REST API Functional Tester

Diptikalyan Saha, Devika Sondhi
diptsaha@in.ibm.com,devika.sondhi@ibm.com
IBM Research, India

Swagatam Haldar
swagatam.haldar@student.uni-tuebingen.de
University of Tubingen, Germany

Saurabh Sinha
sinhas@us.ibm.com
IBM Research, USA

ABSTRACT

Software developers nowadays routinely publish REST APIs to make their applications accessible to clients. To support the development of REST APIs, many testing techniques and tools have been developed that automatically explore the API endpoints, using various strategies, with the goal of increasing coverage and detecting bugs. However, these techniques can produce test cases that may not be suitable for creating functional test suites, required for regression testing. In this paper, we address the problem of automatically generating realistic functional test cases for REST APIs. Our work is guided by interactions with practitioners and the approaches they follow for performing functional testing of REST APIs. Our technique first groups the API operations by the resources they manipulate; it then employs a novel approach for inferring producer-consumer relations between operations, fine-grained subtyping of HTTP methods, and language-model-based sequencing to construct operation sequences that correspond to functional scenarios. We evaluated the effectiveness of our technique and compared it against state-of-the-art REST API testing tools. Our results show that the technique produces test cases that can better support functional testing of REST APIs than the other tools.

1 INTRODUCTION

The advances in cloud computing and the increasing adoption of cloud-native application architectures over the past decade have led to a huge growth in the availability of web APIs. Enterprises now routinely publish such APIs to enable clients and application developers to access their services; online API catalogs, such as APIs guru [6], list thousands of APIs. Although web APIs can be developed in different architectural styles, the REpresentational State Transfer (REST) style [22] is the most popular of them [42]. APIs that adhere to the REST architectural style are called RESTful APIs or simply REST APIs. To enable their discovery and use, REST APIs are described using a formal specification language, such as OpenAPI [36], RAML [38], and API Blueprint [5].

Given the widespread use of REST APIs and the large number of applications that consume them, it is essential to test the APIs thoroughly before deployment. To address this problem, a number of automated test generation techniques and tools have been developed (e.g., [12, 16, 18, 20, 21, 25, 28, 29, 32, 39–41, 44, 45, 47]) that take as input the API specification and employ various strategies for generating test cases that exercise the APIs, with the goals of increasing coverage and detecting bugs [27], important for early phases of development. However, a common limitation of these techniques is that they are not suited for functional testing of REST APIs, where test suites are repeatedly used for regression testing, ensuring that the functionalities are not broken over changes.

In this work, we focus on enabling the automated creation of functional test suites for REST APIs, in the form of a tool that could be easily used in the industry. Our work is guided by, and grounded in, the functional testing practices of developers. To learn about these practices, we conducted a small survey of industry practitioners. Specifically, we interacted with four industry testing practitioners. A number of key activities involved in the functional testing of REST APIs came up in these conversations. For instance, practitioners said that a good functional test suite for REST APIs should consist of tests that exercise the individual operations of an API as well as sequences of operations that cover functional scenarios. Moreover, the functional scenarios typically revolve around create-read-update-delete (CRUD) operations on the resources managed by the API (e.g., account, product, order). This resource-centric view is essential for creating good functional test suites that, in practice, can be effectively employed for regression testing APIs as they evolve. We note that a resource-centered test case is not restricted to manipulating only one resource. Although such a test focuses on exercising a sequence of operations on a target resource r, the manipulation of r may require, as a prerequisite, other resources to exist. Thus, a test case focused on exercising r would contain the necessary prerequisite sequence as well.

Although existing test generation techniques for REST APIs (e.g., [16, 20, 29, 32, 45, 49]) target individual operations as well as sequences of operations—created based on producer-consumer relations, operation dependencies, and schema dependencies—they do not take this resource-centric view. Thus, they can generate operation sequences that do not respect resource boundaries and also miss sequences that are relevant for functional testing with respect to a resource.

To address the limitations of existing techniques, we present a novel black-box testing technique and tool called RAFT (REST API Functional Tester). RAFT takes as input the OpenAPI specification of the API under test (AUT) and automatically generates a test suite of functional test cases for the AUT; the generated tests cover individual API operations as well as sequences of operations in a systematic and well-defined manner. RAFT first analyzes the AUT specification to identify the resources that are managed by the AUT and groups the API operations based on the resources they manipulate. By doing this, it can explore the space of operation sequences one resource at a time.

To construct operation sequences for a resource, RAFT computes producer-consumer relations between operations. Our definition of producer-consumer relation is a refinement of existing definitions (e.g., [16, 29]) that restricts the relations so that they occur with respect to resources and also extends the relations to capture dependencies beyond the output of an operation and the input of another operation. We present a novel algorithm for computing resource-based producer-consumer relations. In creating operation sequences, RAFT also employs a fine-grained subtyping of HTTP methods (e.g., GET, POST, PUT, PATCH) based on whether an operation processes one or multiple instances of a resource. Moreover, for ordering operations with the same HTTP method, RAFT leverages a transformer language model to compute natural orders between verbs occurring in the operation names. From these analyses, RAFT constructs a per-resource operation graph to represent the possible operation orderings. It also adds prerequisite nodes to the graph and, finally, creates an operation sequence as the topological order of nodes in the operation graph.

We evaluated the effectiveness of RAFT in creating feasible functional test sequences and the contributions of its various components to its overall effectiveness. We assessed effectiveness using a set of metrics, including conventional coverage metrics as well as new metrics designed to measure functional coverage. We also compared RAFT against three existing REST API testing tools: EvoMaster [11], RestTestGen [46], and Morest [29]. Experimental results indicate that RAFT consistently generates more functional (on average 19% more than Morest) and compact test cases with better parameter (on average 26% more than RestTestGen) and operation coverage. Moreover, sequences produced by RAFT respect the producer-consumer relationships more than the other tools (on average 49% more than RestTestGen).

The main contributions of this work are:

- A new resource-based definition of functional testing for REST APIs.
- A novel approach for generating test cases for REST APIs that creates per-resource operation sequences by analyzing resource-based dependencies between operations, applying fine-grained HTTP method subtyping, and leveraging a transformer language model for operation ordering.
- Empirical results that demonstrate that RAFT is highly effective in generating functional test sequences and outperforms state-of-the-art REST API testing tools.

2 INDUSTRY REQUIREMENTS

To illustrate our approach, we use the Petstore API [37] as the running example. The API specification consists of 20 operations and six schemas (`Pet`, `User`, `Order`, `Category`, `APIResponse`, `Tag`) and will serve as our running example. Figure 1 lists the operations (Column 2) grouped by three resources (`Pet`, `Order`, and `User`) managed by the API.

As mentioned earlier, we developed our approach based on API testing approaches used in practice. We consulted with four testing practitioners and leaders from the industry. Two of these practitioners have four years of experience in application testing, one has 15 years of experience and has led various testing projects, and one is a strategic leader with twenty years of experience who leads the testing platform and testing strategy for an organization.

We discussed with them different ways in which functional test cases for Petstore could be created. For instance, a functional test case could chain together all operations in the specification while respecting parameter relationships (e.g., ⟨`addPet`, `placeorder`, `getPet`, `getOrder`, ...⟩). Another option is to create more fine-grained sequences by chaining all CRUD operations for related resources (e.g., ⟨`addPet`, `getPetById`, `updatePet`, `placeorder`, `getOrder`, `deleteOrder`, `deletePet`⟩). Yet another option, which is even more granular, is to create a sequence from per-resource CRUD order of operations; one such sequence can be ⟨`addPet`, `placeorder`, `getOrder`, `deleteOrder`⟩. The practitioners said that the last option aligns best with how they think about creating functional test cases for REST APIs, where a functional test case focuses on one resource. But that test case could include operations that create other resources, required as a prerequisite for testing the target resource. For example, in the second per-resource sequence illustrated above, the target resource for testing is `Order`, but the sequence also includes an operation for creating a pet, which is a prerequisite for creating any order.

Resource	Related Operations	Type	Subtype
root.pet	findPetsByStatus	get	get-all
	deletePet	delete	delete-one
	getPetById	get	get-one
	uploadFile	post	post-put
	findPetsByTags	get	get-all
	addPet	post	post-one
	updatePet	put	put-one
	updatePetWithForm	post	post-put
root	getInventory	get	get-all
root.store.order	placeOrder	post	post-one
	getOrderById	get	get-one
	deleteOrder	delete	delete-one
root.user	getUserByName	get	get-one
	updateUser	put	put-one
	logoutUser	get	function
	createUser	post	post-one
	createUsersWithArrayInput	post	post-all
	createUsersWithListInput	post	post-all
	deleteUser	delete	delete-one
	loginUser	get	function

Figure 1: Operations of the Petstore API grouped by resources, and the HTTP method type and subtype for each operation.

Another requirement that came out of the discussions was the capability of defining the pattern of HTTP methods to start a sequence. For many resources, the sequence can begin from a GET operation that retrieves many instances of a resource.

Next, practitioners mentioned that the ability to generate test cases in online and offline modes would be very useful in practice. The offline-mode test generation (in which the API implementation is not available for dynamic interaction) can help with bootstrapping test-suite development with just the API specification and encourage test-driven development of the APIs. This mode, however, cannot rely on dynamic feedback to guide test generation, which online test generation can benefit from.

Finally, they wanted the flexibility of selecting and changing the automatically bootstrapped test cases in the tool which can cater to specific application needs. The development of the RAFT was guided by these interactions with and requirements from, practitioners.

3 OUR TECHNIQUE

RAFT takes two inputs: an OpenAPI specification with additional precondition/post-condition to execute the API (defined by *Spec*) and, optionally, the API run-time parameters through which the API can be executed. RAFT produces three outputs: a set of test cases, execution results for the test cases, and coverage results. We

TOp	addPet
TVal	addPet.body.name = 'scallop'
TCond	$addPet.responsecode \in [200, 399]$
TPC	placeOrder.body.petId=addPet.200.id
	updatePetWithForm.path.petId:addPet.200.id
	uploadFile.path.petId:addPet.200.id
TOp	uploadFile
TVal	uploadFile.formData.file=File('file.jpg')
TCond	$uploadFile.responsecode \in [200, 399]$
TOp	updatePetWithForm
TVal	updatePetWithForm.formData.name='bon'
	updatePetWithForm.formData.status='available'
TCond	$updatePetWithForm.responsecode \in [200, 399]$
TOp	placeOrder
TVal	placeOrder.body.status='placed'
TCond	$placeOrder.responsecode \in [200, 399]$
TPC	getOrderById.path.orderId=placeOrder.200.id
	deleteOrder.path.orderId=placeOrder.200.id
TOp	getOrderById
TVal	
TCond	$getOrderById.responsecode \in [200, 399]$
TPC	
TOp	deleteOrder
TVal	
TCond	$deleteOrder.responsecode \in [200, 399]$
TPC	-

Figure 2: An example test case for Petstore.

first present formal notations for the input and output of RAFT and then describe the technique.

3.1 Notation

API Specification. An API specification is a triple $Spec := \langle O, S, P\rangle$ where O is the set of operations (also called endpoints), S is the set of schemas, and P is the set of pre- and post-conditions for the API execution such as authentication details. Each operation $o \in O$ is represented as $\langle opname, path, tag, type, IP, OP, GC\rangle$ as described below:

- *opname* is an unique name for the operation. This is typically presented in OpenAPI specification as *operationid*.
- *path* is the relative URL for accessing the operation.
- *tag* is an optional set of strings to tag the operation.
- *type* is the HTTP method for RESTful operation and typically contains POST (create), GET (read), PUT (update/replace), PATCH (update/modify), and DELETE (delete) [36].
- *IP* is the set of input parameters (top-level or through transitively unrolled schema). Each input parameter $inp \in IP$ is represented as a 6-tuple $\langle pname, ptype, is\_required, loc, pc, id\rangle$, where *pname* is the parameter name, *ptype* is the parameter type which can be either primitive type [36] or a schema, *is_required* denotes if the parameter is mandatory, *loc* can be one of `BODY`, `PATH`, `QUERY`, `HEADER`, and `FORMDATA`, and *pc* uniformly captures various constraints, such as range, format (email, ipv4), etc. in Z3 [34] compliant form.
- *OP* is a set of output parameters similar to IP but has *responsecode* and *responsetime* as an additional property for each parameter. $o.responsecode$ denotes the HTTP response code at runtime.
- *GC* stands for global constraints such as inter-parameter dependencies [30].

Each schema $s \in S$ is represented as $\langle sname, SF\rangle$ where *sname* represents the schema name and *SF* is the set of fields of that schema represented as $\langle fname, ftype, is\_required, fieldconstraint\rangle$; these are same as the components of the input parameter.

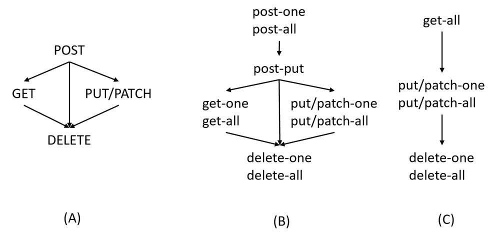

Figure 3: (A) CRUD partial order, (B) subtype partial order, and (C) alternate order in the absence of POST operations.

Test Cases. A test case T is a sequence of $\langle TOp, TPC, TVal, TCond\rangle$. Each $TOp_i \in O$, $TVal_i$ maps a subset of input parameters $TOp_i.IP$ to concrete values, $TPC_i \subseteq \{TOp_i.OP \times TOp_j.IP \mid i < j\}$ (written as $TOp_j.IP = TOp_i.OP$) is an optional parameter that denotes the mapping between some of the output parameters of an operation (producer) to the input parameters of other operation (consumer) later in the sequence, and finally, $TCond_i$ is a boolean expression on the output of ($TOp_i.OP$).

Intuitively, a test engine executes a test case by sequentially executing each operation TOp_i, instantiating its parameter values using $TVal_i$, executing it and receiving the response, and checking the condition $TCond_i$ on the response. If the check is satisfied, it further updates its state by extracting the parameter values in $TOp_i.OP$ from the response to instantiate the consumer's parameters ($TOp_j.IP$).

A test instance corresponding to a test case T is a sequence of operations in T with concrete values for all its input parameters and output parameters. A failure test instance is one where one of $TCond_i$ has failed.

The RAFT output consists of a set of test cases one of which is shown in Figure 2 along with test instances. One can easily see the sequence of operations ⟨`addPet`, `updatePetWithForm`, `uploadFile`, `placeOrder`, `getOrderById`, and `deleteOrder`⟩ which performs `Order`-related operations with the prerequisite of creation of a `Pet` instance.

3.2 Sequence Generation

To perform functional testing, RAFT first generates test cases to test each operation (akin to unit testing). These sequences contain the required prerequisites followed by the intended test operation, e.g., `addPet` is a prerequisite for `getPetById`. Such sequences are shown in Figure 4(A). We discuss later how such prerequisites are computed. Such sequences are also generated in a recent work [47].

Subsequently, RAFT generates functional sequences for each resource to check the feasibility of the functionality. Formally, a sequence of operations is said to be *complete* w.r.t a resource if it contains all `types` of operations (i.e. at least one operation of each `type`) related to the resource and *sound* if such operations maintain the CRUD partial order (Figure 3(A)). A resource is said to be *functionally covered* if there exists at least one sequence (called a functional sequence) in the test suite which is sound and complete w.r.t the resource. The functional sequences for each resource are shown in Figure 4(B). Note they also contain the prerequisite operations (in italics) as seen for resource `root.store.order`. To the best

Operation-level
addPet
addPet, deletePet
addPet, findPetsByStatus
addPet, findPetsByTags
addPet, getPetById
addPet, placeOrder
addPet, placeOrder, deleteOrder
addPet, placeOrder, getOrderById
addPet, updatePet
addPet, updatePetWithForm
addPet, uploadFile
createUser, deleteUser
createUser, getUserByName
createUser, updateUser
createUsersWithArrayInput
createUsersWithListInput
getInventory

(A)

Resource	Sequence
root.store.order	*addPet*
	uploadFile
	updatePetWithForm
	placeOrder
	getOrderById
	deleteOrder
root.pet	addPet
	uploadFile
	updatePetWithForm
	getPetById
	updatePet
	deletePet
root.user	createUser
	getUserByName
	updateUser
	deleteUser

(B)

Figure 4: (A) Operation-level sequences and (B) resource-based sequences.

of our knowledge, we are the first to propose the resource-based functional sequence generation by design. Our first algorithmic challenge is therefore to associate operations to resources and group the operations (to create sequences later) on the basis of such relationship.

Resource identification and Resource-Operation Relationship. There are various options for identifying resources in an API specification. For example, the schema present in the input-output parameters can be considered a resource. Other alternatives could be user-provided tags. However, based on our experience, operation paths are the best indicators of resources. For example, all operations in the second functional sequence tagged as `root.pet` have a common prefix `/Pet/`.

Recent related works such as Morest [29] capture the relationship between operation and schema. Here we distinguish between the schema and resources. As defined in Section 3.1, schema refers to the syntactic entity used to define the input and output parameter types, whereas, resources is a logical entity. For example, the three schemas `Category`, `APIResponse`, `Tag` in Petstore are not resources. A common practice (as also seen in Person-controller benchmark [2]) is to represent the request and response as two schemas (e.g., `PersonReq`, `PersonRes` in Person-controller) corresponding to the logical entity `Person`.

Our algorithm identifies a resource by the maximal length path (removing versions and path parameters) that ends in a noun (resource names are typically nouns) in the path component (a substring of the path between consecutive '/'s) or has multiple operations in it. The operations for the resource are all operations that are associated with the suffix of any path that has the resource as a prefix. We capture this association in the resource-operation (*RO*) relationship shown in Figure 1 (columns 1 and 2).

Once we obtain the group of operations through $RO : R \mapsto O$ relationship, there are two additional challenges: how to order these operations and how to create prerequisites sequences for the operations. We address these challenges based on a few types of dependencies, described below.

Resource-based producer-consumer dependency. The resource-based producer-consumer dependency (*RPC*) occurs between an operation that creates the resource (the producer) and the operation that accesses/uses the resource (the consumer). The consumption is done through the *id* field of a resource, which uniquely identifies a resource. There are three instances of such a relationship in our sample test case - `placeOrder` (consumer) requires a `Pet` (producer) where `placeOrder.body.petId` (consumed parameter) refers to `addPet.200.id` (produced parameter). A similar relationship exists between `placeOrder` and `getOrderById` and between `placeOrder` and `deleteOrder` where `placeOrder` acts as a producer for resource `Order`.

We now explain why it is not important to match all fields between operations which many related works do using the example of `updateUser`. A PUT operation such as `updateUser` uses non-id fields such as firstname, lastname to update the existing resource and, therefore, there is no need to carry forward their old values which non-id field relationship entails. RAFT finds the producer-consumer relationship between three producers `createUsersWithArrayInput`, `createUser`, `createUsersWithListInput` and the consumer `updateUser` for two inputs fields `updateUser.path.username` and `updateUser.body.id`.

Data availability (*DA*) relationship. A consumer operation can directly access a resource through its *id* parameter which is captured in the previous relationship. In addition, a consumer can also search for an object through other non-id fields. The success of these consumer operations at runtime depends on whether it finds the object. This is ensured by having equality relationship between the producer's input or output to the consumer's input. An example of such a relationship exists between `addPet.200.status` and `findPetsByStatus.query.status[]`.

Relationship Inference. There are five algorithmic challenges in implementing these relationships between parameters of different operations: (1) identifying producer operations, (2) determining 'id' fields of a resource, (3) which fields to match, (4) restriction of parameter relationships between operations, and (5) matching parameters.

Producers. As stated earlier, the producers are the operations that create the resource by generating the id-field. These operations are usually POST operations. However, our experience suggests that developers use POST operations not only to create resources but to update the resource and perform other functional operations. To obtain producers and for creating sequence (we will see later), we need a fine-grained sub-typing of operations. There are three notions of operation sub-typing:

- Cardinality: Whether the operation works on a single resource or multiple resources. For example, `createPet` creates one resource whereas `createUsersWithArrayInput` and `createUsersWithListInput` create multiple resources. This can be easily inferred by the presence of schema arrays in either input or output. We use *one* and *all* suffixes with operation types to denote these categories of subtypes.

- CRUD-Anomaly: Let us consider the POST operations `uploadFile` and `updatePetWithForm` that update the `Pet` resource.[1] Such anomaly happens in practice and our algorithm uses input-output pattern-based matching and keyword-based matching to find such an anomaly. `updatePetWithForm` takes `petId` as a parameter and `uploadFile` does not return an *id*. Note that, the above operations are subtyped as 'post-put' and they are not considered for producers as they are not creators of resources.
- Functional Operation: Consider the GET operations `loginUser` and `logoutUser`. The verb in these operations does not match with any synonym of retrieval operation and therefore they are considered functional operation which is more relevant to application functionality than the CRUD functionality.

The last column of Figure 1 shows all the operation subtypes. The producer is defined below:[2]

$$producer(o1)\ if\ o1.subtype \in \{post\text{-}all, post\text{-}one\}$$

Id-fields/Id-Param. The id-fields are fields of a schema. Analysis of databases or workload can give a better idea of unique fields in a schema, however, we do not assume the availability of such information and therefore resort to static analysis of API Specification. We use a preference order (1) type *UUID*, (2) names containing substring such as *id*, *code*, *name*, *timestamp*, and (3) in their absence, the most frequently accessed name matching field from other operations to select the best id field. This simple heuristics works surprisingly well in practice.

Consumer parameter selection for matching. Given an operation (i.e., a potential consumer) we always try to find a match for all its path and query parameters with the producer's output params and only id fields of any schema used in the input if such not match is found with the output params.

Producer-consumer operation restriction . The producer and consumer cannot be the same operation. If there are multiple producers of the same resource, those operations cannot be involved in the same relationship.

Parameter matching. Given the selection of parameters described above, we use the following features to match the consumer and producer parameters: (1) parameter name, (2) parameter type, (3) nouns in schema name (if any), (4) nouns in the resource name, and (5) resource. Consider the consumer field `placeOrder.body.petId` with features ⟨petId, integer, Order, Order, 'root.Order'⟩ and its corresponding matched producer field is `addPet.200.id` with features ⟨id, integer, Pet, Pet, root.Pet⟩. Given two fields with their features, RAFT uses a matching function that returns a rank. The rank is a weighted addition of the following matches: (1) type equality match (weight W1) (2) field names match (exact or substring or acronym) (W2) (3) schema or resource names match (W3), (4) exact and acronym match between nouns in consumer field name to schema name/resource name match (W4) (5) producer and consumer resource match if the consumer operation is a non-post operation (W5). We give the following weights: W1=500, W2=50, W3=20, W4=W5=10, prioritizing type match, param name match, and resource-name match in that order. The intuition for (4) is that consumers will typically use the schema/resource name in the field name to unambiguously identify the correct resource. Without this match, we do not consider the producer-consumer relationship across resources. We use the Ronin splitting algorithm from Spiral [3] to break the words and use Flair pos-tagger [1]. Note that one consumer field can match with fields from multiple producers (if exists) of the same resource (e.g. `createUser` and `createUsersWithArrayInput`).

[1]Beyond Petstore, we have seen a PUT operation semantically act as a GET operation because the developer wanted to send an object in the request body.
[2]In RAFT, the subtype list is user-configurable. Users typically include *get-all* if no POST operation exists for a resource but has other types of operations. The benchmark market is an example of this.

Resource-based Operation Sequencing. As stated earlier, we group the operations based on resources. For each resource, we essentially order the related operations based on the inherent partial order present in CRUD semantics, shown in Figure 3(A). This partial order is translated to a subtype-based partial order relationship shown in Figure 3(B). Practical experience also suggests an alternative start from *get-all* when POST operations are not present for a resource as shown in Figure 3(C).

RAFT takes a three-step process for resource-based sequence generation: (1) operation graph per resource, (2) prerequisite addition, and (3) ordering.

Operation Graphs per resource. Given a resource and all operations related to it, one operation graph is created by adding edges between the operations for each subtype-partial order relationship. For our example test case, this step will add $placeOrder \longrightarrow getOrderById$, $placeOrder \longrightarrow deleteOrder$, and $getOrderById \longrightarrow deleteOrder$.

Prerequisite addition. Next, RAFT introduces the producer nodes (if it does not exist already) for consumers in the operation graph from both *DA* and *RPC* relations, and recursively adds the producers of the newly added nodes along with their edges. However, if there exist multiple producers for the same consumed field, then multiple copies of operation graphs are created having different producers. $addPet \longrightarrow placeOrder$ is added using *RPC*.

For any *post-one* operation added as a prerequisite, RAFT also adds (only for resource-based sequencing and not for operation-based) the *post-put* operations (and recursively their dependencies) for the same resource. This is based on our assumption that post-put operations are meant for required initialization (in contrast to PUT operation which is not essential for resource creation and setup). $addPet \longrightarrow uploadFile$, $addPet \longrightarrow updatePetWithForm$ is added through this.

Ordering. Three types of ordering edges are added to the graph. For multiple operations having the same subtype for a resource, we use a novel strategy for ordering the verbs of each operation based on their natural order. RAFT uses a transformer language model to find the permutation of the verbs with the highest probability. This will order `uploadFile` before `updateWithForm`. Even though login and logout are not part of execution in Petstore, this strategy will correctly identify the order ⟨`login`, `logout`⟩.

To break the partial order between GET, PUT, and PATCH operations for resource, we add the ordering edges between operations in the type order ⟨GET, PUT, PATCH⟩. For any operation edges between post-one and post-put operations added in the prerequisite,

Table 1: Benchmark services used in the evaluation.

API	Operations	Params	GET, POST, PUT, DEL	Schemas
petstore [37]	20	72	8, 7, 2, 3	6
person [2]	12	47	5, 2, 2, 3	6
gestao [24]	20	87	10, 5, 3, 2	6
restful web [2]	33	41	19, 5, 2, 5	17
user mgmt. [2]	23	64	6, 7, 2, 5	12
market [2]	13	50	7, 2, 3, 1	13
project cont. [2]	8	26	2, 3, 1, 2	4
car [4]	14	102	13, 1, 0, 0	24
langtool [43]	2	11	1, 1, 0 , 0	5
xkcd [48]	2	1	2, 0, 0, 0	1
scs [2]	11	26	11, 0, 0, 0	0
ncs [2]	6	14	6, 0, 0, 0	1
rest countries [2]	22	34	22, 0, 0, 0	0

to keep them subsequent we also add outgoing edges from post-put operation to the same target operations of post-one. RAFT generates the final sequence based on the topological order of operations.

3.3 Parameter and Data Scenarios

For each operation, we create three types of parameter scenarios: all mandatory parameters, all parameters, and each optional parameter along with required mandatory parameters. Operation-level test cases explore all parameter scenarios for target operation but only mandatory parameter scenario for the prerequisites. For functional sequences, only mandatory parameters are considered for all operations. For each parameter, the data is automatically generated by respecting all constraints, types, and examples specified in the specification using Z3 constraint solver. RAFT includes a semantic specification extension to denote an input parameter to be unique. We specify id parameters as unique. For one parameter scenario, only one set of data values is generated.

4 EXPERIMENTAL EVALUATION

4.1 Benchmark and Setup

We selected 13 public APIs, that have been used in past research ([2, 9]). We used these as benchmarks to evaluate RAFT. Table 1 lists the benchmarks, along with details of number of operations, request parameters, number of schema definitions, and distribution of operations over various REST calls. We also scanned through API gurus [6], but many of the services were found to be either behind paywall or inaccessible, and also unfit for our algorithms, e.g., due to lack of POST operations. Table 1 divides the benchmarks into APIs two groups—those with POST operations (top) and those without (bottom). Note that languagetool and car have one POST call each, but they are unrelated to any resource.

All experiments were run on a machine with Microsoft Windows 11 OS, with 32 GB RAM and 11th Gen Intel Core i7-1165G7 2.80GHz processor.

4.2 Research Questions

We investigated the following research questions.

- **RQ1:** How does RAFT compare with other tools on functional test case generation based on the definition of per-resource-based CRUD ordering?
- **RQ2:** How does RAFT compare with other tools in terms of operation, parameter, and sequence coverage?
- **RQ3:** How does RAFT compare with the state-of-the-art tools in computing correct producer-consumer relations?
- **RQ4:** How does RAFT's effectiveness in computing producer-consumer relations vary under different algorithmic choices, such as other resource-operation grouping strategies and other parameter-matching functions?

4.3 Metrics

We define the metrics in three categories.

- *Functional Test Cases*: Given a set of test cases, each containing a sequence of operations and a set of successful test cases (ending with status code 2xx), we compute three metrics related to functional test case generation. The first metric, called functional coverage (*FC*), is the percentage of resources functionally covered. The second metric, *CFC*, measures the compactness of the functional sequences; it is computed as the ratio of the span of operations for a covered resource and the number of unique operations in the sequence related to the resource, averaged over all functional sequences. This metric has values ≥ 1 with values closer to 1 indicating more compact functional test cases. The third metric, *SFC*, is the percentage of successful functional sequences.
- *Coverage*: Parameter coverage (*Pc*) denotes the percentage of input parameters covered by the test suite. Sequence coverage metric *s2* denotes the percentage of all sequences that complete successfully. Operation coverage (*oc*) measures the percentage of operations executed with respect to the total operations in the API specification. Operation instance coverage metrics (*oik* for $k \in \{2, 4, 5\}$) denote the percentage of operation instances across all test executions that return kxx response code.
- *Producer-Consumer Relations*: We compute two metrics related to the producer-consumer relationships.[3] We manually constructed the ground truth (i.e., the set of correct producer-consumer relationships). For each service, one author manually identified all resources and their producer operation and output id parameters. Then, all path and query parameters and input id parameters were matched against the first set. In case of confusion, the correct matching was established by running the APIs. The first metric called producer-consumer accuracy (*PCAcc*) is the percentage of correct producer-consumer relationships. The second metric (*PCSeq*) is the percentage of sequences where the correct producer-consumer relationship is maintained, i.e., if a sequence contains a consumer, it should contain at least one corresponding producer preceding it.

4.4 Testing Tools

To investigate the first three RQs, we compare our tool with three state-of-the-art API testing tools, MOREST [29], EvoMaster (v1.6.0) [14], and RestTestGen [20, 46]. These tools have been executed with their default configurations. Note that as the focus of this paper is on the evaluation of functional testing, we have considered the results

[3]We use the term producer-consumer relationship to denote both RPC and DA relationships.

Table 2: Effectiveness in creating functional test sequences.

API	RAFT			MOREST			EvoMaster			RestTesetGen		
	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC
petstore	**100**	**1.0**	**100**	50	**1.0**	**100**	50	**1.0**	50	50	**1.0**	**100**
person	**50**	**1.0**	0	**50**	**1.0**	0	0			0		
gestao	**100**	**1.0**	52	**100**	**1.0**	**73**	55	**1.0**	0	66	**1.0**	2
restful	**91**	**1.0**	0	83	1.1	86	58	**1.0**	**100**	58	**1.0**	0
user	**87**	**1.0**	80	-	-	-	-	-	-	37	**1.0**	2
market	71	**1.0**	0	**85**	**1.0**	0	71	**1.0**	0	71	**1.0**	0
project	**100**	**1.0**	0	33	**1.0**	0	33	**1.0**	0	33	**1.0**	0
	86	1.0	33	67	1.02	43	44	1.0	30	45	1.0	17

(A) All Sequences

API	RAFT			MOREST			EvoMaster			RestTesetGen		
	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC
petstore	**100**	**1.0**	**100**	0			0			0		
person	**50**	**1.0**	0	**50**	**1.0**	0	0			0		
gestao	**100**	**1.0**	66	**100**	3.2	0	0			0		
restful	**87**	1.1	0	75	1.1	82	37	**1.0**	**100**	37	**1.0**	0
user	**80**	**1.0**	50	-	-	-	-	-	-	0		
market	50	**1.0**	0	**75**	**1.0**	0	50	**1.0**	0	50	**1.0**	0
project	**100**	**1.0**	0	0			0			0		
	81	1.01	31	50	1.6	21	15	1.0	50	12	1.0	0

(B) Non-trivial Sequences

of only the NominalFuzzer in the case of RestTestGen. EvoMaster and MOREST gave errors for the user-management service.

4.5 Results

4.5.1 RQ1. In this study, we compare RAFT with the state-of-the-art tools on the ability of functional sequence production and characteristics. The result is shown in Table 2, with bold-faced values representing the non-zero maximal solution. We selected the 7 benchmarks that have non-empty producer-consumer relations. We show results for two cases: (A) functional sequences covering all resources and (B) non-trivial functional sequences covering resources having at least two operations. If the FC value is zero, CFC and SFC cannot be computed and are shown as blank in the table.

Comparison of FC metric for the tools suggests that the RAFT performs better or the same for six of the seven services on producing functional sequences (non-trivial or all) as defined in Section 3.2. Our tool improves the functional coverage of MOREST by 93% (average of percent pairwise improvements). The low FC value of RAFT for person-controller is attributed to the low producer-consumer accuracy as explained in RQ3 results later. Many zero values for FC in state-of-the-art tools suggest these tools cannot consistently produce non-trivial functional sequences.

The average CFC value is 1.01 for RAFT and 1.6 for MOREST for producing non-trivial functional sequences. RAFT consistently produced more compact functional test cases. Other tools are not consistent.

For generating a successful functional sequence, MOREST is better (avg. 33%) than RAFT (avg. 43%) for all types of functional sequences including trivial ones. However, for producing non-trivial sequences RAFT (avg. 31%) is much better than MOREST (avg. 21%).

RAFT is considerably better and more consistent than MOREST, EvoMaster, and RestTestGen in generating compact functional test sequences.

4.5.2 RQ2. Table 3 presents the metrics for each of the tools, with the best-performing tools highlighted in boldface. The table indicates that RAFT performs better or at par with respect to other tools in most of the benchmarks, with *oc* and *Pc* values being the best all across, indicating the test inputs generated by RAFT gives the highest operation and parameter coverage. Furthermore, the successful run of operation instances ((*io2*)) is also observed to be the highest in the case of RAFT, followed by MOREST. This observation is indicative of tests generated by RAFT being functional in nature. RAFT is thus better suited for creating functional test suites, whereas the other tools are better suited for bug detection by generating negative or invalid tests, as indicated by the higher number of operations resulting in 4xx or 5xx responses observed from the other tools.

We also observed some 4xx responses on running RAFT on the benchmarks; 4xx is usually indicative of invalid inputs. On closely analysing the reason behind these 4xx, we made the following observations.

- We observed 401 responses due to unauthorized access or a subscription requirement from the user, as in the case of market API and car API, which is not exactly related to the input values passed as being non-functional.
- Loosely defined specification. Many data value constraints were missing from the specification due to which invalid parameter values were generated for car and restcountries APIs. For example, `horsepower_hp` must be a positive number although the specification mentions it to be a string.
- Most of the 4xx's observed in the case of project API can be attributed to its low producer-consumer accuracy.
- Language tool API enforces an inter-parameter constraint for one of the endpoints where two optional parameters are stated but at a time exactly for one parameter a value should be specified, while RAFT produces scenarios where for none of the parameters the value is specified, in addition to producing other parameter scenarios as well. As RAFT's algorithm explores various possible parameter scenarios, such a case is bound to get generated unless we support a way to process inter-parameter constraints that a human should specify along with the specification.

RAFT achieves greater parameter, operation, and successful operation coverage than MOREST, EvoMaster, and RestTestGen.

4.5.3 RQ3. We perform two studies in RQ3. The first study checks the accuracy of the producer-consumer relationship compared to the manually created ground truth for the seven benchmarks that have a non-empty producer-consumer relationship based on our definition of producers. The result is shown in Column 2 of Table 4. Except for two benchmarks—person and project controller—RAFT has near-perfect accuracy. For person-controller, the majority of misses are due to the mismatch between the consumer and producer parameter name (`persons` and `timestamp` respectively).

Table 3: Parameter, operation, and sequence coverage achieved.

API	RAFT						MOREST						EvoMaster						RestTesetGen					
	Pc	oc	s2	oi2	oi4	oi5	Pc	oc	s2	oi2	oi4	oi5	Pc	oc	s2	oi2	oi4	oi5	Pc	oc	s2	oi2	oi4	oi5
petstore	**100**	**100**	**100**	**100**	**0**	0	80	95	97	96	1	**3**	88	**100**	50	51	48	2	56	**100**	6	6	94	0
person	**100**	**100**	**61**	**62**	**0**	26	50	92	1	2	**0**	**99**	50	67	50	46	**0**	55	50	**100**	11	11	**0**	89
gestao	**100**	**100**	**82**	**92**	**8**	0	77	**100**	51	35	36	**30**	69	85	8	8	92	0	77	**100**	1	1	98	1
restful	**100**	**100**	47	**53**	12	4	28	**100**	8	8	**9**	**83**	72	**100**	**50**	48	34	18	72	**100**	0	0	**100**	0
user	**100**	**100**	**29**	**40**	**0**	39	-	-	-	-	-	-	-	-	-	-	-	-	98	**100**	1	1	6	**93**
market	**100**	**100**	**8**	**51**	24	20	77	**100**	0	0	**0**	**100**	53	**100**	0	0	71	29	77	**100**	0	0	57	43
project	**100**	**100**	0	14	39	0	30	63	**46**	**46**	**0**	**54**	60	100	19	19	69	13	60	**100**	2	2	86	12
car	**100**	**100**	**57**	**57**	**42**	**1**	-	-	-	-	-	-	3	43	25	14	86	0	-	-	-	-	-	-
langto.	**100**	**100**	80	80	20	0	0	50	**100**	**100**	**0**	0	0	50	**100**	**100**	**0**	0	0	100	8	8	92	0
xkcd	**100**	**100**	**100**	**100**	**0**	0	0	**100**	**100**	**100**	**0**	0	**100**	**100**	**100**	90	10	0	**100**	**100**	71	71	29	0
scs	**100**	**100**	**100**	**100**	**0**	0	**100**	**100**	**100**	**100**	**0**	0	**100**	**100**	**100**	**100**	**0**	0	**100**	**100**	98	98	2	0
ncs	**100**	**100**	50	50	50	0	**100**	100	**100**	**100**	**0**	0	**100**	**100**	60	60	40	0	**100**	**100**	55	55	46	0
restc	**100**	**100**	39	39	61	0	94	96	**73**	**73**	**0**	**27**	85	**100**	15	20	79	1	**100**	**100**	6	6	94	1
	100	**100**	58	**65**	20	7	58	91	**62**	60	**4**	**36**	65	87	48	46	44	10	74	100	22	22	59	20

Table 4: Effectiveness in computing producer-consumer relations.

API	PCAcc	PCSeq			
	RAFT	RAFT	MOREST	EvoMaster	RestTesetGen
petstore	100	**79**	21	10	30
person	57	**100**	37	37	50
gestao	100	**91**	2	0	15
restful	100	33	1	35	**36**
user	92	**100**	-	-	36
market	100	**100**	2	15	15
project	50	**50**	40	25	25
	86	79	17	20	30

For the project-controller benchmark, RAFT confuses between `project-code` and `problem-code` as the API uses a non-standard path-parameter convention: `code` in resource path `/api/problem/{code}` refers to the project code.

To compare tools that do not explicitly output the producer-consumer relationship, we checked how many of the ground truth producer-consumer relationships are respected in the generated sequences. The results are shown in Columns 3–6 of Table 4. RAFT does better on six of the seven services by a huge margin. This shows other tools are not suitable for testing sequences that involve producer-consumer relationships.

RAFT achieves much greater accuracy in computing producer-consumer relationships than MOREST, EvoMaster, and RestTestGen.

4.5.4 RQ4. We performed ablation studies to see the effect of our algorithmic choices on producing resource-based dependency relations. We use *PCAcc* metric to compare all these different cases with RAFT . We use the same set of seven benchmarks as we select for RQ3. Table 5 shows the results. The second column (RAFT) is replicated from Table 4 for ease of comparison. The ablation studies are explained below:

- ParamMatch: Instead of RAFT's weighted matching function which takes five features, we change the matching function only to perform an equality match between producer and consumer parameter names along with the resource path match. The result shows a 10% average reduction in relationship accuracy. This shows that resource and schema names play an important role beyond parameter name matching.
- OnlyParam: We created a parameter relationship inference algorithm that (1) considers equality of the input and output parameter names with the restriction that producer and consumer operations cannot be equal, (2) considers only POST operations as producers, and (3) makes a random choice in case of multiple matches to the consumed parameter. The result shows 82% average drop in *PCAcc*. This shows the importance of the matching function along with the importance of including producer input parameters, handling schema fields, and path/query parameters separately.
- PostRO: In this study, we change the resource identification algorithm to consider a path as a resource if it has a post request in the path. This considerably decreased the *PCAcc* values in two benchmarks—*Petstore* and *market*. For *Petstore*, the PostRO strategy incorrectly associates `createUsersWithArrayInput` and `createUsersWithListInput` operations with resources other than the `root.user` resource, which has `createUser` and therefore gives lower preference to them than `createUser` as producer.
- LLM: We also did a study to see the effect of the LLM-based ordering in sequences and found that it performs sequence changes only one (petstore) of the seven benchmarks.

Table 5: Results of the ablation study.

API	PCAcc			
	RAFT	ParamMatch	OnlyParam	PostRO
petstore	100	76	0	66
person	57	57	14	57
gestao	100	92	7	100
restful	100	89	10	100
user	92	78	0	100
market	100	91	83	25
project	50	50	0	50

The resource-identification and relationship-inference algorithmic choices of RAFT are effective whereas the applicability of LLM-based ordering in the context of other deterministic ordering is limited.

4.6 Threats to Validity

- We have manually created the gold standard for the producer-consumer relationship which may have errors.
- Our notion of resource-based functional test case generation can be subjective and can differ between practitioners.
- EvoMaster, MOREST, and RestTestGen have been used with their default configuration which may not be the best.
- Although we have considered a sample of real APIs for our experimental validation, we cannot assume that our results hold for any other arbitrary REST APIs.

5 INDUSTRIAL USAGE

Tool. We have developed the RAFT tool that incorporates the algorithms described in this paper, along with additional functionalities. The tool includes a user interface (developed using HTML/-JavaScript) that interacts with the backend Python APIs. The user flow within the tool is illustrated in Figure 5. IBM has internally utilized this proprietary tool for over a year.

The tool analyzes the uploaded specification to identify specifications issues and enables customization of the specification model, auto-generated dependencies, sequences, parameter scenarios, and data scenarios. It also allows users to define new checks in addition to the default response code check and schema conformity checks. Users can create test configurations that selectively generate test cases, rerun existing test suites, and review test execution results and coverage details with various coverage criteria. While the detailed functionality of the tool is beyond the scope of this paper, interested readers can view a demo at https://youtu.be/ElQO3eUbd0c. It's worth noting that many UI features of the tool have been suggested by real-life practitioners.

Industry Usage and Process. The IBM Consulting team, hereafter referred to as the QE (Quality Engineering) team, currently consists of three testing practitioners who regularly employ this tool to test both internal and client APIs. To date, they have utilized the tool for eight clients spanning the Finance, Telecom, and Airline industries, testing a total of 25 client APIs and approximately 25 internal APIs.[4]

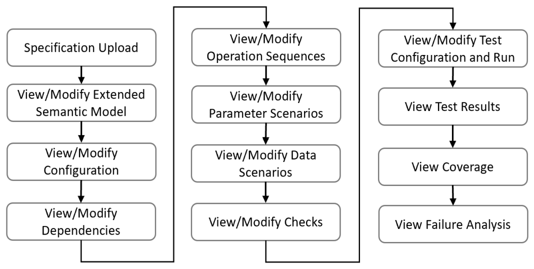
Specification Upload
View/Modify Extended Semantic Model
View/Modify Configuration
View/Modify Dependencies
View/Modify Operation Sequences
View/Modify Parameter Scenarios
View/Modify Data Scenarios
View/Modify Checks
View/Modify Test Configuration and Run
View Test Results
View Coverage
View Failure Analysis

Figure 5: User flow

We outline the typical process followed by the QE team for client API testing. This description will provide readers with insights into the typical scenarios for using the tool.

- The QE team conducted a prior study on the cost savings achieved with the RAFT in an IBM internal application with 150 operations. Manual test case generation for that API took four person-months; however, using this tool, they were able to generate test suites in just one week. This case study forms part of the value proposition presented to clients.
- The QE team demonstrates the tool's usage along with its installation requirements at the client network, along with the value proposition demonstrated in the aforementioned case study.
- Once approved, the team arranges a meeting with the client's API developers/subject matter experts (SMEs) to gain an understanding of the application, resources, typical functional scenarios, and any additional constraints.
- Next, the API specification is shared by the client. This is used by the QE team in the offline mode to generate producer-consumer dependencies, sequences, parameter scenarios, data scenarios, and test cases. Insight from the customer aids in customizing any producer-consumer dependencies and the addition of constraints.
- The test cases are presented to the clients, demonstrating how to use the tool to generate the clients. The API test/development team may provide feedback on the testcases, which is then relayed back to the authors.
- The RAFT is then installed on the client network and used by the client team in online mode to recreate and execute the test cases. In some cases, statistics of the results are shared with the authors, under confidentiality terms.
- The entire testing process typically takes one week.

Observation, Feedback, and Learnings: Below, we present some of our learnings and design decisions based on industrial usage, without divulging any confidential information.

- Specification: Clients use various specification versions (2.x.x and 3.x.x) and languages (OpenAPI, RAML), necessitating adaptation to these variations. As part of our automation efforts, the QE team has developed an automated RAML to OpenAPI converter. Additionally, RAFT now supports testing multiple APIs dependent on each other, allowing for the upload of a zip file containing OpenAPI specifications, which are then parsed to create a unified specification model. The tool's static analysis of specifications has proven valuable in aiding API developers to enhance their specifications.
- Dependencies: Industrial OpenAPI specifications are often imperfect. We have observed scenarios where no response type is specified, the response type is specified as 'object' without reference to any schema, and producer-consumer relationships are defined across resources without indicating the referred resource. These instances can lead to inaccurate producer-consumer relationships, thus necessitating initial discussions with SMEs to

[4]Please note that we cannot disclose any results from this usage due to confidentiality agreements.

rectify such relationships. The tool is equipped with the capability for such customization.
- Sequences: SMEs endorsed the creation of sequences based on a per-resource basis. Although our benchmarks did not initially highlight the utility of LLM-based operation ordering, it has proven beneficial in at least two known cases in practice.
- Data: The tool's ability to specify additional constraints, such as uniqueness, regex, and custom data, has been essential.
- Checks: SMEs expressed interest in utilizing the tool for custom response-time checks and concrete value checks within the response. As a result, RAFT supports the addition of custom checks involving equality, set-based, and range-based conditionals.
- Offline & Online modes: Having an offline mode was crucial for the QE team to review test cases before presenting them to clients and providing feedback to the authors. This implies that feedback-based techniques [26] may be less applicable in this type of engagement.
- Test export and pipeline: Some clients already have automated DevOps pipelines for executing manually written test cases. Our tool supports exporting test cases in RestAssured-based Java programs and BDD-Cucumber scripts, enabling clients to incorporate auto-test generation into their DevOps pipeline.

6 RELATED WORK

Rest API Testing techniques can be broadly classified into black-box and white-box. As our technique is black-box, below we focus on black-box API testing papers. We mainly discuss sequence generation strategies.

Atlidakis et al. [15, 16] created RESTler, which creates operation sequences maintaining the producer-consumer relationship and exploring the operations in a breadth-first fashion. Their sequences do not maintain any resource life-cycle relationship. As identified by MOREST [29], their producer-relationship contains many false positives. In [23], the authors extended RESTler to include intelligent data fuzzing.

Arcuri et al. [7–14, 33] developed EvoMaster which includes novel techniques for test case generation for API with white-box access using a genetic algorithm. They also have a black-box version of the algorithm [12] which essentially performs random testing adding heuristics to maximize HTTP response code coverage.

RestTestGen [20, 46] infers parameter dependencies with better field name matching than just plain field-name equality which uses field names and object names inferred from operations and schema names. They maintain the CRUD order and create a single sequence to test all the operations for *all* resources in the API. It uses a response dictionary and uses examples and constraints in API spec and random values if such information is not available.

bBOXRT [28] performs robustness testing of APIs. Even though this is a black-box tester, it performs negative testing and therefore is not comparable to our approach.

Giamattei et al. [39] proposed a black-box testing approach called uTest which uses a combinatorial testing strategy [35]. The input partitioning for all parameters is formed based on [17] and divided into valid and invalid classes. It uses examples and default values from spec and uses bounds to generate random values. They perform test case formation for each operation and no details about the producer-consumer relationship or sequence formation are mentioned. They used the coverage metrics defined in [31] for evaluation.

RESTest [32] uses fuzzing, adaptive random testing, and constrained-based testing where constraints are specified manually in an OpenApi extension language, especially containing the inter-parameter dependency. Their sequence generation is confined to each operation with the required pre-requisite.

Corradini et al. [19] implemented the black-box test coverage metrics defined in [31] in their tool called Restats and performs operation-wise test case generation.

Tsai et al. [44] uses test coverage level [31] as feedback to and guides the black-box fuzzers in their tool called HsuanFuzz. They do not produce resource-based functional test cases but follow the order of test operations based on paths and CRUD order. They also use the concept of ID parameters.

MOREST [29] performs black-box API testing where it first captures the following dependencies - 1) operation o returns an object of schema s, 2) operation o uses a field or whole of the schema s, 3) operation o_1 and o_2 are in the same API 4) two schema s_1, s_2 which has a field in common in the form of resource-dependency property graph. It uses the graph to generate the test cases and rectify the property graph online.

A recent approach, called RestCT [47], uses a sequence generation algorithm that considers the CRUD partial ordering constraints and producer-consumer relationship (same as ours) but differs in the notion of resource-boundary-based generation or language transformer-based total ordering, instead generates all combinations of partial order to create total order.

All the above test case generation strategies are not related to resource-based test case generation. Therefore they don't need the operation grouping and neither they need operation ordering having the same HTTP methods. Our parameter matching algorithm is a rank-based algorithm that considers matching between schema names, field names, operation names, resource names, and path information. Also, most related works perform input-output field matching between all operations. We restrict such matching on the operation level as well as at the parameter level. Also, we have a data availability relationship that is not considered in other algorithms.

7 CONCLUSION

In this paper, we introduced the notion of functional test case generation for testing APIs from practitioners' perspectives and present novel algorithms pertaining to functional sequence generation and accurate parameter dependency inference. Our tool can handle multiple API specifications combining them into a single model and finding relationships that go across specifications. Even though the experimental results with open source do not reveal the utility of LLM-based ordering, our experience with industry APIs suggests otherwise. The experimental results suggest that we need a better data creation algorithm to create valid data which can reduce invalid requests and automatically identify and refine specification inconsistencies.

REFERENCES

[1] Flair Pos tagger. https://huggingface.co/flair/pos-english, 2022. [Online; accessed 19-August-2022].

[2] RestGo repository. https://github.com/codingsoo/REST_Go, 2022. [Online; accessed 19-August-2022].

[3] Spiral Repository. https://github.com/casics/spiral, 2022. [Online; accessed 19-August-2022].

[4] CAR API. CAR API. https://carapi.app/api, 2022. [Online; accessed 2-May-2023].

[5] Api blueprint, 2023. Accessed: Apr 17, 2023.

[6] Apis.guru api directory, 2022. Accessed: Apr 7, 2023.

[7] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In *International symposium on search based software engineering*, pages 3–17. Springer, 2017.

[8] Andrea Arcuri. Restful api automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20. IEEE, 2017.

[9] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397. IEEE, 2018.

[10] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018.

[11] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[12] Andrea Arcuri. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software*, 38(3):72–78, 2020.

[13] Andrea Arcuri and Juan P Galeotti. Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–34, 2021.

[14] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. Evomaster: A search-based system test generation tool. *Journal of Open Source Software*, 6(57):2153, 2021.

[15] Vaggelis Atlidakis. Testing of cloud services.

[16] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[17] Antonia Bertolino, Guglielmo De Angelis, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Devopret: Continuous reliability testing in devops. *Journal of Software: Evolution and Process*, page e2298, 2020.

[18] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Empirical comparison of black-box test case generation tools for restful apis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 226–236. IEEE, 2021.

[19] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Restats: A test coverage tool for restful apis. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 594–598. IEEE, 2021.

[20] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Automated black-box testing of nominal and error scenarios in restful apis. *Software Testing, Verification and Reliability*, page e1808, 2022.

[21] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi 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. IEEE, 2018.

[22] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[23] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.

[24] Gestao Hospital. Gestao Swagger Specification. https://github.com/ValchanOficial/GestaoHospital, 2022. [Online; accessed 19-August-2022].

[25] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141. IEEE, 2020.

[26] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. Adaptive rest api testing with reinforcement learning. *arXiv preprint arXiv:2309.04583*, 2023.

[27] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 289–301, 2022.

[28] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. A black box tool for robustness testing of rest services. *IEEE Access*, 9:24738–24754, 2021.

[29] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. Morest: Model-based restful api testing with execution feedback. *arXiv preprint arXiv:2204.12148*, 2022.

[30] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. Specification and automated analysis of inter-parameter dependencies in web apis. *IEEE Transactions on Services Computing*, 2021.

[31] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Test coverage criteria for restful web apis. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 15–21, 2019.

[32] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 682–685, 2021.

[33] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[34] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[35] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.

[36] OpenAPI. OpenAPI Specification, 2023. accessed Apr 17, 2023.

[37] Petstore. PetStore Swagger Specification. https://petstore.swagger.io/, 2022. [Online; accessed 19-August-2022].

[38] Restful api modeling language, 2023. Accessed: Apr 17, 2023.

[39] Stefano Russo. Assessing black-box test case generation techniques for microservices. In *Quality of Information and Communications Technology: 15th International Conference, QUATIC 2022, Talavera de la Rina, Spain, September 12-14, 2022, Proceedings*, page 46. Springer Nature.

[40] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2017.

[41] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 117–128. IEEE, 2021.

[42] 2022 State of the API Report, 2022.

[43] Language Tool. Language Tool. https://github.com/languagetool-org/languagetool, 2022. [Online; accessed 19-August-2022].

[44] Chung-Hsuan Tsai, Shi-Chun Tsai, and Shih-Kun Huang. Rest api fuzzing by coverage level guided blackbox testing. *arXiv preprint arXiv:2112.15485*, 2021.

[45] Emanuele Viglianisi, Michael Dallago, and Mariano 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. IEEE, 2020.

[46] Emanuele Viglianisi, Michael Dallago, and Mariano 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. IEEE, 2020.

[47] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of restful apis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.

[48] XKCD. XKCD Swagger Specification. https://raw.githubusercontent.com/APIs-guru/openapi-directory/main/APIs/xkcd.com/1.0.0/openapi.yaml, 2022. [Online; accessed 19-August-2022].

[49] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. Resource and dependency based test case generation for restful web services. *Empirical Software Engineering*, 26(4):1–61, 2021.