Enhancing Resource-based Test Case Generation For RESTful APIs with SQL Handling [*]

Man Zhang[1][0000−0003−1204−9322] and Andrea Arcuri[1,2][0000−0003−0799−2930]

[1] Kristiania University College, Norway
[2] Oslo Metropolitan University, Norway
{man.zhang, andrea.arcuri}@kristiania.no

Abstract. Nowadays, many companies use RESTful web services to develop their enterprise applications. These web services typically interact with databases. In REST, resource handling is a fundamental concept, where resources are manipulated by exposing HTTP endpoints. Rd-MIO* is an evolutionary algorithm which is specialized in test generation for such kind of services, i.e., RESTful APIs, via manipulating *resource*s in various ways using HTTP actions (e.g., `GET` and `POST`). In this paper, we further extended Rd-MIO* by employing SQL commands to manipulate the resources for test generation, directly into the databases. We implemented our novel technique as an extension of the EvoMaster tool. To evaluate our approach, we selected Rd-MIO* as a baseline technique and conducted an empirical study with five open source REST APIs. Results showed that our approach clearly outperforms the baseline over all of the five case studies.

Keywords: White-box test generation · SQL · REST API Testing · SBST.

1 Introduction

REST is widely applied in developing web enterprise systems for providing services over the network, e.g., Google Drive[3] and Azure[4]. This kind of web services typically need communications over the network (e.g., with clients and external services), and interact with databases. Due to these interactions, it is challenging to test these systems, especially for system-level test case generation.

In REST, there exists a set of endpoints (e.g., POST and GET), which are exposed for providing services over HTTP. Dealing with *resources* is a fundamental concept, where the exposed endpoints enable manipulating these resources. Rd-MIO* [14,15] is a search-based testing approach which is developed by handling resources for white-box test generations particularly for RESTful APIs. The approach defines a set of templates to structure HTTP calls in a test in

---

[3] https://developers.google.com/drive/api/v3/reference
[4] https://docs.microsoft.com/en-us/rest/api/azure/

terms of resources, and developed a set of novel strategies to sample and mutate the tests with such templates.

In this paper, we extended Rd-MIO* by employing Structured Query Language (SQL) to enhance resource handling, i.e., enable adding resources to be performed into database directly, and utilizing existing resources for the actions to be tested. We integrated our approach into EvoMaster [2] open-source tool, and conducted an empirical study with five open-source case studies (one artificial and four real-world). Results show that tests generated by our novel approach are capable of achieving on average 45.0% (up to 65.5%) line coverage and 20.5% (up to 27.3%) branch coverage, among the five case studies. Compared to the existing Rd-MIO* using the default setting, our novel approach demonstrates consistent and clear improvements on all of the five case studies. Relative improvements are up to 26.0% for target coverage, up to 26.2% for line coverage, 20.3% for branch coverage, and 40.6% for fault detection.

The rest of the paper is organized as: Background and Related work are described in Section 2. The proposed approach is presented in Section 3, followed by an empirical study on it (Section 4). We discuss the threats to validity in Section 5, and conclude the paper in Section 6.

2   Background and Related Work

2.1   Resource and dependency based MIO (Rd-MIO*)

The Many Independent Objective (MIO) [3] algorithm is an evolutionary algorithm inspired by the (1+1) Evolutionary Algorithm which only contains sampling and mutation. MIO is designed for generating system-level white-box tests, and Rd-MIO* [14,15] is an extension of it by handling test generation with an explicit consideration of resources and their dependencies in RESTful APIs. In Rd-MIO*, based on HTTP semantics, ten templates of structuring actions on manipulating resources in a test were developed [15], as follows:

T1. GET is to retrieve resource(s);
T2. PATCH is to partially update a resource which is likely nonexistent;
T3. DELETE it to delete a resource which is likely nonexistent;
T4. PUT is to replace a resource which is likely nonexistent;
T5. POST is to create a resource;
T6. POST/PUT-POST is to create a resource which likely exists;
T7. POST/PUT-GET is to retrieve a resource which likely exists;
T8. POST/PUT-PUT is to replace a resource which likely exists;
T9. POST/PUT-PATCH is to partially update a resource which likely exists;
T10. POST/PUT-DELETE is to delete a resource which likely exists.

Thus, a sequence of actions following the template can be regarded as a *resource-handling* with a specific purpose, and a test can be regarded as a sequence of such handlings. In addition, each of such templates has a property indicating that it is either *independent* (i.e., T1 – T4) or *possibly-dependent* (i.e., T5 – T10). With

such definitions, an individual is defined as a sequence of *resource-handling*s which perform a sequence of actions (e.g., HTTP calls) on the resources. Moreover, Zhang *et al.* [15] defined *resource-based sampling* and *resource-based mutation* for producing and evolving the individual with such structure, i.e., *resource-handling*s. To investigate dependencies in REST APIs, Rd-MIO* is integrated with *resource dependency heuristic handling*, which is capable of identifying possible dependencies during the search [15]. Then the sampling and mutation in Rd-MIO* can further employ such identified dependencies to produce new individuals. In this paper, we extended the individual and *resource-handling* with direct SQL commands for enhancing such resource-based handling.

2.2   SQL handling in EvoMaster

SQL is a widely used language for managing data in databases. To track all interactions with the database, in EvoMaster, SQL commands monitoring is implemented which is capable of tracing all executed SQL commands for accessing the SQL database of the SUT during the search [6]. Thus, when executing actions to be performed on a resource, we can know what tables are accessed with SQL. In Rd-MIO*, dependencies between resources and tables are also collected for identifying possible dependencies among the resources. For instance, if the same table is accessed by manipulations on resource A and resource B, then there might exist a dependency between A and B. For REST APIs, if a HTTP action triggers an access to tables, it is likely that the table is related to the resources to be manipulated. In addition, EvoMaster is integrated with a Domain Specific Language (DSL) (developed by [6]) which enables direct data insertions with SQL from the generated JUnit tests. With such existing support, we can employ SQL for manipulating the resources throughout the search.

2.3   REST API Testing

With a wide application of REST, there exists an increase research effort in test methods for REST APIs. To test RESTful API services, many methods [7–10,12, 13] have been developed with OpenAPI, which is a machine-readable schema that describes how to create requests for the services. The existing EvoMaster we extended in this work also uses such schemas to produce tests. In [7], Atlidakis *et al.* developed RESTler for generating a sequence of requests to test REST APIs. The sequence is decided by an inference on dependencies among the endpoints based on the OpenAPI (at the beginning) and an analysis on runtime responses. Godefroid *et al.* [9] also employs the OpenAPI schema to generate test data for REST APIs using fuzzing techniques. In [10], the schema is applied for studying differences among different versions of RESTful API services with differential testing technique. However, most of the approaches on REST APIs are in the context of black-box testing [8, 13].

   To our best knowledge, the only approaches for handling white-box test generation for REST APIs are from our work on EvoMaster [3, 4, 6, 15]. In this paper, we extended the approach which is for handling test generation with

resource-based methods (as described in Section 2.1) on EvoMaster, and further selected the approach as our baseline in the empirical study.

3   Resource handling with SQL

Resource-based technique (i.e., Rd-MIO*) has demonstrated its effectiveness in white-box test generation for RESTful APIs [15]. In Rd-MIO*, the *resource-handling* is based on the templates which only rely on HTTP actions. However, it might not be always feasible to apply HTTP calls on manipulating resources, e.g., the dependent resources might require different levels of authorizations, or the creation of the resource is not clearly defined in the schema. In these cases, the state of the resources can not be changed with the Rd-MIO* templates during search, and that could limit the effectiveness of resource-based techniques for maximizing code coverage and faults finding in the context of white-box testing. To manipulate such states, instead of using HTTP actions, it is also applicable to directly modify data in the database, if any is used. In addition, this is typically true in RESTful web services which interact with databases for persisting *resource*s.

Considering an example, where a REST API which interacts with a database has two resources, *foo* (with POST and GET) and *bar* (only with GET). *foo* is required to refer to an existing *bar*, but there does not exist a clear creation action for *bar* in the schema as shown in Figure 1. In this case, without an existing *bar*, this issue limits the achievable line coverage on all of the endpoints. However, based on the *SQL commands monitoring* (see Section 2.2), the accessed

```
1  "/foo/{id}": {
2    "get": {..},
3    "post": {
4      "parameters": [
5      {
6        "name": "id",
7        "in": "path",
8        "required": true,
9        "type": "integer",
10       "format": "int32"
11     },{
12       "name": "barId",
13       "in": "query",
14       "required": true,
15       "type": "string"
16     }],
17     "responses": {
18       "200": {..},"201": {..},"401": {..},"403": {..},"404": {..}
19     },
20     "deprecated": false
21   }
22 },
23 "/bar/{id}": {
24   "get": {..}
25 }
```

Fig. 1: Snippet code of a schema with OpenAPI

Table 1: Resource-based sampling templates with SQL commands

# Template	Description	Independent?
11 *SQL*-POST	To create an existing resource	No
12 *SQL*-GET	To retrieve an existing resource	No
13 *SQL*-PUT	To replace an existing resource	No
14 *SQL*-PATCH(-PATCH)	To (partially) update an existing resource	No
15 *SQL*-DELETE	To delete an existing resource	No

Note that *SQL* refers to `INSERT` and `SELECT` commands, and the template is only applicable to the resources which has identified possibly-related tables.

table can be known when executing GET on *bar*. Thus, we could possibly add a resource for the *bar* by using `INSERT` on the accessed tables.

To enable the application of SQL in *resource-handling*, we firstly extend the templates by involving SQL to manipulate resources. Based on semantics of HTTP actions, we further develop five templates with SQL (as shown in Table 1). The templates share similar testing purposes on the endpoints of the SUT with T6 – T10 templates in Rd-MIO* (see Section 2.1). However, we extend them for resource preparation by using SQL (i.e., `SELECT` and `INSERT`) commands. As the resource handled by the proposed templates is possible to have an impact to following actions in the test, then we identify them as *possibly-dependent* templates, i.e., the *independent* property is `False` in Table 1. With SQL, `SELECT` can be applied to the situation whereby there exist resources in the SUT (e.g., seeded data), then endpoints can be tested with such existing resources, i.e., link the endpoints with existing ones. For `INSERT`, it is to create new resources directly into the database, then further employ the newly created ones to test the endpoints. In addition, we also provide a further extension for T6 – T10 templates with SQL, e.g., an extension would be *SQL*-POST/PUT-GET for T7 that is applicable when the POST/PUT cannot function properly to create required resources. For instance, to test a retrieve operation of an existing *foo*, it requires a preparation of the *foo* resource. But POST *foo* could not be created due to lack of dependent resources, i.e., *bar*. In this case, we can either employ *SQL*-GET instead of POST-GET or create the *bar* for the POST. Such dependency could be identified with dependency handling in Rd-MIO* [15]. Here, we can employ
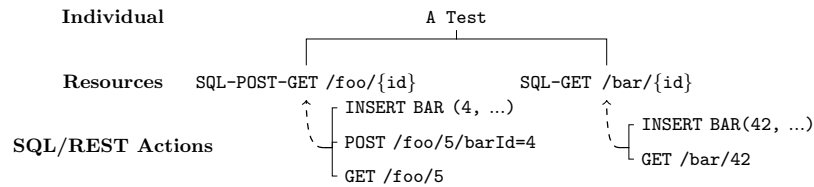

Fig. 2: An example to illustrate a representation of resource-based individual with SQL handling

such information for resource creation with SQL. Figure 2 illustrates a test with a representation of resource-based individual employing the proposed templates. The test comprises two resource handlings: one is to retrieve an existing *foo* with an extended POST-GET template, and the other is to retrieve an existing *bar* with *SQL*-GET template.

To ensure that SQL actions and REST actions perform on the same resource, we need to further handle value binding among the actions during sampling. The binding is implemented based on name matching using the Trigram Algorithm [11] (which has been applied in [15]) for calculating a degree of similarity between a column name of a table and a gene name in REST actions. Note that for a SQL action, its genes are typically flatten, while for a REST action, its genes might be structured (e.g., when representing JSON body payloads). In this case, we need to go through every genes in the REST action in order to find the most matched one (but the similarity degree must be more than 0.6 [15]), then bind the SQL gene and the REST gene. The SQL gene and the REST gene might have different types, e.g., a *id* of a resource might be `Long` in SQL but `String` in the REST action. In this case, we handle a type conversion for genes to be bound with different types. The binding direction depends on the type of the SQL, i.e., we bind the rest gene based on when SQL is `SELECT`, and bind the SQL gene based on the REST gene when SQL is `INSERT`. For a resource handling, values on the binding genes might be modified, then we need to further synchronize such binding genes after the mutation. Notice, genes for a `SELECT` and REST genes binding with `SELECT` are not mutable. Based on such binding, SQL actions and REST actions could be restrained for performing on the same resource, as shown in Figure 2, e.g., `barId` is bound with `id` of the INSERT on the table `BAR`, and so when one is mutated, then the value of the other is automatically updated.

4   Empirical Study

To assess our novel proposed approach, we have carried out an empirical study which aims at answering two research questions:

RQ1: How does resource-based MIO with SQL perform? Among the different settings, which one performs best?
RQ2: How much improvement does our approach achieve compared to the existing work in terms of code coverage and fault detection?

4.1   Experiment Setup

In these experiments, we employed five REST APIs from an existing benchmark[5] that were previously used in conducting empirical studies on RESTful API testing approaches [4–6, 14]. All of the APIs are open-source Java/Kotlin projects that interact with a database. Table 2 reports descriptive statistics of

---
[5] https://github.com/EMResearch/EMB

Table 2: Descriptive statistics of the case studies

Name	#Classes	LOCs	#Endpoints	Resource #R (#Indep.)	Database (#Tables, #Columns)
rest-news	10	718	7	4 (1)	(1, 5)
catwatch	69	5442	13	13 (11)	(5, 45)
features-service	23	2347	18	11 (1)	(6, 20)
proxyprint	68	7534	74	56 (26)	(15, 92)
scout-api	75	7479	49	21 (2)	(14, 70)
Total	245	23520	161	105 (41)	(41, 232)

#R: a number of resources; #Indep: a number of independent resources out of #R

the case studies with the number of classes (#Classes), lines of codes (LOCs), number of endpoints (#Endpoints), number of resources (#R), number of independent resources (#Indep) out of #R, number of tables (#Tables) and number of columns (#Columns). Regarding the case studies, *rest-news* is an artificial API which was applied in a university course of enterprise development, and the remaining four (i.e., *catwatch, features-service, proxyprint, scout-api*) are real open-source projects searched from GitHub (https://github.com/).

We implemented our approach (denoted as Rd-MIO*$_{sql}$) by extending Rd-MIO* in EvoMaster with SQL handling on the resources. To assess its performance, we firstly studied the probability of employing SQL to handle resources with three settings, i.e., $P_{sql} \in \{0.1, 0.3, 0.5\}$. Note that, in the context of REST API testing, the endpoints should have a higher priority to be involved in a test. Therefore, we set the maximum value of the setting as 0.5, i.e., SQL and POST/PUT have an (at most) equal probability to be involved in a resource-handling for preparing resources. Then we selected Rd-MIO* with its best configuration [15] as the baseline technique to compare with. The performance of the techniques are compared with three coverage metrics: the number of covered targets (#Targets), line coverage (%Lines) and branch coverage (%Branches). #Targets is a coverage criterion used in EvoMaster for test generation which comprises status code coverage, code coverage and fault finding (i.e., the aggregation of all other coverage criteria). More details about the target coverage can be found in [3]. %Lines and %Branches are metrics typically used in evaluating white-box testing techniques. To compare with the baseline, we also employ the number of potential faults (#Faults) as a metric for fault detection. In these experiments, considering the stochastic nature of the search algorithm, we repeated our experiments 30 times, following common guidelines on conducting SBSE experiments [1]. All of the techniques were executed with the same search budget (i.e., $100k$ HTTP calls), on the same machine.

4.2 Experiment Results

Results for RQ1 In Table 3, we report the average #Targets, %Lines and %Branches achieved by our approach combined with all settings (i.e., $P_{sql} =$

Table 3: Average #Targets, %Lines, %Branches obtained by Rd-MIO*$_{sql}$.

SUT	#Targets	%Lines	%Branches
rest-news	341.44	52.9%	**27.3%**
catwatch	1238.01	33.1%	17.8%
features-service	723.02	**65.5%**	21.2%
proxyprint	2522.48	32.7%	14.5%
scout-api	1967.17	40.8%	21.9%
Average		45.0%	20.5%

$\{0.1, 0.3, 0.5\}$) for all of the case studies. Results show that our approach enables covering on average 45.0% (up to 65.5%) of lines and 20.5%(up to 27.3%) of branches among the five case studies.

Table 4 represents further results for each of the settings with a rank among the settings. The setting with the maximum value (i.e., $P_{sql} = 0.5$) achieves the best results on *rest-news*, *proxyprint*, *scout-api*. This might indicate that, in these case studies, there might exist some difficulties by using endpoints (i.e., POST or PUT) to create resources. Thus, a relatively higher probability (such as 0.5) of applying SQL has a high chance to obtain better results. Compared with *features-service* and *catwatch*, the preference on applying SQL is relatively lower, i.e., 0.1 for *features-service* and 0.3 for *catwatch*. In total, based on average rank among the case studies, we selected 0.5 (i.e., 50%) as the default configuration for P_{sql}.

> RQ1: Among the five REST APIs, our approach is capable of automatically
> generating tests that cover 45.0% of lines (up to 65.5%) and 20.5% of
> branches (up to 27.3%) on average. We recommend to apply SQL for
> handling resource with a 50% probability.

Table 4: Average #Targets, %Lines, %Branches by different configurations of $P_{sql} \in \{0.1, 0.3, 0.5\}$. We also report ranks of the configurations for each SUT (value **1** indicates the best), and p-value and χ^2 of the Friedman test based on the ranks.

	#Targets			%Lines			%Branches		
SUT	0.1	0.3	0.5	0.1	0.3	0.5	0.1	0.3	0.5
rest-news	337.29(3)	342.08(2)	**345.20(1)**	52.6%(3)	53.0%(2)	**53.3%(1)**	26.7%(3)	27.4%(2)	**27.9%(1)**
catwatch	1240.03(2)	**1241.00(1)**	1233.15(3)	33.1%(2)	**33.2%(1)**	33.0%(3)	17.8%(2)	**17.9%(1)**	17.6%(3)
features-service	**724.34(1)**	723.65(2)	721.09(3)	**65.7%(1)**	65.6%(2)	65.4%(3)	**21.4%(1)**	21.1%(2)	21.0%(3)
proxyprint	2520.97(2)	2509.59(3)	**2536.91(1)**	32.7%(2)	32.6%(3)	**32.9%(1)**	14.4%(2)	14.3%(3)	**14.7%(1)**
scout-api	1955.09(3)	1966.76(2)	**1979.67(1)**	40.6%(3)	40.8%(2)	**41.1%(1)**	21.6%(3)	22.0%(2)	**22.2%(1)**
Average Rank	2.20	2.00	**1.80**	2.20	2.00	**1.80**	2.20	2.00	**1.80**
Friedman Test		(0.400,	0.819)		(0.400,	0.819)		(0.400,	0.819)

Results for RQ2 To compare with the baseline technique, Table 5 reports the average of #Targets, %Lines, %Branches and #Faults, and their results of pair comparison analysis by Mann-Whitney-Wilcoxon U-tests (p-value) and Vargha-Delaney effect sizes ($\hat{A}_{xy}$). For coverage metrics, based on the average results, our approach performs consistently better than the baseline for all of the metrics. Regarding the pair comparison results, in four out of the five case studies (i.e., *rest-news*, *catwatch features-service* and *proxyprint*), our approach achieves clearly significant improvement which can be demonstrated by the low p-value (i.e., < 0.01) and the high effect size (i.e., > 0.86). For *scout-api*, there exists modest improvement, and the improvements on #Targets and %Lines are statistically significant (i.e., p-value < 0.05 and $\hat{A}_{xy} > 0.5$).

To assess the fault detection by our approach, we also report the number of "potential" faults (#Faults) detected by our proposed approach and the baseline in Table 5. Note that faults can be detected with the HTTP status code (i.e., 500 in RESTful APIs). Regarding the #Faults metric, our approach achieves significant improvements over all of the case studies.

In Figure 3, we also analyze the average number of covered targets (i.e., a metric combined several coverage metrics) using line plots over time, i.e., at every 5% of the used budget, during the search. Based on these results, for all of the case studies, our approach shows a clear margin throughout the search

Table 5: Results by comparing with the baseline technique

SUT	Metrics	A	B	$\hat{A}_{ba}$	A(Base) p-value	B(Rd-MIO*$_{sql}$) relative$_{(b-a)/a}$
rest-news	#Targets	273.91	**345.20**	**1.00**	**<0.01**	**+26.0%**
	%Lines	42.2%	**53.3%**	**1.00**	**<0.01**	**+26.2%**
	%Branches	23.2%	**27.9%**	**1.00**	**<0.01**	**+20.3%**
	#Faults	4.72	**6.63**	**1.00**	**<0.01**	**+40.6%**
catwatch	#Targets	1055.03	**1233.15**	**1.00**	**<0.01**	**+16.9%**
	%Lines	27.4%	**33.0%**	**1.00**	**<0.01**	**+20.5%**
	%Branches	14.8%	**17.6%**	**1.00**	**<0.01**	**+19.1%**
	#Faults	16.70	**20.76**	**0.97**	**<0.01**	**+24.3%**
features-service	#Targets	707.52	**721.09**	**0.94**	**<0.01**	**+1.9%**
	%Lines	64.3%	**65.4%**	**0.86**	**<0.01**	**+1.7%**
	%Branches	18.5%	**21.0%**	**0.95**	**<0.01**	**+14.0%**
	#Faults	37.55	**39.57**	**0.92**	**<0.01**	**+5.4%**
proxyprint	#Targets	2342.90	**2536.91**	**0.89**	**<0.01**	**+8.3%**
	%Lines	30.7%	**32.9%**	**0.86**	**<0.01**	**+7.3%**
	%Branches	13.5%	**14.7%**	**0.92**	**<0.01**	**+9.1%**
	#Faults	90.19	**100.74**	**0.99**	**<0.01**	**+11.7%**
scout-api	#Targets	1944.03	**1979.67**	**0.69**	**0.003**	**+1.8%**
	%Lines	40.4%	**41.1%**	**0.68**	**0.004**	**+1.7%**
	%Branches	21.9%	**22.2%**	0.61	0.069	+1.4%
	#Faults	111.47	**113.42**	**0.71**	**<0.01**	**+1.8%**

compared with the baseline. This demonstrates the advantage of our approach on both exploration and exploitation phases with SBST on RESTful APIs.

Thus, we can conclude that

> *RQ2: Rd-MIO* enhanced with our SQL handling (with a 50% probability of its application) consistently outperforms the baseline technique in all of the five REST APIs in terms of target coverage, line coverage, branch coverage and fault detection.*

Discussion Regarding the coverage improvement on the case studies, we found that our approach is the most effective to *rest-news*. By checking this case study, we found that there exist some difficulties when creating *news* resource with POST endpoint, because a *news* need to refer to a valid country specified with `String`, i.e., one of pre-defined list restored in a textual file (i.e., `.txt`). With the search, it is difficult to get a valid `String` within the limited budget, especially when there exist many objectives to be optimized in our context. Since the *news* resource cannot be created, it would have an impact on testing related actions on this resource, e.g., GET, UPDATE. For instance, Figure 4 shows a snippet code of the GET endpoint (see `NewsRestApi.kt`[5]) on this resource (i.e., `GET /news/{id}`) with coverage information achieved by tests generated by our approach (i.e., all lines are covered). Note that lines with green color indicate the line covered by tests. Regarding the line coverage on this endpoint, we found
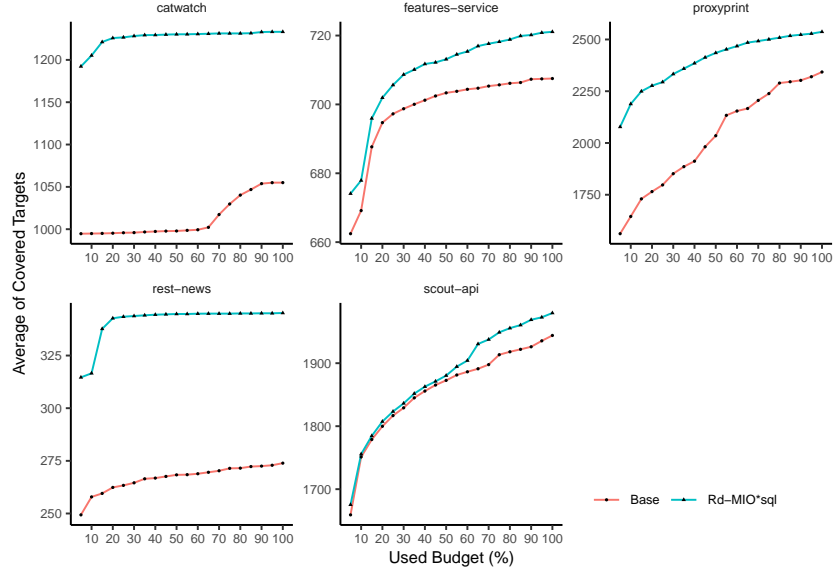

Fig. 3: Average covered targets (*y-axis*) with Base and Rd-MIO*$_{sql}$ throughout the search, reported at 5% intervals of the used budget allocated for the search (*x-axis*).

```
141      @ApiOperation("Get a single news specified by id")
142      @GetMapping(path = arrayOf("/{id}"))
143      fun getNews(@ApiParam(ID_PARAM)
144                  @PathVariable("id")
145                  pathId: String?)
146             : ResponseEntity<NewsDto> {
147
148          val id: Long
149          try {
150              id = pathId!!.toLong()
151          } catch (e: Exception) {
152              /*
153                  invalid id. But here we return 404 instead of 400,
154                  as in the API we defined the id as string instead of long
155              */
156              return ResponseEntity.status(404).build()
157          }
158
159          val dto = crud.findById(id).orElse(null) ?: return ResponseEntity.status(404).build()
160
161 X        return ResponseEntity.ok(NewsConverter.transform(dto))
162      }
```

Fig. 4: An example of coverage of `GET /news/{id}` endpoint in *rest-news* achieved by our technique

```
1  @Test
2  public void test_5() throws Exception {
3      List<InsertionDto> insertions0 = sql().insertInto("NEWS_ENTITY", 275L)
4               .d("ID", "862")
5               .d("AUTHOR_ID", "\"xxuxEf_0uLPFbIX\"")
6               .d("COUNTRY", "\"8ih8sWF\"")
7               .d("CREATION_TIME", "\"2042-10-16 10:29:25\"")
8               .d("TEXT", "\"B1zjmtfSzgzi\"")
9          .dtos();
10     controller.execInsertionsIntoDatabase(insertions0);
11     given().accept("application/vnd.tsdes.news+json;charset=UTF-8;version=2"
       )
12             .get(baseUrlOfSut + "/news/862")
13             .then()
14             .statusCode(200)
15             .assertThat()
16             .contentType("application/vnd.tsdes.news+json")
17             .body("'newsId'", containsString("862"))
18             .body("'authorId'", containsString("xxuxEf_0uLPFbIX"))
19             .body("'text'", containsString("B1zjmtfSzgzi"))
20             .body("'country'", containsString("8ih8sWF"))
21             .body("'creationTime'", containsString("2042-10-16T"));
22 }
```

Fig. 5: A test automatically generated by our approach which is able to cover the lines 159 and 161 in Figure 4

that the line 161 is not solved by tests generated by the baseline (denoted with × in the figure) with $100k$ search budget, and a precondition to reach the line is related to an existing *news* resource with the specified `id`. However, with the proposed approach, the problem can be easily addressed by directly inserting a *news* resource, despite that the country might not be valid (as the check is carried out only when a new country entry is created). A test to handle the problem is shown in Figure 5.

Regarding *catwatch* case study, it also obtains a noticeable improvement with our approach. As its statistics shown in Table 2, 11 out of 13 endpoints are *independent*, i.e., GET actions. The remaining two are endpoints such as `/import`

and `/export` that do not refer to any specific resources. Thus, it is unlikely to manipulate the resources in this REST API (e.g., *language*, *contributor*) using HTTP actions with the default Rd-MIO*. An alternative solution such as SQL handling would show its effectiveness to this situation. For instance, regarding `LanguageService.java` related to *language* resource, the line coverage by Rd-MIO* and Rd-MIO*$_{sql}$ are 56.2% *vs.* 96.9%. Due to space limitation, complete coverage report can be found with a link[6] that are conducted by Intellij coverage report[7].

Regarding *features-service*, most of the resources have a reference to a POST action for their creation. In addition, its schema clearly shows their hierarchical relationships among the resources, which further makes required resources complete (i.e., prepare corresponding ancestor's resources) with a high probability. This could explain the modest improvement and the effectiveness with a relatively low ($P_{sql} = 0.1$) application probability (as shown in Table 4) on this case study. However, our approach still demonstrates its effectiveness compared with the baseline, i.e., the significant improvement shown for all of the metrics. This indicates that SQL handling is possibly required for resource handling for testing REST API despite that POST/PUT endpoints for the resources have been clearly defined.

Regarding *proxyprint*, there exist various resources (i.e., 56 #R in Table 2) and their relationships are not clearly identified with the schema. For instance, `POST /request/accept/{id}` is to accept a request to add a new printshop, and the request can be registered via `POST /request/register`. `POST` uses to create a resource, and its dependent resources if exist are typically specified with a hierarchical form, e.g., `/products/{productName}/features/{featureName}` and `/products/{productName}` in *features-service*. But in this case, the dependency between the two endpoints are not obvious with their URIs, (i.e., `/request/register` and `/request/accept`) that might limit an effectiveness of HTTP actions on manipulating such resources. However, for `POST /request/accept/{id}`, the accessed table can be identified (as described in Section 2.2) that allow SQL to prepare such resource for the `POST`. The effectiveness of our approaches can be shown with 95.2% line coverage on `RegisterRequestController.java`[5] for handling the request, compared with 35.7% by Rd-MIO*. In addition, in this case study, there exists an endpoint `/admin/seed` for initializing data into the service (see `AdminController.java`[5]). To test the system, we seed such data by requesting the endpoint before each test as shown in Figure 6a. An effective test to `POST /request/accept/{id}` (shown in Figure 6b) is just employed such seeded data, i.e., `id=2` at line 6 refers to a seeded request (see line 119 in `AdminController.java`[5]).

Regarding *scout-api*, its schema is similar with *features-service*, i.e., *resources* are connected and have a POST for its creations. By further identifying the source code, we found that there exist some difficulties in handling *media_file* resources. For instance, `POST /v1/media_files` is to "Add a media file to the

---

[6] https://doi.org/10.5281/zenodo.5059928
[7] https://www.jetbrains.com/help/idea/code-coverage.html

```
1  @Override
2  public void resetStateOfSUT() {
3    DbCleaner.clearDatabase_H2(connection);
4    deleteDir(new File("./target/temp"));
5    try {
6      URL url = new URL("http://localhost:" + getSutPort() + "/admin/seed");
7      HttpURLConnection con = (HttpURLConnection) url.openConnection();
8      con.setRequestMethod("POST");
9      con.setDoOutput(true);
10     con.connect();
11     ...
12   } catch (Exception e) {...}
13 }
```

(a) snippet code for seeding data for each test using an endpoint of SUT

```
1  @Test
2  public void test_74() throws Exception {
3    try{
4      given().accept("*/*")
5      .header("Authorization", "Basic bWFzdGVyOjEyMzQ=") // admin
6      .post(baseUrlOfSut + "/request/accept/2")
7      .then()
8      .assertThat()
9      .contentType("text/plain")
10     .body(containsString("{\"success\":true}"));
11   } catch(Exception e){
12   }
13 }
```

(b) a test generated by our approach which refers to an existing data

Fig. 6: A test automatically generated by our approach which employs the existing data for preparing a resource for `/request/accept/{id}`

system. Specify URL of media file or use 'data URI' to upload base64-encoded file". Currently, it is not effective to generate such data with the search, i.e., a valid URI referring to a media file (e.g., image). But we could employ SQL to add data into the database for making the resource exists. Besides, in this case study, most of data retrieves are implemented with a query parameter named attrs which indicates "The attributes to include in the response. Comma-separated list". However, with $100k$ HTTP calls, currently EvoMaster cannot handle such constraints (e.g., specified with textual language) properly. This might be a reason for the modest improvements on this case study, but they are still statistically significant.

Significant improvements on fault detection could be a result of such improvement on coverage. Thus, by carefully analyzing results on the five case studies, we summarize that:

> *Our proposed technique significantly enhances the handling on resources which is capable of generating more effective tests against REST APIs, particularly effective on the SUTs whose creation actions are restricted or unclear.*

5   Threats to Validity

Conclusion validity. Our applied technique is in the context of search-based testing. To handle its randomness nature, we repeated our experiments 30 times, which is recommended by standard guidelines [1] in search-based software engineering for conducting experiments. To properly draw the conclusions based on the results, we employed a set of statistical tools, i.e., Friedman test (p-value and χ^2) for variance analysis of performances of different settings among case studies, Mann-Whitney U-test (p-value) and Vargha-Delaney ($\hat{A}_{12}$) for reporting comparison results with the baseline technique.

Construct validity. In these experiments, outputs are obtained from search-based techniques. To avoid any potential bias in such outputs, we first used the same stopping criterion (i.e., $100k$ HTTP calls) for baseline and proposed techniques. In addition, all the experiments were executed on the same machine for further dealing with this validity threat.

Internal validity. The approach is implemented on the EVOMASTER tool that is open-source, and the experiments were conducted with case studies which are available online as well. We cannot guarantee that our implementation is free of bugs, but the implementation and experiments can be examined by anyone, as we made them open-source (`www.evomaster.org`).

External validity. The approach was evaluated with five REST APIs which interact with SQL databases, taken from a benchmark for REST API testing. More case studies would help to generalize results of our approach. REST is widely applied in industry, however, most of them are not open-source, which limits our experiments with more case studies.

6   Conclusions and Future Work

Testing REST APIs is challenging, especially for system-level test generation, due to their possible complex interactions with SQL databases. In REST, exposed endpoints are typically defined based on resources and actions that can be performed on them. Thus, via the endpoints, manipulating resources with different states can help obtaining better code coverage in the context of whitebox testing. Rd-MIO* is such approach for automatically generating tests using search-based techniques. In this paper, we further extended it by enhancing the resource manipulation with direct SQL handling. We implemented our approach in the EVOMASTER open-source tool, and conducted an empirical study with five open-source case studies. Experimental results show that our approach significantly outperforms the existing Rd-MIO*, in terms of code coverage and fault detection. Our novel technique demonstrates its advantages on all of the five case studies, particularly on the SUTs whose creation actions are restricted or too complex.

In future, we plan to 1) conduct more studies with various types of databases, and 2) investigate solutions for handling constraints specified with textual language. For more information, visit our webpage at `www.evomaster.org`.

References

1. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) **24**(3), 219–250 (2014)
2. Arcuri, A.: Evomaster: Evolutionary multi-context automated system test generation. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE (2018)
3. Arcuri, A.: Test suite generation with the Many Independent Objective (MIO) algorithm. Information and Software Technology (IST) **104**, 195–206 (2018)
4. Arcuri, A.: Restful api automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM) **28**(1), 3 (2019)
5. Arcuri, A.: Automated blackbox and whitebox testing of restful apis with evomaster. IEEE Software (2020)
6. Arcuri, A., Galeotti, J.P.: Handling sql databases in automated system test generation. ACM Transactions on Software Engineering and Methodology (TOSEM) **29**(4), 1–31 (2020)
7. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: Proceedings of the 41st International Conference on Software Engineering. p. 748–758. ICSE '19, IEEE Press (2019). https://doi.org/10.1109/ICSE.2019.00083, https://doi.org/10.1109/ICSE.2019.00083
8. Ed-douibi, H., Cánovas Izquierdo, J.L., Cabot, J.: Automatic generation of test cases for rest apis: A specification-based approach. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC). pp. 181–190 (2018)
9. Godefroid, P., Huang, B.Y., Polishchuk, M.: 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. p. 725–736. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409719, https://doi.org/10.1145/3368089.3409719
10. Godefroid, P., Lehmann, D., Polishchuk, M.: Differential regression testing for rest apis. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 312–323. ISSTA 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3395363.3397374, https://doi.org/10.1145/3395363.3397374
11. Martin, S., Liermann, J., Ney, H.: Algorithms for bigram and trigram word clustering. Speech communication **24**(1), 19–37 (1998)
12. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In: International Conference on Service-Oriented Computing (2020)
13. Viglianisi, E., Dallago, M., Ceccato, M.: Resttestgen: Automated black-box testing of restful apis. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE (2020)
14. Zhang, M., Marculescu, B., Arcuri, A.: Resource-based test case generation for restful web services. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1426–1434 (2019)
15. Zhang, M., Marculescu, B., Arcuri, A.: Resource and dependency based test case generation for restful web services. Empirical Software Engineering (EMSE) **26**(4), 1–61 (2021). https://doi.org/10.1007/s10664-020-09937-1