed a formal constraints-describing language for the OpenAPI specification. They also developed automatic approaches [33] to generate constraints-satisfying value assignments, as long as the constraints can be formally described in the specification. More recently, Mirabella et al. [35] proposed to use deep learning techniques to predict whether a value assignment is constraints-satisfying or not.

Discussions. The RestCT approach presented in this paper differs from previous approaches in three main aspects. First, unlike previous model-based approaches [18, 22], RestCT is fully automatic (that is, the test models of combinatorial testing are automatically created). Second, RestCT applies combinatorial testing to systematically cover and test the interactions of available operations, while the primary goal of previous approaches is either to test every operation once [18, 27, 33, 42, 43], or to explore all possible operation sequences [9]. At last, RestCT can automatically extract constraints between input-parameters from natural language descriptions in the Swagger specification. This differs from approaches [33] that rely on a specific formal language (should be written by testers) to realise automatic constraints handling.

7 CONCLUSION

This paper presents the RestCT approach that adopts combinatorial testing to test RESTful APIs. The primary goal of RestCT is to systematically and automatically cover and test the interactions of available operations (and also, input-parameters of each operation) described in the Swagger specification. To this end, RestCT utilises the concepts of both sequence and classical covering arrays to generate concrete HTTP requests, during which the constraints in both operations and input-parameters are automatically extracted and handled. The effectiveness and efficiency of RestCT, as well as the impact of coverage strengths applied, are evaluated on 11 real-world RESTful APIs of *GitLab* and *Bing Maps*. The experimental results demonstrate the superiority of RestCT against the state-of-the-art testing tool of RESTful APIs.

In order to aid others to replicate and extend our experiment, we provide Swagger specifications of all subject APIs and scripts to replicate experiment at https://github.com/GIST-NJU/RestCT. An archived artifact is also available at https://doi.org/10.5281/zenodo.5909761.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Program of China (No. 2018YFB1003800), National Natural Science Foundation of China (No. 61902174, No. 62072226), and Natural Science Foundation of Jiangsu Province (No. BK20190291).

REFERENCES

[1] Juan C Alonso. 2021. Automated generation of realistic test inputs for web APIs. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1666–1668. https://doi.org/10.1145/3468264.3473491

[2] Fuad Sameh Alshraiedeh and Norliza Katuk. 2020. A URI parsing technique and algorithm for anti-pattern detection in RESTful Web services. *International Journal of Web Information Systems* 17, 1 (2020), 1–17. https://doi.org/10.1108/IJWIS-08-2020-0052

[3] Andrea Arcuri. 2017. RESTful API automated test case generation. In *International Conference on Software Quality, Reliability and Security (QRS)*. 9–20. https://doi.org/10.1109/QRS.2017.11

[4] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37. https://doi.org/10.1145/3293455

[5] Andrea Arcuri. 2020. Automated blackbox and whitebox testing of RESTful APIs with EvoMaster. *IEEE Software* (2020). https://doi.org/10.1109/MS.2020.3013820

[6] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31. https://doi.org/10.1145/3391533

[7] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–34. https://doi.org/10.1145/3477271

[8] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint* (2020). https://arxiv.org/abs/2005.11498.

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: stateful REST API fuzzing. In *International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083

[10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking security properties of cloud service REST APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*. 387–397. https://doi.org/10.1109/ICST46399.2020.00046

[11] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. 2020. An OpenAPI-based testing framework to monitor non-functional properties of REST APIs. In *International Conference on Web Engineering*. 533–537. https://doi.org/10.1007/978-3-030-50578-3_39

[12] Sujit Kumar Chakrabarti and Prashant Kumar. 2009. Test-the-REST: An approach to testing RESTful web-services. In *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. 302–308. https://doi.org/10.1109/ComputationWorld.2009.116

[13] Sujit Kumar Chakrabarti and Reswin Rodriquez. 2010. Connectedness testing of RESTful web-services. In *India software engineering conference*. 143–152. https://doi.org/10.1145/1730874.1730902

[14] Yixiong Chen, Yang Yang, Zhanyao Lei, Mingyuan Xia, and Zhengwei Qi. 2021. Bootstrapping automated testing for RESTful web services. In *International Conference on Fundamental Approaches to Software Engineering*. 46–66. https://doi.org/10.1007/978-3-030-71500-7_3

[15] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful APIs. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 226–236. https://doi.org/10.1109/SCAM52516.2021.00035

[16] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Restats: A test coverage tool for RESTful APIs. In *International Conference on Software Maintenance and Evolution (ICSME)*. 594–598. https://doi.org/10.1109/ICSME52107.2021.00063

[17] Alexandre L. Correa, Thiago Silva-de-Souza, Eber Assis Schmitz, and Antonio Juarez Alencar. 2012. Defining RESTful web services test cases from UML models. In *International Conference on Software Engineering & Knowledge Engineering*. 319–323.

[18] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: a specification-based approach. In *International Enterprise Distributed Object Computing Conference*. 181–190. https://doi.org/10.1109/EDOC.2018.00031

[19] Tobias Fertig and Peter Braun. 2015. Model-driven testing of RESTful APIs. In *International Conference on World Wide Web: Companion*. 1497–1502. https://doi.org/10.1145/2740908.2743045

[20] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 725–736. https://doi.org/10.1145/3368089.3409719

[21] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *International Symposium on Software Testing and Analysis (ISSTA)*. 312–323. https://doi.org/10.1145/3395363.3397374

[22] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: property-based test generation of OpenAPI-described RESTful APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*. 131–141. https://doi.org/10.1109/ICST46399.2020.00023

[23] D Richard Kuhn, James M Higdon, James F Lawrence, Raghu N Kacker, and Yu Lei. 2012. Combinatorial methods for event sequence testing. In *International Conference on Software Testing, Verification and Validation (ICST)*. 601–609. https://doi.org/10.1109/ICST.2012.147

[24] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to combinatorial testing*. CRC press.

[25] D. Richard Kuhn and Dolores R. Wallace. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering (TSE)* 30, 6 (2004), 418–421. https://doi.org/10.1109/TSE.2004.24

[26] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In *ACM SIGPLAN workshop on Erlang*. 77–78. https://doi.org/10.1145/2505305.2505317

[27] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738–24754. https://doi.org/10.1109/ACCESS.2021.3056505

[28] Li Li and Wu Chou. 2015. Compatibility modeling and testing of REST API based on REST chart. In *International Conference on Web Information Systems and Technologies*. 194–202. https://doi.org/10.5220/0005441301940202

[29] Alberto Martin-Lopez. 2020. Automated analysis of inter-parameter dependencies in web APIs. In *International Conference on Software Engineering: Companion (ICSE-Companion)*. 140–142. https://doi.org/10.1145/3377812.3382173

[30] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* (2021). https://doi.org/10.1109/TSC.2021.3050610 to-be-published.

[31] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In *International Conference on Service-Oriented Computing (ICSOC)*. 399–414. https://doi.org/10.1007/978-3-030-33702-5_31

[32] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 15–21. https://doi.org/10.1145/3340433.3342822

[33] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: black-box constraint-based testing of RESTful web APIs. In *International Conference on Service-Oriented Computing (ICSOC)*. 459–475. https://doi.org/10.1007/978-3-030-65310-1_33

[34] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *International Conference on Software Engineering (ICSE)*. 643–654. https://doi.org/10.1145/2884781.2884793

[35] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep learning-based prediction of test input validity for RESTful APIs. In *International Workshop on Deep Learning for Testing and Testing for Deep Learning*. 9–16. https://doi.org/10.1109/DeepTest52559.2021.00008

[36] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *Comput. Surveys* 43, 2 (2011), 1–29. https://doi.org/10.1145/1883612.1883618

[37] Justyna Petke, Myra B Cohen, Mark Harman, and Shin Yoo. 2015. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering (TSE)* 41, 9 (2015), 901–924. https://doi.org/10.1109/TSE.2015.2421279

[38] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-based testing of RESTful web services using UML protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*. 1–10.

[39] Leonard Richardson, Mike Amundsen, Michael Amundsen, and Sam Ruby. 2013. *RESTful web APIs: services for a changing world*. O'Reilly Media, Inc.

[40] Omur Sahin and Bahriye Akay. 2021. A discrete dynamic artificial bee colony with hyper-scout for RESTful web service API test suite generation. *Applied Soft Computing* 104 (2021), 107246. https://doi.org/10.1016/j.asoc.2021.107246

[41] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099. https://doi.org/10.1109/TSE.2017.2764464

[42] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: automated black-box testing of RESTful APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*. 142–152. https://doi.org/10.1109/ICST46399.2020.00024

[43] Diba Vosta. 2020. *Evaluation of the t-wise approach for testing REST APIs*. Master's thesis. KTH, School of Electrical Engineering and Computer Science.

[44] Henry Vu, Tobias Fertig, and Peter Braun. 2018. Automation of integration testing of RESTful hypermedia systems: A model-driven approach. In *International Conference on Web Information Systems and Technologies*. 404–411. https://doi.org/10.5220/0006932004040411

[45] Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. 2021. Comparative analysis of constraint handling techniques for constrained combinatorial testing. *IEEE Transactions on Software Engineering (TSE)* 47, 11 (2021), 2549–2562. https://doi.org/10.1109/TSE.2019.2955687

[46] Man Zhang and Andrea Arcuri. 2021. Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–52. https://doi.org/10.1145/3464940

[47] Man Zhang and Andrea Arcuri. 2021. Enhancing resource-based test case generation for RESTful APIs with SQL handling. In *International Symposium on Search Based Software Engineering (SSBSE)*. 103–117. https://doi.org/10.1007/978-3-030-88106-1_8

[48] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Genetic and Evolutionary Computation Conference (GECCO)*. 1426–1434. https://doi.org/10.1145/3321707.3321815

[49] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 1–61. https://doi.org/10.1007/s10664-020-09937-1