MOREST: Model-based RESTful API Testing with Execution Feedback

Yi Liu
Nanyang Technological University
Singapore

Yuekang Li†
Nanyang Technological University
Singapore

Gelei Deng
Nanyang Technological University
Singapore

Yang Liu
Nanyang Technological University
Singapore

Ruiyuan Wan
Huawei Cloud Computing Technologies Co., Ltd
China

Runchao Wu
Huawei Cloud Computing Technologies Co., Ltd
China

Dandan Ji
Huawei Technologies Co., Ltd
China

Shiheng Xu
Huawei Cloud Computing Technologies Co., Ltd
China

Minli Bao
Huawei Cloud Computing Technologies Co., Ltd
China

ABSTRACT

RESTful APIs are arguably the most popular endpoints for accessing Web services. Blackbox testing is one of the emerging techniques for ensuring the reliability of RESTful APIs. The major challenge in testing RESTful APIs is the need for correct sequences of API operation calls for in-depth testing. To build meaningful operation call sequences, researchers have proposed techniques to learn and utilize the API dependencies based on OpenAPI specifications. However, these techniques either lack the overall awareness of how all the APIs are connected or the flexibility of adaptively fixing the learned knowledge.

In this paper, we propose MOREST, a model-based RESTful API testing technique that builds and maintains a dynamically updating RESTful-service Property Graph (RPG) to model the behaviors of RESTful-services and guide the call sequence generation. We empirically evaluated MOREST and the results demonstrate that MOREST can successfully request an average of 152.66%-232.45% more API operations, cover 26.16%-103.24% more lines of code, and detect 40.64%-215.94% more bugs than state-of-the-art techniques. In total, we applied MOREST to 6 real-world projects and found 44 bugs (13 of them cannot be detected by existing approaches). Specifically, 2 of the confirmed bugs are from Bitbucket, a famous code management service with more than 6 million users.

KEYWORDS

RESTful service, model-based testing

†Corresponding author.

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

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510133

ACM Reference Format:
Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. MOREST: Model-based RESTful API Testing with Execution Feedback. 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/10.1145/3510003.3510133

1 INTRODUCTION

Representational state transfer (REST) has become a de-facto standard for Web service interactions since it was introduced in 2000 [21]. Web-based APIs which follow this standard are called RESTful APIs and the web services providing the RESTful APIs are called RESTful services. Nowadays, most web service providers, such as Google [26], Twitter [27] and Amazon [25], expose RESTful APIs to grant access to other applications or services. As the RESTful APIs gain popularity, techniques for automatically testing them become important. Based on whether the knowledge of the program internals is needed or not, these testing techniques can be categorized as whitebox and blackbox techniques. Whitebox techniques are normally more effective but require source code [7]. On the contrary, blackbox techniques only rely on a well-defined interface to conduct testing [11, 47]. In comparison to their whitebox counterparts, blackbox techniques enjoy superior applicability considering that a cloud service can be implemented with different programming languages and may use third-party libraries whose source code is not available. In this paper, we concentrate on the blackbox RESTful API testing techniques.

One of the most challenging problems in testing RESTful APIs is how to infer the correct sequences of calling the API operations (aka, *call sequences*) where each API operation can be one of the four basic types — create, read, update, and delete (CRUD). This is because RESTful APIs are often organized sparsely to encapsulate different micro services and fulfilling a single task can involve a chain of API calls. Take the Petstore service [39] as an example, it is a RESTful service for selling and ordering pets. In Petstore, before calling the API to order a pet, APIs for creating the pet and updating its status as "available" must be invoked. Skipping any of

the prerequisite APIs will cause the ordering pet operation to fail, preventing the coverage of deeper logic in the code. To address the challenge of generating proper API call sequences, researchers have proposed several testing techniques [11, 20, 47] which can infer the dependencies between RESTful APIs to guide the test generation. For the purpose of API dependency inference, these techniques leverage API specifications such as OpenAPI [38], RAML [41] and API Blueprint [6]. Among these API specifications, OpenAPI (aka, Swagger) is becoming increasingly popular and gets adopted by major IT companies (e.g., Google, Microsoft, and IBM).

Two most recent state-of-the-art blackbox RESTful API testing techniques — Restler [11] and Resttestgen [47] use the OpenAPI specifications of the target RESTful services to facilitate their call sequence generation. On the one hand, both of them learn the *producer-consumer* dependencies [1] between the APIs to enforce the correct ordering of APIs in the generated call sequences. On the other hand, the difference between Restler and Resttestgen lies in how they utilize the learnt dependencies: ❶ For Restler, it uses a *bottom-up* approach, which starts with testing single APIs and then extends the API call sequences by heuristically appending API calls. Although Restler can limit the search space with dynamic feedbacks (i.e., if certain combination of APIs fails to execute, Restler avoids this pattern in the future), the search space for extending the test sequences is still very large due to the lack of overall awareness of how the APIs are connected. ❷ To gain such awareness, Resttestgen proposes a *top-down* approach to connect the APIs into an *Operation Dependency Graph* (ODG), where APIs are nodes and their dependencies are edges. With the ODG built, Resttestgen can then traverse it and aggregate the visited API nodes to generate call sequences. In this sense, Resttestgen can generate valid call sequences more efficiently. However, the quality of the tests generated by Resttestgen might be hindered since it heavily depends on the ODG, which may not reflect the API behavior correctly due to some pitfalls (e.g., poorly written OpenAPI specifications). In short, both the *bottom-up* and *top-down* approaches have strengths and weaknesses, leaving the generation of proper API call sequences an under-researched field.

In this paper, we propose Morest – a blackbox RESTful API testing technique with a dynamically updating RESTful-service Property Graph (RPG). The workflow of Morest contains two major procedures: building the RPG with the OpenAPI specifications of the target RESTful service and the model-based testing with dynamic updates of the RPG. The RPG is a novel representation of RESTful services proposed in this paper which encodes API and object schema information in the form of a mixed, edge-labeled, attributed multigraph. Compared to ODG, RPG can model not only the producer-consumer dependencies between APIs with more details but also the property equivalence relations between schemas, which allows RPG to both describe more behaviors of the RESTful services and flexibly update itself with execution feedback. By traversing the RPG, Morest can aggregate the visited APIs to build meaningful call sequences. During the testing process, Morest constantly collect the responses of the RESTful service and use the dynamic information to update the RPG adaptively. With the updated RPG, Morest can then generate test sequences with better quality for the next iteration of testing. In this sense, Morest enjoys the benefits of both the *bottom-up* and *top-down* approaches while avoiding their drawbacks by having both the overall awareness of all APIs and the flexibility of making changes.

We empirically evaluated Morest on six RESTful services running in local environment. In our experiments, Morest outperforms the state-of-the-art blackbox RESTful API testing tools, namely Restler and Resttestgen with superior average code coverage (26.16% and 103.24% respectively), average successfully requested operations (152.66% and 232.45% respectively) and average number of detected bugs (40.64% and 111.65% respectively). In total, we applied Morest to six real-world projects and found 44 bugs (13 of them cannot be detected by existing approaches). In specific, 2 of the confirmed bugs are from Bitbucket, a famous Git code management service with more than six million users [34].

Contribution. We summarize our contributions as follows:

- We propose a novel model called RESTful-service Property Graph (RPG) for describing Web services and adopt it for RESTful API testing.
- We develop a methodology for adaptively updating the RPG for enhanced performance.
- We evaluate the performance of Morest and demonstrate the superiority of Morest comparing to the state-of-the-art techniques with 1,440 CPU Hours. To the best of our knowledge, this is the first work to empirically compare blackbox RESTful API testing techniques.
- We detect 44 bugs with Morest in 6 projects. We responsibly disclose the bugs to the developers and 2 of them are confirmed until the time of writing this paper.
- We release our datasets and implementation of Morest to facilitate future research.

Currently, we have released the raw experimental data, the prototype of Morest and the evaluated benchmarks on the companion website of this paper. The link to the website is https://sites.google.com/view/restful-morest/home.

2 BACKGROUND & RUNNING EXAMPLE

2.1 Background

RESTful API. The REpresentational State Transfer (REST) architecture is is first proposed by Roy Fielding in 2000 [21]. A Web API using the REST architecture is called a RESTful API. A Web service providing RESTful APIs is called a RESTful service. One of the fundamental constraints of REST architectural style is *Uniform Interface*, which regulates the CRUD operations on the resources. In modern Web API design practice, RESTful APIs often use HTTP protocol as the transportation layer. Therefore, the CRUD operations of RESTful APIs can be mapped to the HTTP methods POST, GET, PUT and DELETE respectively.

OpenAPI Specification. OpenAPI (previously known as Swagger) defines a standard for describing RESTful APIs [38] and an API document following this standard is called an OpenAPI specification. OpenAPI specifications contain the information of the object *schemas* as well as the API *endpoints*, including but not limited to the available CRUD *operations*, input parameters as well as

[1] If a resource in the response of an API A is used as an input argument of another API B, then B depends on A.

```yaml
swagger: "2.0"
paths:
  /pet:
    post:
      operationId: "addPet"
      parameters:
      - name: "body"
        schema:
          ↪ "#/defs/Pet"
  /pet/findByStatus:
    get:
      operationId:
       ↪ "findPetsByStatus"
      parameters:
      - name: "status"
        type: "string"
      responses:
        "200":
          description:
           ↪ "success"
          schema:
           ↪ "#/defs/Pet"
  /pet/{petId}:
    get:
      operationId:
       ↪ "getPetById"
      parameters:
      - name: "petId"
        type: "integer"
      responses:
        "200":
          schema:
           ↪ "#/defs/Pet"
  /store/order:
    post:
      operationId:
       ↪ "placeOrder"
      parameters:
      - name: "body"
        schema:
         ↪ "#/defs/Order"
  /store/order/{orderId}:
    get:
      operationId:
       ↪ "getOrderById"
      parameters:
      - name: "orderId"
        type: "integer"
      responses:
        "200":
          schema:
           ↪ "#/defs/Order"
    delete:
      operationId:
       ↪ "deleteOrder"
      parameters:
      - name: "orderId"
        type: "integer"
```

(a) **(b)**

Figure 1: The OpenAPI specification of Petstore APIs*

* For clarity, we omit some details in the YAML file.

```yaml
Order:
  type: "object"
  properties:
    id:
      type: "integer"
    petId:
      type: "integer"
    status:
      type: "string"
```

(a)

```yaml
Pet:
  type: "object"
  properties:
    id:
      type: "integer"
    name:
      type: "string"
    status:
      type: "string"
```

(b)

Figure 2: The OpenAPI specification of Petstore Schemas *

* For clarity, we omit some details in the YAML file.

responses. These specifications can be stored as either YAML or JSON files.

Fig. 1 shows a fragment of the OpenAPI specification for APIs in the Petstore service [39]. In this example, five API endpoints are specified and they are marked with grey background. We can see that each API endpoint supports one or more CRUD operations, which are specified by the `operationId` property. In total, six operations are described in Fig. 2, showing their input parameters and responses. For an input parameter, it can be inside the request body (body of `addPet`) or in the URL path [2] (`petId` of `getPetById`). For a response, it contains the HTTP status code as well as the content body. In addition, some operation parameters and responses may involve objects that are described by schemas. For example, the operation `getPetById` returns responses with objects under the `Pet` schema. Fig. 2 shows the OpenAPI specifications of the schemas.

Operation Dependency Graph In RESTTESTGEN, Viglianisi et al. propose to use Operation Dependency Graph (ODG) to model data dependencies among operations which can be inferred from the OpenAPI specifications [47]. The benefit of using ODG is that aggregating operations by traversing the ODG can generate meaningful call sequences. The ODG is a graph $G = (V, E)$ where V is a set of nodes and E is a set of directed edges. For each node $v \in V$, v corresponds to a unique operation in the OpenAPI specification. The graph is said to contain a directed edge e where $e = (v_2, v_1)$ for $v_1, v_2 \in V$ and $v_1 \neq v_2$, if and only if (iff) there exists a *common field* in the response of v_1 and in the request parameter of v_2. Two fields are *common* when they have the same name if they are of atomic type (e.g., string or numeric) or when they are related to the same schema if they are of a non-atomic type. Thus, following this definition, $v_2 \longrightarrow v_1$ means v_2 depends on v_1.

2.2 Running Example

Here we introduce a running example extracted from the Petstore service [39], which can demonstrate the limitations of existing approaches and aid in the explanation of MOREST's strategies in Section 3. Fig. 1 and 2 illustrate the OpenAPI specifications of the Petstore service. The question is, given these specifications, how to generate meaningful API call sequences. For this purpose, we need to infer the dependencies among the APIs. Intuitively, the definition of API dependencies in ODG (§ 2.1) can be applied here (both RESTLER [11] and RESTTESTGEN [47] follow this definition). For the example in Fig. 1, we can get the following dependencies:

(1) `findPetsByStatus`→`getPetById`
(2) `findPetsByStatus`→`getOrderById`
(3) `addPet`→`findPetsByStatus`
(4) `addPet`→`getPetById`
(5) `getPetById`→`getOrderById`
(6) `placeOrder`→`getOrderById`

Note that some of the dependencies are infeasible: One example is in dependency 2 where `findPetsByStatus` should not depend on the `status` of the Order object returned by `getOrderById` because the status of a pet and the status of an order are not the same. Another example is in dependency 4 where `addPet` should not depend on `getPetById` although both of them use the Pet schema. While there are infeasible dependencies, they cannot be safely filtered out as yet because the current ODG model lacks the ability to describe the relation between an API and a schema in detail and dynamic execution feedbacks are needed to infer correct dependencies.

After the dependencies are acquired, we can then use them to guide the call sequence generation. To start with, we can use a *bottom-up* approach by testing single APIs first and then extend the test sequences by appending more APIs one at a time. For example, we can start with a new sequence: `<getOrderById>`. After a successful call of `getOrderById`, we can feed the `petId` of the returned Order object to `getPetById` according to dependency 5. Then the sequence becomes `<getOrderById, getPetById>`. Similarly, we can append `findPetsByStatus` to the sequence according to dependency 1 and so on. In this process, we can identify the infeasible dependencies and avoid using them in the future. For instance, we can tell dependency 4 is faulty because if we try to append `addPet` to `<getOrderById, getPetById>` according to dependency 4, the `addPet` operation will *always* fail since the service refuses to create

[2] Some GET operations may involve the usage of query parameters and they are also considered as serialized in the URL path.

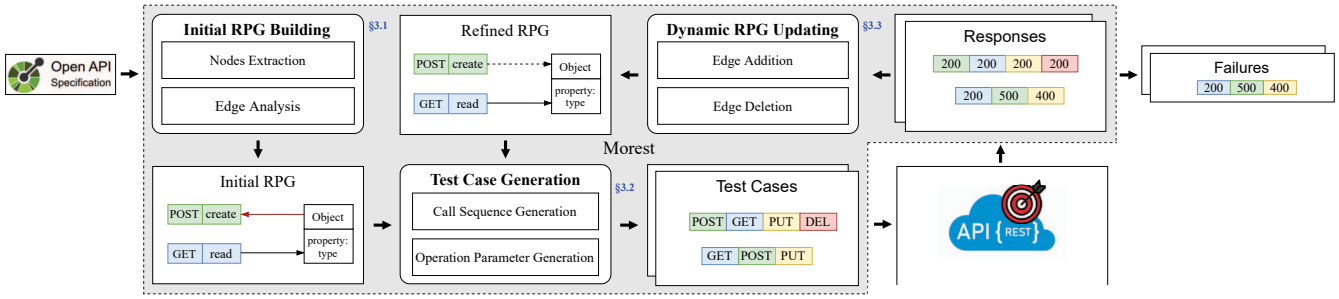


Figure 3: The workflow of Morest

an already existing pet. The limitation of the *bottom-up* approach is that it lacks the overall awareness of how the APIs are connected with each other. For example, it is hard to conduct Depth-First-Search (DFS) when extending a sequence because the potential length of the sequence is unknown and if it is very long, the risk of getting stuck in a local optimal is high.

Alternatively, we can use ODG to facilitate the generation of call sequences, which is a *top-down* approach. The call sequences can be generated by traversing the ODG and chaining the visited operations. Compared with the *bottom-up* approach, with ODG, we can quickly generate call sequences without trial and error. However, because some extracted dependencies (such as 2 and 4) might be infeasible, the quality of the generated call sequences can be affected. In other words, the *top-down* approach lacks the flexibility of dynamically fixing the call sequences.

In summary, both the *bottom-up* and *top-down* approach have their own strengths and weaknesses and they can complement each other. Therefore, a new model is needed to both provide high level guidance and perform self-updates. Following this observation, we propose Morest, which leverages RESTful-service Property Graph (RPG) to adopt the advantages of both approaches while circumventing their disadvantages.

3 METHODOLOGY

Fig. 3 shows the detailed workflow of Morest. To test a RESTful service, Morest first takes its OpenAPI specifications as input to build the RPG — a novel representation of the relations among RESTful APIs and the schemas (§ 3.1). After building the initial RPG, Morest uses it to generate call sequences and replace the API calls in the call sequences with actual requests (§ 3.2). Then the generated test cases are fed to the target RESTful service and Morest shall collect the responses. By analyzing the collected responses, Morest reports the detected failures for bug analysis. Moreover, Morest also uses the responses to refine the RPG by adding missing edges and removing infeasible edges (§ 3.3). The refined RPG is then used for generating more test sequences. This marks the end of one iteration and Morest will keep testing the target RESTful service and refining the RPG until the time budget is reached.

3.1 Initial RPG Building

In Morest, we propose the concept of RPG to encode the CRUD relations of operations, the relations of the schemas, and the data-flow among the schemas and operations. The design of RPG is based on the concept of *property graph* [42]. The definition of *property graph* is as follows:

Definition 1 (Property Graph). *A property graph is a directed, edge-labeled, attributed multigraph $G = (V, E, \lambda, \mu)$ where V is a set of nodes (or vertices), E is a set of directed edges, $\lambda : E \rightarrow \Sigma$ is an edge labeling function assigning a label from the alphabet Σ to each edge and $\mu : (V \cup E) \times K \rightarrow S$ is a function assigning key(from K)-value(from S) pairs of properties to the edges and nodes.*

Given the definition of *property graph*, RPG is defined as follows:

Definition 2 (RESTful-service Property Graph). *A RPG is a mixed,edge-labeled, attributed multigraph $G = (V, E, \lambda, \mu)$ with:*

- $V = V_{schema} \cup V_{operation}$
- $E = E_{os} \cup E_{so} \cup E_{ss} \cup E_{oo}$
- $\lambda = \lambda_{so} \cup \lambda_{ss}$
- $\mu = \mu_{schema} \cup \mu_{operation}$

where V_{schema} is a set of schema nodes, $V_{operation}$ is a set of operation nodes, E_{os} is a set of directed edges pointing from a node in $V_{operation}$ to a node in V_{schema}, E_{so} is a set of directed edges each of pointing from a node in V_{schema} to a node in $V_{operation}$, E_{ss} is a set of undirected edges connecting two nodes in V_{schema}, E_{oo} is a set of undirected edges connecting two nodes in $V_{operation}$, λ_{so} is a set of labeling functions to label edges in E_{so}, λ_{ss} is a set of labeling functions to label edges in E_{ss}, μ_{schema} is a set of functions to assign properties to nodes in V_{schema}, and $\mu_{operation}$ is a set of functions to assign properties to nodes in $V_{operation}$

Note that RPG is not exactly a *property graph* because it is a mixed graph while the latter is a directed graph and this is the only difference. In the following of this paper, for simplicity, we use o_x to represent an element of $V_{operation}$, s_x to represent an element of V_{schema}, e_{os} to represent an element of E_{os} and so on.

Design & Rationale. Here we explain the details and rationale for the design of RPG. ❶ The usage of V_{schema} and $V_{operation}$ are straight forward, we need them to represent the schemas and operations. ❷ As for edges, the edges in E_{os} and E_{so} are directed to represent whether an operation produces or consumes an object of a schema. For example, the edge $e_{o1s1} = (o_1, s_1)$ means operation o_1 returns an object of schema s_1 as its response. Correspondingly, the edge $e_{s1o1} = (s_1, o_1)$ means operation o_1 requires an object or at least one property of the object of schema s_1 in its parameter(s). The edges in E_{ss} represent the equivalence relation between two properties from two different schemas. In the running example, the property `id` in the Pet schema is referring to the same thing

as the property `petId` in the Order schema. We denote the edge connecting these two schemas as $e_{s_{pet}s_{order}} = \{s_{pet}, s_{order}\}$. The edges in E_{oo} are used to connect two operations if they are under the same API endpoint. In the running example, the operation `getOrderById` and the operation `deleteOrder` are connected by this type of edge. ❸ Only the edges in E_{so} and E_{ss} are labeled by functions in λ. The labels for E_{so} are vectors of property names showing which exact properties of a schema are used as parameters for an operation. Note that the empty vectors are ignored in Fig. 4. The labels for E_{ss} are vectors of tuples, where each tuple is pair of properties with equivalence relations from two different schemas. The edges in E_{oo} requires no labelling since they can only indicate two operations belong to the same API endpoint. The labelling for E_{os} is ignored for two reasons. First, labelling edges is an error-prone process, especially when the OpenAPI specification is poorly written, the more edges we label, the more errors we may introduce. Second, comparing with the edges in E_{so}, the edges in E_{os} are less important for successfully requesting API operations because the former ones directly affect the input parameters of the operations. ❹ Naturally, the functions to assign properties to the nodes (μ) correspond to the process of parsing the OpenAPI specification and extracting data from it.

With the OpenAPI specifications, MOREST can build the initial RPG consisting of nodes and edges together with their properties or labels according to Def. 2. This initial RPG, like shown in Fig. 4a, may contain false edges or miss some edges and MOREST will refine it with dynamically collected feedback later on.

RPG vs ODG. Here we compare the RPG model with the ODG model. The first difference is that ODG has only the operation nodes and their producer-consumer dependencies as edges while RPG has more types of nodes and edges. The second difference is that RPG allows nodes and edges to have properties and labels. Both of the differences indicate that RPG can describe more details of the RESTful services. The additional details captured by RPG can help to generate longer call sequences and provide the information needed for dynamic self-updating. For example, for the RPG shown in Fig. 4b, the connection between the `Pet` and `Order` schemas indicates that `Order.petId` refers to the same thing as `Pet.id`. With this information, we can infer that the `Order.petId` can be used to query `getPetById` and therefore generate the call sequence (`placeOrder`, `getPetById`, `findPetsByStatus`), which cannot be generated only with the dependencies between operations in an ODG.

3.2 Test Case Generation

Here we introduce how MOREST uses RPG for test case generation. In general, this requires two steps: first, MOREST traverses the RPG and aggregates visited operations to form call sequences (*Call Sequence Generation*); second, MOREST generates concrete inputs for the APIs in the sequences to build test cases (*Operation Parameter Generation*).

Call Sequence Generation. Algo. 1 shows how MOREST generates call sequences where $\mathcal{S}$ is a set of call sequences and $\mathbb{S} = (o, o_1, ..., o_n)$ is a single call sequence made up of a vector of operations. The function *call_sequence_generation* is the start of the whole process.

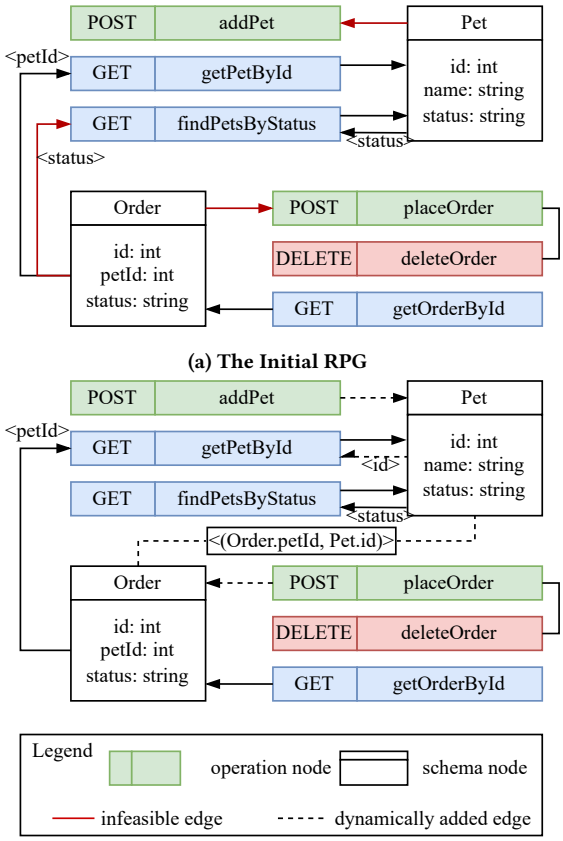


Figure 4: The RESTful-service Property Graphs (RPGs) for the Petstore running example.

In general, the idea of this algorithm is to visit every schema in the RPG and collect the operations related to the schema to build up call sequences. While accessing a schema, we can recursively traverse other schemas which are connected to the current one, collect the related operations to build new call sequences and concatenate the new call sequences and the existing call sequences to build longer sequences. After we have generated the call sequences, we further apply a *crud_filter* to filter out the call sequences violating the CRUD rules. A call sequence is said to violate the CRUD rule if it encounters any of the two situations: for the same schema, a delete operation appears on the sequence before another operation; for the same schema, any of the read, update or delete operation appears on the sequence before the first create operation.

In particular, when we visit a schema s (line 8), we first need to identify two sets of operations: the operations which produce schema s in their responses (O_{out}) and the operations which consume schema s or its properties as their input parameters (O_{in}). With O_{out} and O_{in}, we can use their Cartesian product to build a set of call sequences ($\mathcal{S}_{new}$) with the length of two (line 16). Then, if we already have a set of call sequences ($\mathcal{S}$), we can try to concatenate them and the new call sequences to build longer sequences (line 17). Two sequences can be concatenated if the last operation of one of them is the same as the first operation of the other one

Algorithm 1: Call Sequence Generation

```
1  def call_sequence_generation(V, E, λ, μ):
2      S ← ∅;
3      for s ∈ V_schema do
4          S ← S ∪ visit(s, ∅, V_schema, ∅);
5      S ← crud_filter(S);
6      return S;
7
8  def visit(s, V_visited_schema, V_schema, S):
9      O_out ← {o ∈ V_operation | (o, s) ∈ E_os};
10     O_in ← {o ∈ V_operation | (s, o) ∈ E_so};
11     if O_in = ∅ ∨ O_out = ∅ then
12         return S;
13     if {e_ss ∈ E_ss | e_ss = {s, s'} ∧ s' ≠ s, s' ∈ V_schema} = ∅ then
14         return S;
15     V_visited_schema ← V_visited_schema ∪ {s};
16     S_new = O_out × O_in;
17     S ← concat(S, S_new);
18     for s' ∈ {s' ∈ V_schema | (∃ e_ss' ∈ E_ss)[e_ss' = {s, s'} ∧ s ≠
         s'] ∧ s' ∉ V_visited_schema} do
19         S ← S ∪ visit(s', V_visited_schema, V_schema, S);
20     return S;
21
22 def concat(S, S_new):
23     for 𝕊 = (o_1, ..., o_n) ∈ S do
24         for 𝕊' = (o_out, o_in) ∈ S_new do
25             if o_n = o_out then
26                 S ← S ∪ {(o_1, ..., o_n, o_in)};
27             if o_in = o_1 then
28                 S ← S ∪ {(o_out, o_1, ..., o_n)};
29     return S;
30
31 def crud_filter(S):
32     for 𝕊 ∈ S do
33         if not crud_valid(𝕊) then
34             S ← S − {𝕊};
35     return S;
```

(line 22). After that, if the current schema node is connected to another schema node which has not been visited in the current iteration (line 19), we will visit that schema and repeat the same process until we run out of schema nodes. Note that each schema node is visited only once per iteration to avoid infinite loops.

In the running example, with the RPG in Fig. 4b, assume we are now visiting the `Pet` node and the `Order` node is not visited yet. For the `Pet` node, its O_{out} is {`addPet`, `getPetById`, `findPetsByStatus`} and its O_{in} is { `getPetById`, `findPetsByStatus`}. The Cartesian product of O_{out} and O_{in} can generate six new call sequences including (`getPetById`, `findPetsByStatus`) etc. (line 16). Since we do not have any existing call sequences so far, we do not need to perform call sequence concatenation (line 17). Then, because the `Order` node is connected to the `Pet` node and it has not been visited, we will visit it and generate new call sequences similarly (line 19). One of the newly generated call sequences for the `Order` node is (`placeOrder`, `getPetById`) (line 16). Now, since we already have generated some sequences when we visit the `Pet` node, we can try to concatenate them with the newly generated call sequences for the `Order` node (line 17). In this example, we can generate the call sequence (`placeOrder`, `getPetById`, `findPetsByStatus`). This demonstrates how MOREST generates call sequences.

Note that, in some cases, there are standalone operations that are not connected to any schemas. For clarity, we omit handling these cases in the algorithm. But in our implementation, the standalone operations are included as single-element call sequences.

Operation Parameter Generation. The generated call sequences cannot be used directly for testing since they are just sequences of operations without concrete parameter values. For a given operation, if its input parameters are *not* from the responses of other operations on the same test sequences, they are decided as follows: if the operation has been called correctly before, MOREST assigns a high chance of using the parameters of the last successful run, else if the OpenAPI specifications have specified the valid ranges of values for the operation, MOREST has a high chance of selecting values from the valid ranges, otherwise, MOREST will just use random values. Given an object schema, MOREST recursively traverses the schema to generate values for the object attributes according to their parameter types. If the parameter is of a basic type (e.g., string and int), we will concretize the value accordingly. If the parameter is an object or a list, we will jump in and recursively deduce the basic parameter type.

An important detail worth discussing is that the parameters for a sequence of operations is not generated all at once. Instead, these parameters are generated on the fly. Because if the input parameter for an operation is based on the response of a previous operation, we will need to wait for the previous operation to finish execution to get the needed parameter values.

3.3 Dynamic RPG Updating

The initial RPG generated with the OpenAPI specifications may contain errors. For example, the red lines in Fig. 4a are infeasible edges, and the dashed lines in Fig. 4b are the missing edges for the initial RPG. This is often due to ambiguities in the specifications. For example, although a human can quickly recognize that the `petId` property of `Order` refers to the same thing as the `id` property of `Pet`, it is hard to use rules and heuristics to reconstruct this relation. To address this problem, MOREST dynamically fixes the RPG with execution feedback.

Edge Addition In MOREST, new edges are added to a RPG in three scenarios: ❶ The most straightforward case is where the response value of an API aligns with a certain schema but it is not documented in the specification. For example, in Fig. 1, the response of the `addPet` operation is not documented. However, after `addPet` is corrected during the testing, it will respond with the newly created `Pet` object. MOREST will notice that the properties of the object returned by `addPet` matches the properties in the `Pet` schema. Then MOREST can add the edge (`addPet`,`Pet`) into E_{os}. ❷ The edges of E_{ss} are added after the collection of execution feedback. For two schema nodes s_1 and s_2 if both of them are related to an operation node o, MOREST makes the assumption that s_1 and s_2 share at least one property in common but MOREST cannot decide which property is shared in the RPG building stage. During the execution of the test cases, MOREST may encounter some sequences involving both the objects of s_1 and the objects of s_2. With the responses of these sequences, MOREST can compare the concrete values of properties between the two types of objects. As far as the value of a property from an object of s_1 can match the value of multiple properties from

Table 1: Open source RESTful services

Subjects	LoC	Language	Operation	Endpoint	Source
Petstore [39]	989	Java	20	18	OpenAPI
SpreeCommerce [7]	42,385	Ruby	35	31	EMB
Bitbucket [10]	46,321	Java	32	23	Bitbucket
LanguageTool [7]	4,760	Java	5	5	EMB
FeatureService [7]	1,525	Java	18	11	EMB
Magento [7]	315,120	PHP	47	39	EMB

an object of s_2, the relation between s_1 and s_2 remains undecided. Until we find a property whose value only matches with one property from the other set of objects, we can tell that these two schemas share this common property. For example, both `Order` and `Pet` connect with `getPetById` (Fig. 4a). MOREST can generate multiple test cases with the sequence of (`getOrderById`, `getPetById`). Suppose these test cases yield the following two pairs of orders and pets: (`{id: 1, petId: 1, status: "succ"}`, `{id: 1, name: "cat", status: "sold"}`) and (`{id: 2, petId: 4, status: "succ"}`, `{id: 4, name: "dog", status: "sold"}`). After the generation of the first pair of order and pet, MOREST cannot infer the relation since the `id` of the pet equals to both `id` and `petId` of the order. MOREST can only infer that `Pet.id` is equal to either `Order.id` or `Order.petId`. Nevertheless, after the second pair is generated, MOREST can draw the conclusion of `Pet.id` equals to Order.petId. ❸ The last scenario is where the edges belonging to E_{so} and E_{os} are added based on the inferences used for building E_{ss}. For example, after MOREST learns the fact that `Pet.id` equals to Order.petId, it propagates this information to other nodes, say `getPetById`. Then, MOREST can link the `petId` parameter needed by `getPetById` with the property `Pet.id` despite that they have different names. As a result, MOREST can add the edge (`Pet`, `getPetById`) to E_{so} as shown in Fig. 4b.

Edge Deletion For an operation, if its inputs are from other operations on the same call sequence (i.e. not generated randomly) and it fails to execute correctly after Θ tries, then the edge pointing from the respective schema to this operation is temporarily recognized as infeasible. Empirically, we find that increasing the value of Θ can help to reduce the number of falsely deleted edges but the overall performance is not affected too much since we are wasting more time on the truly infeasible edges and even if a benign edge is deleted, MOREST always has the chance of adding it back.

4 IMPLEMENTATION & EVALUATION

We have implemented MOREST based on Python 3.9.0 with 11,659 lines of code and conducted experiments to evaluate the performance of MOREST. We aim to answer the following research questions with the evaluation:

RQ1 (Coverage) How is the code coverage and operation exploration capability of MOREST?

RQ2 (Bug Detection) How is the bug detection capability of MOREST?

RQ3 (Ablation Study) How do RPG guidance and dynamic RPG updating affect the performance of MOREST separately?

4.1 Experiment Setup

Evaluation Datasets. We built the evaluation benchmark with six open source RESTful services shown in Table 1. In this benchmark, four services were selected from the EVOMASTER Benchmark (EMB) [7] [3]. In addition, Petstore (the official demo of OpenAPI specification) and Bitbucket (a popular online version control system with complicated business scenarios) were also included.

From Table 1, we can see that the target RESTful services are diverse in sizes, features and are implemented with different programming languages. Bitbucket is not strictly open source since it is a commercial product. However, we have access to its released .jar file to collect line coverage and conduct detailed bug analysis.

Evaluation Baselines. We use three state-of-the-art blackbox RESTful service testing techniques as baselines to study the performance of MOREST.

(1) **EVOMASTER-BB**: EVOMASTER [7] is a search-based whitebox technique. In our evaluation, we disabled its code instrumentation (only works for Java programs) and database monitoring modules to turn it into blackbox mode (named as EVOMASTER-BB). It uses various heuristics to generate call sequences according to the OpenAPI specifications. Thus, EVOMASTER-BB represents the heuristic-based approach.
(2) **RESTTESTGEN**: RESTTESTGEN [47] is a blackbox technique which generates Operation Dependency Graphs (ODGs) from OpenAPI specifications to guide the call sequence generation. RESTTESTGEN represents the top-down approach.
(3) **RESTLER**: RESTLER is a blackbox technique which build call sequences by appending new API operations according to execution feedback. RESTLER represents the bottom-up approach.

Evaluation Criteria. We use three criteria to evaluate MOREST and the baselines.

(1) **Code Coverage**: Code coverage can reflect the exploration capability of the techniques. In the experiments, we use line coverage since it is the finest granularity we can get with our current tools (PHPCoverage [5] for PHP, CoverBand [1] for Ruby and JaCoCo [2] for Java).
(2) **Successfully Requested Operations**: In HTTP, responses with status code 2xx are successful responses [37]. We use the number of successfully requested operations (SROs) as a criterion because it can reflect whether a technique can generate valid requests to test deeper code logic of the RESTful service and valid requests are partially the results of correct call sequences.
(3) **Bugs**: The goal of testing is to expose bugs, so the number of detected bugs is a necessary criterion. In the context of RESTful service, a *failure* is considered to happen when an operation returns 5xx status code and a *bug* can be related to many failures. We manually classified the failures into unique bugs in the experiments according to response bodies, server logs, etc.

Evaluation Settings. For the open source RESTful services, we hosted them on a local machine and ran each technique with three time budgets — 1, 4 and 8 hour(s). The purpose is to evaluate how the performance of these techniques change over time. To the best

[3] We only use the open source services available in EMB up to the time of writing this paper.

of our knowledge, the time budget of 8 hours is the longest time budget used in RESTful service testing research. After each round, we tear down and restore the environment (e.g., docker containers, self-hosted virtual machines) to guarantee the consistence of all RESTful services. In addition, we repeated all experiments for 5 times to mitigate randomness and applied Mann-Whitney U test and $\hat{A}_{12}$ [3, 46] calculation for statistical tests. Thus, we conducted a total number of 1,440, i.e., 6 projects * 6 settings * 8 (hour time budget for one round) * 5 repetitions, CPU hours of experiments.

4.2 Coverage (RQ1)

To comprehensively compare different approaches, EvoMaster-bb, RestTestGen, and Restler are directly adopted from the prior work [7, 11, 47]. Besides, to avoid the side effect of service under test, we wrap six evaluation subjects into docker image [19] and restore the corresponding image after each round. Table 2 presents the statistical results of five runs. As suggested by the prior work [28], the Mann-Whitney U test (with the confidence threshold $\alpha = 0.05$) is adopted here. Overall, we summarize our findings as follows.

Fig. 5 and Table 2 depict that [4] Morest achieves competitive performance regarding both code coverage and successfully requested operations, significantly outperforming search-based, and model-based approaches in 5/6 RESTful services (bold numbers in Table 2). Fig. 6a demonstrates unique successfully requested operations by each tool. We can figure out that Morest covers the most operations by generating valid call sequences following producer-consumer dependencies. Therefore, existing experimental result demonstrates the exploration effectiveness of Morest regarding given coverage criteria (code coverage increased by 26.16% - 103.24%, successfully requested operations improved by 152.66% - 232.45%). On the other hand, it presents the robustness of Morest gained by RPG guidance and RPG updating fashion (experiment results are statistically significant).

Further, we conduct an in-depth analysis of the generated test sequences by all approaches to figure out why all approaches achieve the same number of successfully requested operations. Taking the LanguageTool as an example, we find that all approaches could not cover three operations getWords, addWord and deleteWord. Two parameters, *username* and *APIKey*), in those operations are labeled as required parameters (i.e., *username* and *APIKey* must be filled in each request when testing those operations). However, we observe that the implementation of LanguageTool is different from descriptions in the OpenAPI specification. Specifically, in the implementation of LanguageTool, *username* and *APIKey* could not be filled simultaneously; otherwise, the server behaves unexpectedly and throws 5xx responses. As a consequence, we could imply that the gap between RESTful API specifications and actual implementation of RESTful service could affect the effectiveness of RESTful API testing.

4.3 Bug Detection (RQ2)

Subsequently, we analyze the bug (defined in Section 4.1) discovering the capability of each approach. Table 3 presents the statistical results of the averaged bugs detection with running tools for 8 hours each round. Since one bug can be detected several times during testing, we only record the number of unique bugs for all approaches. Overall, we summarize our findings as follows.

First, in those baselines, EvoMaster-bb, Restler, and RestTestGen outperform each other in different subjects. We manually analyze the test sequences generated by baseline and find that ❶ Restler could sufficiently explore all call sequences theoretically, however, we have observed that massive invalid call sequences make Restler inefficient;❷ EvoMaster-bb heuristically generates redundant call sequences to make it inefficient;❸ RestTestGen is trapped by infeasible call sequences generated from the operation dependency graph, which limit its performance.

Besides, as shown in Table 3, Morest detects the most bugs (bold numbers in Table 3), and statistically outperforms other baselines on testing RESTful API services. Specifically, Fig. 6b indicates that Morest could detect unknown 13 bugs by existing approaches. It not only shows the comparative performance in bug finding but also the Morest's robustness.

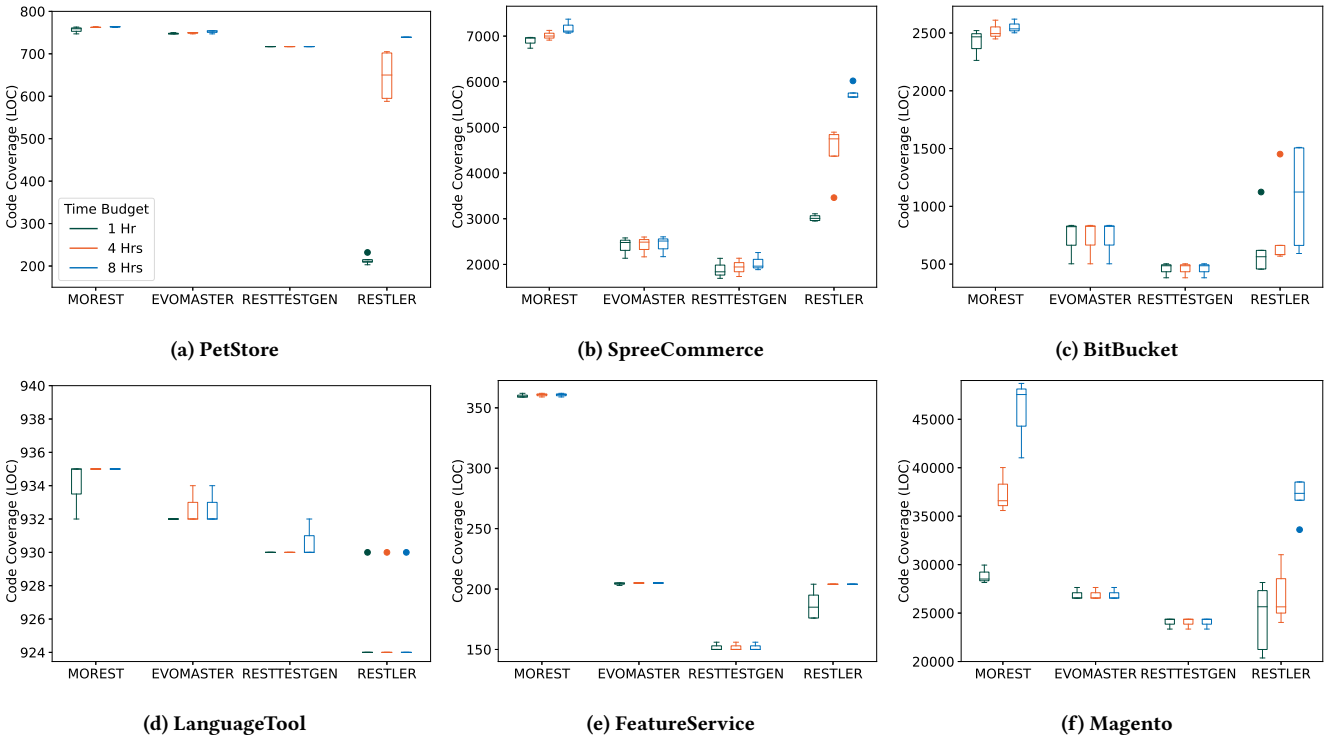
Similarly to Section 4.2, we manually perform a comprehensive study in LanguageTool to figure out why all approaches find identical bugs. We find that due to the gap between RESTful API specification and the actual implementation of RESTful service (see detail description in Section 4.2), all approaches get 5xx responses by following the parameter's specifications (e.g., *word_size* is defined by the string type while implemented by the integer type) in the LanguageTool's specification. This indicates that RESTful API testing could help developers to identify the incompatibility between the RESTful API specifications and actual implementations.

Case Study. Two bugs, discovered by Morest, have been confirmed and fixed by the developers of BitBucket. We conduct a case study to find out how Morest trigers those bugs. Bitbucket is a Git-based source code repository hosting service owned by Atlassian. Bitbucket Server [9] is the self-host service provided by Atlassian that allows users to do unlimited API queries to the target server. During the testing of Bitbucket, we identified the following call sequence that triggers Internal Server Error with 500 status code. In particular, (1) create a project at /rest/.../projects endpoint through POST operation. The project information can be further retrieved by GET request on the same endpoint. (2) create a repository in the project at /rest/.../{projectKey}/repos endpoint through POST operation, where the {projectKey} parameter is defined in the first step as a parameter. (3) A further GET query on /rest/.../repos/{repositorySlug}/ commits with parameters {"path": "test_string"} triggers internal server error. When debugging mode is enabled, the bug message is printed out: "com.atlassian.bitbucket.scm.CommandFailedException". In summary, both a project and a repository should be created firstly to trigger this bug. With the RPG guidance and dynamic RPG updating, Morest could adaptively generate such call sequences.

4.4 Ablation Study (RQ3)

To investigate how RPG and dynamic RPG updating contributes to boosting the testing effectiveness of Morest through high-level guidance and adaptive updating, we perform a ablation study on each component. To study the contributions separately, we implement two variants of Morest as following (1) Morest-NO-RPG by

[4] Due to page limitation, we leave experiment result (code coverage and successfully requested operations), for 1 hour and 4 hours, in our website [4].



(a) PetStore (b) SpreeCommerce (c) BitBucket

(d) LanguageTool (e) FeatureService (f) Magento

Figure 5: Comparing with baselines in terms of code coverage (μLOC)

Table 2: Performance of MOREST over EVOMASTER-BB, RESTTESTGEN, and RESTLER in terms of both the code coverage (μLoC) and successfully requested operations (SRO). We run this experiment 5 times (8 hours each time) and highlight statistically significant results in bold. (We calculate the average increased number by $\frac{(\#\ of\ \textsc{Morest}) - (\#\ of\ baseline)}{\#\ of\ baseline}$.)

Subjects	Average Code Coverage (LoC)							Average # of Successfully Requested Operations (SRO)						
	MOREST	EVOMASTER-BB		RESTTESTGEN		RESTLER		MOREST	EVOMASTER-BB		RESTTESTGEN		RESTLER	
	μLOC	μLOC	$\hat{A}_{12}$	μLOC	$\hat{A}_{12}$	μLOC	$\hat{A}_{12}$	μSRO	μSRO	$\hat{A}_{12}$	μSRO	$\hat{A}_{12}$	μSRO	$\hat{A}_{12}$
Petstore	**763.20**	751.80	1.00	717.00	1.00	739.00	1.00	**20.00**	14.00	1.00	13.00	1.00	18.00	1.00
SpreeCommerce	**7182.00**	2428.80	1.00	2036.00	1.00	5753.80	1.00	**18.40**	1.00	1.00	1.00	1.00	1.00	1.00
Bitbucket	**2552.60**	721.80	1.00	457.00	1.00	1078.80	1.00	**15.00**	2.00	1.00	2.00	1.00	1.00	1.00
LanguageTool	**935.00**	932.80	1.00	930.8	1.00	925.20	1.00	1.00	1.00	0.50	1.00	0.50	1.00	0.50
FeatureService	**360.00**	205.00	1.00	152.00	1.00	204.00	1.00	**18.00**	6.00	1.00	4.40	1.00	9.00	1.00
Magento	**45759.00**	26912.00	1.00	24024.40	1.00	36933.00	1.00	**18.60**	8.00	1.00	6.00	1.00	6.00	1.00
Average Increased (%)	0.00	80.12	-	103.24	-	26.16	-	0.00	184.42	-	232.45	-	152.66	-

Table 3: Comparing with baseline in terms of the number of detected bugs. (Values in bold indicate statistically significant differences between MOREST, EVOMASTER-BB, RESTTESTGEN, and RESTLER.)

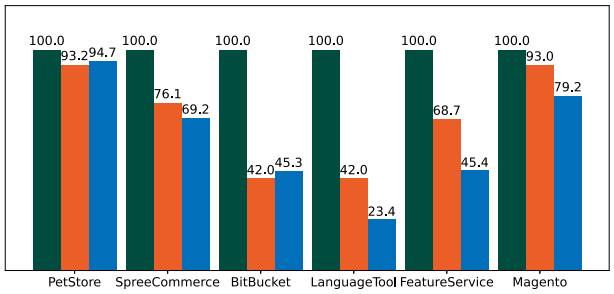
Subjects	Average Detected Bugs (#)						
	MOREST	EVOMASTER-BB		RESTTESTGEN		RESTLER	
	$\mu\#$	$\mu\#$	$\hat{A}_{12}$	$\mu\#$	$\hat{A}_{12}$	$\mu\#$	$\hat{A}_{12}$
Petstore	**9.00**	3.80	1.00	0.00	1.00	8.00	1.00
SpreeCommerce	**3.00**	0.00	1.00	0.00	1.00	0.00	1.00
Bitbucket	**2.00**	0.00	1.00	0.00	1.00	0.00	1.00
LanguageTool	3.00	3.00	0.50	3.00	0.50	3.00	0.50
FeatureService	**12.00**	6.00	1.00	11.60	0.67	10.00	1.00
Magento	**14.60**	1.00	1.00	6.00	1.00	10.00	1.00
Average Increased (%)	0.00	215.94	-	111.65	-	40.64	-

disabling RPG guidance, (2) MOREST-RPG-ONLY by removing the dynamic RPG updating part. The results are averaged using five runs (with time budget one hour) to avoid the statistics bias. Fig. 7a presents normalized code coverage of baselines on our datasets (To better visualization and understanding, we normalize the code coverage by $normalized\_coverage = \frac{LOC}{LOC(\textsc{Morest})}$). Fig. 7b shows the number of detected bugs.

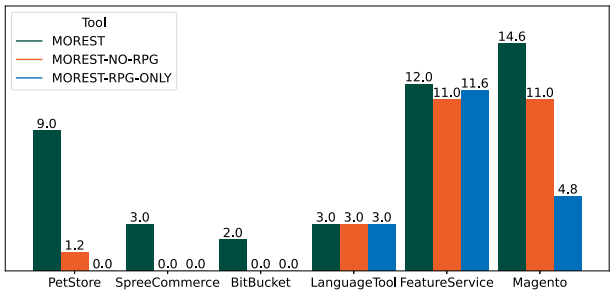
In general, as shown in Fig. 7, we observe that MOREST outperforms MOREST-RPG-ONLY and MOREST-NO-RPG on both code coverage and bug detection. In specific, we take Petstore as an example. To cover more operations related to the 'Pet' schema, an instance of pet should be firstly created by *addPet*. However, MOREST-NO-RPG fails to generate such call sequences, which restricts further exploration during testing. Additionally, properties named *status*, within both 'Order' (refers to the status of order) and 'Pet' (refers to the status of pet) schemas, leading to invalid producer-consumer

(a) Successfully Requested Operations (SRO) (b) Bug detection

Figure 6: Venn diagrams showing both successfully requested operations (SRO) and bug detection that MOREST, EVOMASTER-BB, RESTTESTGEN and RESTLER individually and together.



(a) Normalized average code coverage



(b) Average unique bug detection

Figure 7: The performance of MOREST, MOREST-NO-RPG and MOREST-RPG-ONLY on both normalized average code coverage (μLOC) and bug detection.

dependencies in RPG. The performance of MOREST-RPG-ONLY is limited by such cases. Thus, we can infer that the RPG guidance and dynamic RPG updating can both boost the effectiveness of the RESTful API testing process.

Besides, as we can see from Fig. 7, MOREST achieves better performance other approaches regarding both code coverage and bug detection. In particular, MOREST generates call sequences with the guidance of RPG and dynamically updates RPG to adaptively improve call sequences to obtain higher code coverage, which increasing the possibility to detect bugs.

4.5 Threats To Validity

The first internal threat comes from the choices of configurable options in the design. Currently, we empirically set the values for these options and the actual values of the options used in the experiments can be found on our website [4]. Although the choice of configurable options can affect the performance of MOREST, the experiment results can demonstrate that at least with the current set of option values, MOREST can outperform state-of-the-art techniques. Therefore, we leave the fine-tuning of these options as future work. Another internal threat is that the current RPG model can only reflect the APIs documented in the OpenAPI specifications. For undocumented APIs, a possible solution is to infer their related schemes in an online manner by analyzing the feedback of the testing process. Another possible solution is to use client-side analysis techniques [31, 32, 49] to generate traffic between clients and servers and analyze the traffic to infer the correct usage of the undocumented APIs. We leave the support of undocumented APIs as future work.

The external threats mainly come from the experiment settings. The testing techniques evaluated in the experiments are random by nature. To mitigate the random factors, we repeated each experiment for five times and conducted statistical tests. Therefore, the experiment results are statistically sound. To address the generality concern, we chose a diverse dataset consisting of six RESTful services with various sizes and features. Moreover, we chose open source RESTful services so we can perform in-depth experiment result analyses via scrutinizing the code coverage and classifying the detected failures into unique bugs. Last but not the least, to improve the fairness for technique comparison, we gave each technique a generous time budget of 8 hours while in [11] the longest time budget is 5-hour and in [47] the time budget is 0.5-hour.

5 RELATED WORK

Instead of discussing all related works, we focus on the RESTful service testing techniques, SOAP service testing techniques and model-based testing techniques.

Blackbox RESTful service testing techniques. Several blackbox techniques were proposed to generate meaningful call sequences. RESTTESTGEN [47] builds Operation Dependency Graphs (ODGs) with the OpenAPI specifications to model RESTful services and crafts call sequences via graph traversal. The quality of the generated call sequences is limited by the quality of the OpenAPI specifications which directly affect ODG building. Meanwhile, RESTLER [11] builds call sequences with a bottom-up approach which starts with single operation call sequences and gradually extend the call sequences by appending more operations after trial and error. Comparing to these techniques, MOREST can enjoy both high-level guidance and the flexibility of dynamic adjustments to achieve better performance.

Other than call sequence generation, some blackbox techniques focus on improving the operation input parameter generation [22, 24, 44]. These techniques utilize predefined inter-parameter constraints, input grammar, or mutators to generate diverse input parameters for the test cases. Input parameter generation is orthogonal to call sequence generation, we plan to adopt advanced input parameter generation in the future.

Whitebox RESTful service testing techniques. Evomaster [7] is a whitebox RESTful service testing technique which instruments the target RESTful service and monitors its database to collect useful execution feedback and data to guide the evolutionary algorithm based test case generation. Comparing to the the blackbox techniques, on the one hand, Evomaster is more effective in testing deeper logic inside the RESTful service since it can collect and use more information about the target service to guide the test case generation. On the other hand, Evomaster can only instrument Java/Scala/Kotlin based RESTful services and requires access to the database, limiting its application to closed source projects or projects implemented with other programming languages.

SOAP Testing Techniques Simple Object Access Protocol (SOAP) is a standards-based web service access protocol first proposed by Microsoft. Some previous works focus on testing SOAP [18] based web services [15–17]. Specifically, some of them use Web Services Description Language (WSDL) specifications to guide the testing process [12, 23, 30, 33, 35, 43]. Despite the similarities, REST is proposed to address the shortcomings of SOAP and is gaining more popularity in recent years [21]. The fundamental service interaction models in REST and SOAP are different. Therefore, testing RESTful services requires different strategies.

Model-based testing techniques. Besides the techniques for RESTful API and SOAP service testing, model-based testing techniques [13, 14, 36, 48, 49] are also related to Morest. In general, these techniques use models, say a finite-state machine [29], to describe the system-under-test and generate tests by covering different states in the model [45]. In particular, Palulu [8] is a technique developed based on Randoop [40], which uses a call-sequence model to quickly generate legal but behaviorally-diverse tests for Java classes. Palulu's intuition of using a model to guide the generation of valid API call sequences is similar to Morest's. However, the difference is that the model used in Morest is tailored for RESTful API and it is dynamically updated during testing.

6 CONCLUSION

In this paper, we propose Morest — a model-based blackbox RESTful API testing technique. Morest learns from the OpenAPI specifications to build a RESTful-service Property Graph (RPG), which encodes both schema and API operation information. Morest can use RPG to provide high level guidance for call sequence generation and continuously refine the RPG with execution feedback. We evaluated Morest on 6 open source RESTful services and the results showed that Morest can significantly outperform state-of-the-art techniques in both coverage and bug detection.

ACKNOWLEDGEMENTS

This research is partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRFI06-2020-0022-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001. This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] danmayer/coverband: Ruby production code coverage collection and reporting (line of code usage). https://github.com/danmayer/coverband.
[2] jacoco/jacoco: Java code coverage library. https://github.com/jacoco/jacoco.
[3] Mann–whitney u test - wikipedia. https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test.
[4] Morest. https://sites.google.com/view/restful-morest/home.
[5] Php code coverage for your web/selenium automation · tech adventures by tarun lalwani. https://tarunlalwani.com/post/php-code-coverage-web-selenium/.
[6] api blueprint. API Blueprint. A powerful high-level API description language for web APIs., 2020.
[7] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), January 2019.
[8] Shay Artzi, Michael D. Ernst, Adam Kie. Zun, Carlos Pacheco Jeff, and H. Perkinsmit Csail. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *In M-TOOS*, page 2006, 2006.
[9] Atlassian.
[10] Atlassian. Bitbucket, 2020.
[11] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019.
[12] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Ws-taxi: A wsdl-based testing tool for web services. In *2009 International Conference on Software Testing Verification and Validation*, pages 326–335, 2009.
[13] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. Search based path and input data generation for web application testing. In *International Symposium on Search Based Software Engineering*, pages 18–32. Springer, 2017.
[14] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 142–153, 2019.
[15] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing web services: A survey. 01 2010.
[16] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.
[17] Gerardo Canfora and Massimiliano Di Penta. *Service-Oriented Architectures Testing: A Survey*, pages 78–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
[18] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *Internet Computing, IEEE*, 6:86 – 93, 04 2002.
[19] Docker. Docker: Empowering App Development for Developers, 2020.
[20] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot. Automatic generation of test cases for rest apis: A specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–190, 2018.
[21] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.
[22] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. *Intelligent REST API Data Fuzzing*, page 725–736. Association for Computing Machinery, New York, NY, USA, 2020.
[23] S. Hanna and M. Munro. Fault-based web services testing. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 471–476, 2008.
[24] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, pages 45–48, 2018.
[25] Amazon Inc. Amazon, 2020.
[26] Google Inc. Google, 2020.
[27] Twitter Inc. Twitter, 2020.
[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.
[29] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.
[30] Yin Li, Zhi-an Sun, and Jian-Yong Fang. Generating an automated test suite by variable strength combinatorial testing for web services. *Journal of Computing and Information Technology*, 24:271–282, 11 2016.
[31] Yi Liu. Jsoptimizer: an extensible framework for javascript program optimization. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 168–170. IEEE / ACM, 2019.

[32] Yi Liu, Jinhui Xie, Jianbo Yang, Shiyu Guo, Yuetang Deng, Shuqing Li, Yechang Wu, and Yepang Liu. Industry practice of javascript dynamic analysis on wechat mini-programs. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 1189–1193. IEEE, 2020.

[33] C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai. Wsdl-based automated test data generation for web service. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 731–737, 2008.

[34] Clovity Marketing. Bitbucket Vs. GitHub: Who Hold's Your Company's Future?, 2020.

[35] Evan Martin, Suranjana Basu, and Tao Xie. Automated robustness testing of web services. 01 2006.

[36] Ali Mesbah, Arie Van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, 2011.

[37] Mozilla. Http response status codes.

[38] OpenAPI. OpenAPI Specification, 2020.

[39] OpenAPI. Swagger Petstore, 2020.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 75–84, USA, 2007. IEEE Computer Society.

[41] RAML. RAML: The simplest way to model APIs, 2020.

[42] Marko Rodriguez and Peter Neubauer. The graph traversal pattern. 04 2010.

[43] H. M. Sneed and S. Huang. Wsdltest - a tool for testing web services. In *2006 Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*, pages 14–21, 2006.

[44] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[45] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[46] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[47] E. Viglianisi, M. Dallago, and M. Ceccato. Resttestgen: Automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[48] Bing Yu, Lei Ma, and Cheng Zhang. Incremental web application testing using page object. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 1–6. IEEE, 2015.

[49] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 423–435. IEEE, 2021.