Empirical Software Engineering (2021) 26: 76
https://doi.org/10.1007/s10664-020-09937-1

Resource and dependency based test case generation for RESTful Web services

Man Zhang[1] · Bogdan Marculescu[1] · Andrea Arcuri[1,2]

Accepted: 30 December 2020 / Published online: 2 June 2021

Abstract
Nowadays, RESTful web services are widely used for building enterprise applications. REST is not a protocol, but rather it defines a set of guidelines on how to design APIs to access and manipulate resources using HTTP over a network. In this paper, we propose an enhanced search-based method for automated system test generation for RESTful web services, by exploiting domain knowledge on the handling of HTTP resources. The proposed techniques use domain knowledge specific to RESTful web services and a set of effective templates to structure test actions (i.e., ordered sequences of HTTP calls) within an individual in the evolutionary search. The action templates are developed based on the semantics of HTTP methods and are used to manipulate the web services' resources. In addition, we propose five novel sampling strategies with four sampling methods (i.e., resource-based sampling) for the test cases that can use one or more of these templates. The strategies are further supported with a set of new, specialized mutation operators (i.e., resource-based mutation) in the evolutionary search that take into account the use of these resources in the generated test cases. Moreover, we propose a novel dependency handling to detect possible dependencies among the resources in the tested applications. The resource-based sampling and mutations are then enhanced by exploiting the information of these detected dependencies. To evaluate our approach, we implemented it as an extension to the EvoMaster tool, and conducted an empirical study with two selected baselines on 7 open-source and 12 synthetic RESTful web services. Results show that our novel resource-based approach with dependency handling obtains a significant improvement in performance over the baselines, e.g., up to +130.7% relative improvement (growing from +27.9% to +64.3%) on line coverage.

Keywords Search-based software testing · RESTful APIs · Web services · Test case generation

Communicated by: Antonia Bertolino

✉ Man Zhang
man.zhang@kristiania.no

Extended author information available on the last page of the article.

1 Introduction

REST is an architectural style composed of a set of design constraints on architecture, communication, and web resources for building web services using HTTP (Fielding 2000; Allamaraju 2010). It is useful for developing web services with public APIs over a network. Currently, REST has been applied by many companies for providing their services over the Internet, e.g., Google,[1] Amazon,[2] and Twitter.[3] However, in spite of their widespread use, testing such RESTful web services is quite challenging (Bozkurt et al. 2013; Canfora and Di Penta 2009) (e.g., due to dealing with databases and calls over a network).

In this paper, we propose a novel approach to enhance the automated generation of systems tests for RESTful web services using search-based techniques (Harman et al. 2012). To generate tests using search-based techniques, we use the Many Independent Objectives evolutionary algorithm (MIO) (Arcuri 2018b). The MIO algorithm is specialized for system test case generation with the aim of maximizing code coverage and fault finding. The MIO algorithm is inspired by the (1+1) Evolutionary Algorithm (Droste et al. 1998), so that an individual is mainly manipulated by sampling and mutation (no crossover). However, our novel techniques could be extended and adapted in other search algorithms.

We implemented our approach as an extension of an existing white-box test case generation tool, called EvoMaster (Arcuri 2018a; 2019). The tool targets RESTful APIs, and generates test cases in the JUnit format, where sequences of HTTP calls are made to test such APIs. During the search, EvoMaster assesses the fitness of individual test cases using runtime code-coverage metrics and fault finding ability.

Our novel approach is designed according to REST constraints on the handling of HTTP resources. First, based on the semantics of HTTP methods, we design a set of effective templates to structure test actions (i.e., HTTP calls) on one resource. Then, to distinguish templates based on their possible effects on following actions in a test, we add a property (i.e., *independent or possibly-independent*) to the template. A template is *independent* if actions with the template have no effect on following actions on any resource. Furthermore, we define a resource-based individual (i.e., a test case) by organizing actions on top of such templates. To improve the performance of the MIO algorithm with such individuals (i.e., the test cases evolved in the evolutionary search), we propose a *resource-based sampling* operator and *resource-based mutation* operators in our approach.

For the smart sampling operator, we define four sampling methods. At each sampling of a new random individual in the evolutionary search, one of these methods is applied to sample a new test. These methods are designed by taking into account the intra-relationships among the resources in the system under test (SUT). To determine how to select a method for sampling, we propose five strategies: *Equal-Probability* enables an uniformly distributed random selection; *Action-Based* enables a selection based on the proportions of applicable templates; *Used-Budget-Based* enables an adaptive selection based on the passing of search time; *Archive-Based* enables an adaptive selection based on their achieved improvement on the fitness; and *ConArchive-Based* enables an adaptive selection based on fitness improvement after a certain amount of sampling actions on one resource. Regarding mutation, we

[1] https://developers.google.com/drive/v2/reference/

[2] http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

[3] https://dev.twitter.com/rest/public

propose five novel operators to mutate the structure of the individuals, with respect to their use of the resources.

To seek a proper combinations of resources in the tests, we develop *resource dependency handling* which comprises *dependency identification*, and is integrated with *resource-based sampling* and *resource-based mutation*. In REST, there typically exist some dependencies among resources in the SUTs, and *dependency identification* is used to detect such dependencies based on *REST API Schema*, *Accessed SQL Tables* and *Fitness Feedback*. To exploit combinations of the resources, we enhance *resource-based sampling* and *resource-based mutation* with strategies involving the detected dependencies, e.g., sample actions on dependent resources in a test, and remove actions on a resource which is not related to any other resources.

We conducted an empirical study on our novel approach by comparing it with the existing work on white-box testing of RESTful APIs, i.e., the default version of EVOMASTER. Experiments were carried out on seven open-source case studies, which we used in previous work and gathered together in an open-source repository[4] made for experimentation in automated system testing of web/enterprise applications. To investigate the role of resource dependencies in more detail, we also created twelve synthetic case studies,[5] designed with various resource settings and relationships.

Results of our empirical study show that our novel techniques can significantly improve the performance of the test generation (e.g., relative improvement of line coverage is up to 130.7%) on SUTs that use fully independent, or closely connected, resources. Due to the randomness of the algorithm, in the worst case the improvements can be negligible.

The paper is an extension of a conference paper (Zhang et al. 2019), and the new contributions in this paper are summarized as follows:

- To enable proper handling of multiple resources, *dependency handling* is newly developed that consists of *dependency identification*, *resource-based sampling with dependency* and *resource-based mutation with dependency*. Besides, based on our experiments, *dependency handling* achieves a further improvement on our resource-based solution.
- To better assess our proposed resource-based solution, we designed the synthetic RESTful API generator[6] for automatically generating RESTful APIs with various resource-based configurable properties, i.e., a number of resources, applied HTTP methods, a number of dependencies, a constructed resource graph, different types of dependencies, and show/hide dependency on URIs. Note that the generator is also useful to setup experiments for studying other RESTful APIs-related approaches.
- We designed three resource graphs with two dependency-related constraints and two URI generations that generate a total of 12 synthetic RESTful APIs. Those are new case studies for our experiment in this extension.
- With our novel techniques, we answer new research questions and more experiment settings. Compared with the 22 experiment settings in the conference version, 52 experiment settings are conducted in this extension.

[4]https://github.com/EMResearch/EMB

[5]https://github.com/EMResearch/artificial-rest-api

[6]https://www.evomaster.org

- To investigate the performance of our proposed approach on the various case studies, we characterize in detail five of the real RESTful APIs. We manually derived the resource dependency graphs for each of these APIs by checking their implementation in details. Then, the impact of resources and their dependencies on the SUTs are discussed.
- All the experiments are newly conducted with the latest tool version of EVOMASTER.
- Regarding the main changes in the paper, Sections 4, 6, 7.2 and 8 are all new.

The rest of the paper is organized as follows. In Section 2, we provide a brief description on related background topics, needed to better understand the rest of the paper. Section 3 discusses related work. The overview of the proposed approach is presented in Section 4, followed by Resource-based MIO (Section 5) and Resource Dependency Handling (Section 6). The applied case studies are presented in Section 7, while the empirical study and its results are discussed in Section 8. We discuss threats to validity in Section 9 and conclude the paper in Section 10.

2 Background

2.1 HTTP and REST

The Hypertext Transfer Protocol (HTTP) is an application protocol used by the World Wide Web. The protocol defines a set of rules for data communication over a network. HTTP messages are composed of four main elements:

- *Resource path*: indicates the target of the request, referring to a resource that will be accessed. The resource path defines Uniform Resource Identifier (URI), which can include *query parameters*. These latter are pairs of "key=value", separated by & symbols, following the resource path after a "?", e.g., `/api/someResource?x=foo&y=bar`.
- *Method/Verb*: the type of operation that is performed on the specified resource. The types of operations include: i) *GET*: retrieve the specified resource that should be returned in the *Body* of the response; ii) *HEAD*: similar to GET, but without any payload; iii) *POST*: send data to the server. This is often used to create a new resource; iv) *DELETE*: delete the specified resource; v) *PUT*: similar to POST. But PUT is idempotent, so it is usually employed for replacing an existing resource with a new one; vi) *PATCH*: partially update the specified resource.
- *Headers*: carries additional information with the request or the response.
- *Body*: carries the payload of the message, if any.

The Representational State Transfer (REST) is designed for building web services on top of HTTP. The concept of REST was first introduced by Fielding in his PhD thesis (Fielding 2000) in 2000, and it is now widely applied in industry, e.g., Google,[1] Amazon,[2] and Twitter.[3] REST is not a protocol, but rather it defines an architectural style composed of a set of design constraints on how to build web services using HTTP. A web service using REST should follow some specific guidelines, e.g., the architecture should be client-server by separating the user interface concerns from the data storage concerns, and communications between client and server should be stateless. To manage resources, REST suggests that: i) resources should be identified in the requests by using Uniform Resource Identifiers (URIs); ii) resources should be separated from their representation, i.e., the machine-readable data describing the current state of a resource; iii) the implemented operations should always be in accord with the protocol semantics of HTTP (for example, you should not delete a resource when handling a GET request). In this paper, our novel approach is based on the assumption that the web services are written following the REST constraints, especially following the protocol semantics of HTTP method to develop endpoints. However, our approach should not have any significant negative side-effects when dealing with non-conforming APIs.

In a RESTful API, data can be transfered in any format. However, one of the most typical format is JSON (JavaScript Object Notation). For example, all the SUTs in our empirical study use JSON. Furthermore, JSON is also typically used to specify the schemas of such APIs (e.g., with OpenAPI/Swagger[7]).

2.2 The MIO Algorithm

The Many Independent Objective (MIO) algorithm (Arcuri 2018b) is an evolutionary algorithm specialized for system test case generation in the context of white-box testing. The algorithm is inspired by the (1+1) Evolutionary Algorithm (Droste et al. 1998) with a dynamic population, adaptive exploration/exploitation control and feedback-directed sampling.

Algorithm 1 shows the pseudo-code representation of the MIO algorithm. The search is started with no populations. Each time a testing target is "reached" when executing a test, a new empty population is created for such target, and the test is added to it. For example, when a statement like "`if(predicate)`" is executed (i.e., "reached"), there will be two branch-coverage targets, representing the "then" and "else" branches. Unless the evaluation of the predicate leads to an exception, one of these two branches will be "covered", whereas the other will be "reached" but "uncovered". Afterwards, at each step, with a probability P_r, MIO either samples new tests at random or samples (followed by a mutation) a test from a population that includes reached but uncovered targets.

As the next step, the sampled/mutated test may be added to the populations if it achieves any improvement on covered targets. Once the size of a population exceeds the population limit n, the test with worst performance is removed. In addition, at the end of a step, if an optimization target is covered, the associated population size is shrunk to one, and no more sampling is allowed from that population. At the end, the search outputs a test suite (i.e., a set of test cases) based on the best tests in each population. In the context of testing, users may care about what targets are covered, rather than how heuristically close they are to be covered. Therefore, MIO employs a technique called *feedback-directed* sampling. This technique guides the search to focus the sampling on populations that exhibit recent improvements in the achieved fitness value. This enables an effective way to reduce search time spent on infeasible targets (Arcuri 2018b). Moreover, to make a trade-off between exploration and exploitation of the search landscape, MIO is integrated with adaptive parameter control. When the search reaches a certain point F (e.g., 50% of the budget has been used), the search starts to focus more on exploitation by reducing the probability of random sampling P_r.

[7] https://swagger.io/

Algorithm 1 Pseudo-code of the MIO Algorithm (Arcuri 2018b).

```
Input  : Stopping condition $C$, Fitness function $\delta$, Population size $n$, Probability for
         random sampling $P_r$, Start of focused search $F$
Output: Archive of optimised individuals $A$
1  $T \leftarrow SetEmptyPopulations()$
2  $A \leftarrow \{\}$
3  while $\neg C$ do
4      if $P_r > rand()$ then
5          $p \leftarrow RandomIndividual()$
6      else
7          $p \leftarrow SampleIndividual(T)$
8          $p \leftarrow Mutate(p)$
9      foreach element k of ReachedTargets(p) do
10         if $IsOptimizationTargetCovered(k)$ then
11             $UpdateArchive(A, p)$
12             $T \leftarrow T \setminus T_k$
13         else
14             $T_k \leftarrow T_k \cup \{p\}$
15             if $|T_k| > n$ then
16                 $RemoveWorstTest(T_k, \delta)$
17     $UpdateParameters(F, P_r, n)$
```

- *SetEmptyPopulations()* is to initialize an empty population for each of optimization targets; k represents a target, and T_k is a population for the target k.

In this paper, we introduce *resource-based individual* by reformulating the individual for the REST problem, and propose new sampling and mutation operators that enables handling of resource and dependency in the context of RESTful web services.

The proposed solutions could be applicable and adapted to other evolutionary algorithms addressing test generation for RESTful web services. As MIO was the best in previous studies (Arcuri 2018b; 2019) (in terms of the fitness function, which uses code coverage and fault detection), we employ MIO with the newly proposed solutions in this paper to assess improvements on the problem of testing RESTful web services.

2.3 RESTful API Test Case Generation

In Arcuri (2019), we proposed a search-based approach for automatically generating system tests for RESTful web services, using the MIO algorithm (recall Section 2.2). Testing targets for the fitness function were defined with three perspectives: 1) coverage of statements; 2) coverage of branches; and 3) returned HTTP status codes. In addition, to improve the performance of sampling in the context of REST, smart sampling techniques were developed for sampling tests (i.e., sequences of HTTP calls) with pre-defined structures by taking into account RESTful API design. The structures are described as follows:

– *GET Template*: k POSTs with GET, i.e., add k POSTs before GET. This template attempts to make specified resources available before making a GET on them. k is configurable, e.g., $k = 2$ indicates that add 2 POSTs before a GET.
– *POST Template*: just a single POST.
– *PUT Template*: POSTs with PUT, i.e., add 0, 1, or more POSTs before PUT with a probability p. PUT is an idempotent method. When making a PUT on a resource that does not exist, the PUT could either create it or return an 4xx status. So the template

involves a probability for sampling a test with either a single PUT, or POSTs followed by a single PUT.

- *PATCH Template*: POSTs with PATCH, i.e., add 0, 1, or more POSTs before a PATCH, and possibly add a second PATCH operation at end with a probability p. The second PATCH is used to check if POSTs and the first PATCH are doing partial updates instead of a full resource replacement.
- *DELETE Template*: POSTs with DELETE, i.e., add 0, 1 or more POST operations followed by a single DELETE.

The approach was implemented as an open-source tool, named EvoMaster.[6] It has two components (Arcuri 2018a): *Core* which mainly implements a set of search algorithms for test case generation (e.g., WTS Rojas et al. 2017); and *Driver* that is responsible for controlling (e.g., start, stop, and reset) the SUT, and for instrumenting its source code. With it, the search algorithm assesses the fitness of individual test cases using runtime code-coverage metrics (e.g., lines and branches) and fault finding ability (e.g., based on HTTP status codes such as 500, and on discrepancies of the results with what is expected based on the API schema). For SUTs that compile into JVM byte-code, the instrumentation to collect code-coverage metrics is done fully automatically by the *Driver* when such SUTs are started.

EvoMaster can also analyse all interactions with SQL databases (Arcuri and Galeotti 2020), to improve the generation of test cases (e.g., by analysing which data is queried). Furthermore, to make the test independent from each other, the databases are reset at each fitness evaluation (just the data is cleaned, as there is no need to re-create the SQL schemas or re-start the databases).

3 Related Work

Recently, there has been an increase in research on black-box automated test generation based on REST API schemas defined with OpenAPI (Atlidakis et al. 2019; Karlsson et al. 2020; Viglianisi et al. 2020; Ed-douibi et al. 2018). Atlidakis et al. (2019) developed *RESTler* to generate test sequences based on dependencies inferred from OpenAPI specifications and analysis on dynamic feedback from responses (e.g., status code) during test execution. In their approach, there exists a mutation dictionary for configuring test inputs regarding data types. Karlsson et al. (2020) introduced an approach to produce property-based tests based on OpenAPI specifications. Viglianisi et al. (2020) employ *Operation Dependency Graph* to construct data dependencies among operations. The graph is initialized with an OpenAPI specification and evolved during test execution. Then, tests are generated by ordering the operations based on the graph and considering the semantics of the operations. Ed-douibi et al. (2018) proposed an approach to first generate test models based on OpenAPI specifications, then produce tests with the models.

OpenAPI specifications are also required in our approach for accessing and characterizing the APIs of the SUT (e.g., which endpoints are available, and what types of data they expect as input). As we first introduced in Arcuri (2019) and Zhang et al. (2019), the OpenAPI specifications are further utilized for identifying resource dependencies, similarly to what recently done by approaches like in Atlidakis et al. (2019) and Karlsson et al. (2020). However, the dependencies we identify are for *resources*, and not just *operations*. In our context, we consider that a REST API consists of resources with corresponding operations performed on them, and there typically exist some dependencies among the different resources. To identify such dependencies, we analyze the API specification and collect runtime feedback. We then use the derived dependencies to improve the search by enhancing how test cases are generated and evolved. A key difference here is that, in contrast to all existing work, we can further employ white-box information to exploit and derive the dependency graphs. For instance, if a REST API interacts with a database, manipulating resources often leads to further access data in such database, e.g., retrieving a resource might require to query data from some SQL table(s). This information about which tables are accessed at runtime can be obtained with EvoMaster. Such runtime information helps to identify a relationship between a resource and SQL tables. Thus, through the analysis of which tables are accessed at runtime we can further derive possible dependencies among resources. In this work, to derive the dependencies, we also employ code coverage and the other search-based code-level heuristics by checking the effects on involving different resources.

Note that OpenAPI specifications do not need to be necessarily written by hand. Depending on the libraries/frameworks used to implement the RESTful web services (e.g., with the popular Spring), such OpenAPI schemas can be automatically inferred (e.g., using libraries such as SpringFox and SpringDoc). So, the lack of an existing OpenAPI schema is not necessarily a showstopper preventing the use of tools such as EvoMaster.

Besides existing work on black-box testing based on industry-standards such as OpenAPI/Swagger[7] schemas, there exist previous approaches to test REST APIs that rely on formal models and/or ad-hoc schema specifications (Chakrabarti and Kumar 2009; Chakrabarti and Rodriquez 2010; Fertig and Braun 2015; Pinheiro et al. 2013; Lamela Seijas et al. 2013). The models often describe test inputs, exposed methods of SUTs, behaviors of SUTs, specific characteristics of REST or testing requirements. An XML schema specification used for testing was introduced by Chakrabarti and Kumar (2009). This was then extended in Chakrabarti and Rodriquez (2010) to formalize connections among resources of a RESTful service, and further focus on testing such “connectedness”. Fertig and Braun (2015) developed a Domain Specific Language to describe APIs, including HTTP methods, authentication and resource model. A set of test cases can be generated from such a model. Lamela Seijas et al. (2013) proposed an approach to generate test cases based on property-based test models, and UML state machines are applied (Pinheiro et al. 2013) to construct behavior models for test case generation.

In contrast to such earlier work, to make our approach and tool as usable as possible for practitioners in industry, we rely on industry standards such as OpenAPI/Swagger specifications. Our techniques do not require practitioners to write academic formal models to be able to use our techniques in practice on their systems.

Besides improving coverage of an API, it is important to design new techniques to detect different categories of faults in such APIs. Segura et al. (2017) developed an approach for the metamorphic testing of RESTful Web APIs, for tackling the oracle problem. The approach defined six abstract relations covering possible metamorphic relations in a RESTful SUT. Those can be used to detect faults when test data is generated for which those metamorphic relations are not satisfied. In this work, we do not propose any new approach to tackle the oracle problem in API testing.

All the above are black-box testing approaches that are different from our approach, i.e., white-box system test case generation for RESTful APIs. As discussed, in Arcuri (2017) and Arcuri (2019) our team proposed a means of generating test cases for RESTful APIs by using search-based techniques to create sequences of HTTP calls that has been implemented as a prototype tool, named EvoMaster. In addition, a major novelty is that SQL operations are enabled in EvoMaster for producing tests with handling of databases (Arcuri and Galeotti 2019; 2020). This is a search-based software testing (Ali et al. 2010) approach, relying on information obtained from the API specifications and code instrumentation to generate test cases. It does not, however, identify relationships between resources and consider the relationships when generating these test cases (apart from some basic templates introduced in Arcuri (2019)). Therefore, in this paper, we propose a complete resource-based approach, built upon EvoMaster, by detecting resource dependencies, introducing resource-dependency handling methods and strategies, as well as developing tailored sampling and mutation operators.

Another key difference with existing work is that, not only EvoMaster is open-source and freely available on GitHub,[6] but also it is actively supported, with extensive documentation on how to use it. This is essential to enable replicated studies, and for using EvoMaster in studies involving tool/technique comparisons. For example, in this work, we could compare our novel techniques only with the base version of EvoMaster, as no other tool was available.

4 Overview of the Proposal

REST defines a set of guidelines for creating stateless services which can be accessed over a network using HTTP. Figure 1 shows a snippet example of a specification of API following REST guidelines. The specification is defined using an OpenAPI/Swagger[7] schema. In the example, the APIs are structured with resource URIs, and relevant HTTP methods are defined for each resource.

In our context, an individual is a test case composed of a sequence of HTTP calls. Each HTTP call consists of a specific HTTP method and an associated resource, defined by its URI for performing some actions on the associated resource. Consider an API that deals with *products* and *warehouses*, as the example in Fig. 1. Some tests (in pseudo-code) for such API are shown in Fig. 2. Each line represents an action

```
"/warehouses":{
  "get":{"operationId":"getAllWarehouse"...},},
"/warehouses/{warehouse}":{
  "post":{"operationId":"addWarehouse"...},
  "delete":{"operationId":"deleteWarehouse"...},},
"/products":{
  "get":{"operationId":"getAllProducts"...},},
"/products/{productName}":{
  "post":{
    "operationId":"addProduct",
    "produces": ["application/json"],
    "parameters": [{
        "name": "productName",
        "in": "path",
        "required": true,
        "type": "string"
      },{
        "name": "warehouse",
        "in": "query",
        "required": true,
        "type": "string"
      },{
        "name": "quantity",
        "in": "query",
        "required": false,
        "type": "integer"}, ... ],...},
  "get":{"operationId":"getProductByName"...},
  "delete":{"operationId":"deleteProductByName"...},},
```

Fig. 1 Snippet example of OpenAPI/Swagger JSON definitions for a RESTful API

```
1 POST /warehouses/bar?capacity=40
2 DELETE /products/foo
3 POST /products/foo?warehouse=bar&quantity=20
```

Example 1. *foo* is created successfully

```
1 DELETE /warehouses/bar
2 DELETE /products/foo
3 POST /products/foo?warehouse=bar&quantity=20
```

Example 2. *foo* cannot be created because the referred *bar* cannot be found

```
1 POST /warehouses/bar?capacity=40
2 POST /products/foo?warehouse=bar&quantity=30
3 POST /products/foo?warehouse=bar&quantity=10
```

Example 3. *foo* cannot be created because the resource *foo* already exists

```
1 POST /warehouses/bar?capacity=40
2 DELETE /products/foo
3 POST /products/foo?warehouse=bar&quantity=100
```

Example 4. *foo* cannot be created because the referred *bar* has not enough space

Fig. 2 An example of a HTTP request (at line 3) under different status of resources

which follows the format <*a method on a resource path with/without parameters*><*the method on the path with values of the parameters*>. For instance, the HTTP call POST */products/foo?warehouse=bar&quantity*=20 is an action to add *20* new products named *foo* in a warehouse named *bar*.

Note that, to make the examples more readable, here a resource is created with POST using query parameters. But, in practice, usually the data would be in the body payload of the requests (as URLs have small size limits). Furthermore, for simplicity we are considering a POST that fails if the resource already exists. A different approach could have been to rather use PUT operations to create and/or update these resources.

Regarding the action, we can identify a resource *foo* of type product directly handled by this call, and a referred resource *bar* warehouse. When executing this action in different tests, the status of the resources (i.e., *foo* and *bar*) might be different in the SUT's back-end, and so then result in different code executions. As demonstrated in Fig. 2, four tests represent this action (at line 3) with different statuses of the resources, i.e., Example 1: the warehouse *bar* exists and has space to store 40 new products, and the product *foo* does not exist; Example 2: the warehouse *bar* does not exist; Example 3: the warehouse *bar* exists, and the product *foo* exists; Example 4: the warehouse *bar* exists but there is no enough space to store 100 products. With each of the states, the call at line 3 executes different paths in the source code of the SUT. From a testing perspective, exploring those different possible states of resources may help to improve coverage of the testing targets (e.g., lines, branches and HTTP status codes).

Typically, search-based techniques use random sampling to create new individuals. In our context, an individual is a series of HTTP calls, where the resources are identified with URIs. Those depend on variables that can be part of the search, such as path elements and query parameters. Depending on quantity and complexity of those variables, sampling them at random would lead to different URIs (especially when the variables are strings).

Furthermore, different but related resources will have different URIs, where the relations will be expressed by some specific variable (e.g., an ID that is a path element in a resource, and it is referenced in another resource as a query parameter).

In this manner, it is unlikely we will be able to generate several HTTP calls at random to perform on relevant, related resources, e.g., line 2 and line 3 on *foo* product in Fig. 2. If there exist some relationships among resources and actions just as in the *product-warehouse* example, then it is very unlikely to produce tests that result in good coverage. Therefore, we propose *Resource-based MIO* (Section 5) to enable handling of individuals with respect to resources, i.e., resource-based individual, resource-based sampling and resource-based mutation.

There typically exist some dependencies among resources in a RESTful API. Often, the dependencies can be identified based on hierarchical structures of the URIs. For example, the resource *foo* is hierarchically related to the collection of all products called */products*, i.e., the resource *products/foo* belongs to the collection resource */products*. However, there might exist other kinds of relations, e.g., a *product* depends on a *warehouse*, and that information is not part of the path element in the URI. To derive such further kinds of dependencies and exploit them to generate higher coverage tests, we propose *Resource Dependency Handling* (Section 6).

Algorithm 2 Pseudo-code of the resource-based MIO algorithm with dependency handling.

```
Input  : Stopping condition $C$, Fitness function $\delta$, Population size $n$, Probability for
         sampling $P_r$, Probability for resource-based sampling $P_s$, Probability for
         applying dependency handling $P_d$, Enabling of dependency pre-matching
         $PM$, Start of focused search $F$
Output: Archive of optimised individuals $A$
1  $T \leftarrow SetEmptyPopulations()$
2  $D \leftarrow \{\}$
3  $A \leftarrow \{\}$
4  if $P_d > 0 \wedge PM$ then
5      $D \leftarrow DeriveDependencyBasedOnSchema()$
6  while $\neg C$ do
7      if $P_r > rand()$ then
8          if $P_s > rand()$ then
9              $p \leftarrow ResourceSampling(P_d > rand(), D)$
10         else
11             $p \leftarrow RandomSampling()$
12     else
13         $p_b \leftarrow SampleIndividual(T)$
14         if $P_s > 0$ then
15             $p \leftarrow ResourceMutate(T, p_b, P_d > rand(), D)$
16         else
17             $p \leftarrow Mutate(T, p_d)$
18     if $P_d > 0$ then
19         $E \leftarrow OperatedTables(p)$
20         $D \leftarrow DeriveDependencyBasedOnTables(p, PM, D, E)$
21         $D \leftarrow DeriveDependencyBasedOnFintess(p_b, p, D)$
22     $T \leftarrow UpdatePopulationsAndArchive(T, A, p)$
23     $UpdateParameters(F, P_r, n)$
```

Algorithm 2 represents how the proposed techniques are integrated in MIO (Algorithm 1). These techniques are controlled with parameters, i.e., probability for resource-based sampling P_s, probability for applying dependency handling P_d, and enabling of dependency pre-matching PM. At the beginning of the search, dependencies of the SUT are typically unknown, i.e., an empty D. But there might be some information on the dependencies stated in the RESTful API schema of the SUT (e.g., based on hierarchical relationships in the URI path elements). So we develope a pre-matching process to initialize dependencies with the schema (Section 6.1), the process can be applied when dependency handling and dependency pre-matching are enabled, i.e., $P_d > 0$ and PM (see lines 4-5 in Algorithm 2). During the search, based on a specified probability for resource-based sampling P_s, resource-based sampling (see lines 8-9, discussed in Section 5.2) and mutation (see lines 14-15, discussed in Section 5.3) are applied to sample and mutate an individual regarding resources. Note that the individual is a test for REST API. *The resource-based sampling* and *mutation* can be enabled with dependency-based strategies for producing tests, e.g., sample an individual with actions on dependent resources. The strategies are controlled by the probability P_d and enabled when $P_d > rand()$ is evaluated as true at line 9 (for the sampling introduced in Section 6.2) and line 15 (for the mutator introduced in Section 6.3). After the individual is executed on the SUT and its fitness is evaluated, we make use of the information on which database tables were accessed and changes on fitness to derive the dependencies among resources (see lines 18-21 in Algorithm 2 and introduced in Section 6.1). Based on such dependency handling, the derived dependencies are updated and refined over each iteration of the search. At the end of search, the best individuals are selected to generate the output test suite based on their code coverage and fault finding.

5 Resource-Based MIO

5.1 Resource-Based Individual Representation

To enable the handling of individuals regarding resources, we defined a set of templates that list meaningful combinations of HTTP methods based on their semantics. Then, an individual is reformulated as a sequence of *resource-handling*s, and each of the *resource-handling*s is a sequence of actions (i.e., HTTP calls) on one resource based on the templates (e.g., POST-DELETE). With such an individual, the search can be applied to handle actions based on resources (e.g., sample actions on the same resource) and manipulate resources (e.g., add actions on a new resource), instead of handling each action independently. However, search is still needed, for example to evolve the right query parameters for the URIs, the content of the body payloads (e.g., JSON objects), and the HTTP headers.

Based on the different types of HTTP methods, we define templates in Table 1. Note that we intentionally make the template short (i.e., at most combine two different types of HTTP methods) to allow small modifications on the structure of the individuals. As the example shown in Fig. 2, code coverage does often depend on the status of the resources (e.g., if they exist or not). Different types of HTTP methods can help to manipulate the status of a resource before a following action is executed:

– *POST (PUT)* and *DELETE* may be applied to handle the existence of a resource;
– *PUT and PATCH* may be applied to update some properties of a resource when the resource exists;
– *GET and HEAD* typically cannot change a status of a resource.

Table 1 Definitions of resource-based templates used to generate tests regarding resources

#	Description	**independent?**	Template
1	To retrieve a resource	Yes	GET
2	To (partially) update an nonexistent resource	Yes	PATCH
3	To delete a nonexistent resource	Yes	DELETE
4	To replace a nonexistent resource	Yes	PUT
5	To create a resource	No	POST
6	To create an existing resource	No	**CREATE**-POST
7	To retrieve an existing resource	No	**CREATE**-GET
8	To replace an existing resource	No	**CREATE**-PUT
9	To (partially) update an existing resource	No	**CREATE**-PATCH(-PATCH)
10	To delete an existing resource	No	**CREATE**-DELETE

Note that **CREATE** means either POST or PUT with 20% probability and #6-10 are only applicable if there exists POST or PUT on the resource or one of its ancestors' resource for creating resources

In the design of the templates, we only focused on the existence of resources. This is because the *update* action is restricted by the existence condition. For example, assume that an *update* (i.e., PATCH) performs on an existing resource and a following action DELETE improves the code coverage of the tests. This would normally be due to the existence of the resource itself rather than what update operation was previously performed on it. Even if the success of a DELETE was dependent on a specific value in a field of the resource, such a value could have been directly provided in the operation that created the resource in the first place (e.g., a POST). Therefore, an *update* operation on the resource would not be needed in this context.

In the templates, we only use methods (i.e., POST or PUT) to prepare the existence condition of a resource. We ignore DELETE to make the resources non-existent, i.e., remove resources. This is because, in EvoMaster, the SUT is reset at each test execution (e.g., the database state is cleaned before each test execution). Furthermore, as the search starts by usually choosing new values at random for the parameters of the actions, this means that it is unlikely that the newly sampled values have been previously applied on a creation method (e.g., POST) for creating that corresponding resource in one specific test. In this case, a DELETE is almost the same as the situation when no creation method is used. Thus, executing an extra DELETE per resource (i.e., add DELETE as the first action to templates #1-#5) would be probably a waste of the search budget (e.g., by making the test unnecessarily longer, and so more time consuming to run).

We designed 10 templates (shown in Table 1) based on all types of HTTP methods along with whether the related resource exists. Only 5 new templates are introduced, as templates #5, #7-#10 are the same as the templates from our previous work (Arcuri 2019) (Section 2.3). These templates were applied on the sampling of whole test cases. On the other hand, in this paper, we apply them on a fragment of a test with the aim of handling multiple resources, and each fragment is a sequence of actions performed on a same resource (i.e., a test case can be composed of one or more fragments). In addition, we identify properties for all the templates, i.e., ***independent*** and ***possibly-dependent***.

There might exist some unknown internal relations among resources in the SUT, e.g., */products* /{*productName*} depends on */warehouses*/{*warehouseName*} in Fig. 1. So, it is not clear, based on the URIs alone, whether actions executed first have effects on the following actions in a test. But actions that never have an impact can be derived based on the semantics of HTTP methods (e.g., GET operations are not supposed to change the state of the resources in the SUT).

In the context of testing, we also capitalize on invalid sequences of actions, i.e., #2-#3, that aim to operate on a resource that does not exist. Since the actions are expected as failure operations, they probably do not change any state of the resources. Therefore, we identify ***independent*** templates (#1-#3) that, when actions with the template are executed, do not have any effect on follow-up actions on any resource.

PUT might be implemented to create or update a resource (both options are valid according to the HTTP semantic). However, the implementation may vary among endpoints or case studies and is typically not exposed to the schema. Therefore, to cover the potential creation by PUT, we consider PUT with a 20% probability of creating a resource (i.e., see CREATE applied in #6-#10 in Table 1). Regarding a single PUT (i.e., #4), we consider its semantic as update, thus, #4 is independent. We further define a ***possibly-dependent*** template as a template for which independence cannot be assumed. Note that a ***possibly-dependent*** template might or might not be dependent, because it varies from resource to resource, and dependency of resources is usually unknown before a search starts. Moreover, we further identify resources based on their applicable templates. An *independent resource* is a resource which can only be manipulated with the **independent** template, and a *possibly-dependent resource* is a resource that can be manipulated with at least one **possibly-dependent** template.

With the defined templates, we formulate the *resource-based individual* (shown in Fig. 3) as a sequence of *resource-handling* constrained with one of the templates, i.e., $(R_1, ..., R_i, ...R_n)$ where n is a number of the *resource-handling*. Thus, the *resource-handling* is composed of operations that perform a sequence of actions on the same resource R_i, i.e., $R_i = (A_{i,1}...A_{i,j}...A_{i,m_i})$ where m_i is a number of actions of R_i and $m_i > 0$. Each action is composed of a sequence of genes, i.e., $A_{i,j} = (G_{i,j,1}...G_{i,j,k}...G_{i,j,t_{ij}})$ where t_{ij} is a number of genes in the jth action of the ith *resource-handling* and $t_{ij} \geq 0$. Note that t_{ij} might be 0 if there does not exist any gene in a REST Action, e.g., GET */warehouses*. With the hierarchical formulation, the individual can be seen as either a sequence of actions or a sequence of genes.

Figure 4 shows an example of a test for handling */warehouses*/{*warehouseName*} with template #1 and */products*/{*productName*} with template #7 for *products-warehouse* APIs. In this example, the test covers a scenario that handles two resources, i.e., *warehouse* and *product*. Regarding the handling (lines 2-3) with template #7 for retrieving a *product* with a specified *productName*, the POST action at line 2 is added to prepare the resource for the GET action at line 3, then value on the path element *productName* in the POST is

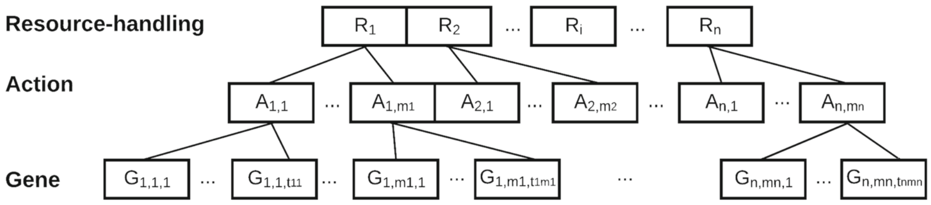


Fig. 3 Representation of *resource-based* individual at hierarchies of *Resource-handling*, *Action* and *Genes*. R_i represents ith *resource-handling* in a test, $A_{i,j}$ represents jth action of ith *resource-handling*, and $G_{i,j,k}$ represents kth gene in jth action of ith *resource-handling*

```
GET /warehouses/abc
```

actions with #1 on $/warehouses/\{warehouseName\}$

```
POST /products/foo?warehouse=bar&
     quantity=20
GET /products/foo
```

actions with #7 on $/products/\{productName\}$

Fig. 4 An example (Example 5) of a test for handling */warehouses* with template #1 and */products/*{*productName*} with template #7

bounded with the same value as in the GET. In addition, the representation of the test as a resource-based individual is shown in Fig. 5.

In the following subsections, we explain how to sample (Section 5.2) and mutate (Section 5.3) such an individual during the search.

Note: most of our templates are based on the semantics of HTTP. But, what if an API is not following such semantics, and for examples it has GET requests with side-effects? Most likely, in such cases we would see a decrease in performance compared with the default version of EvoMaster without our novel templates. However, one of the greatest benefits and strengths of evolutionary search is its adaptiveness. Test cases with lower fitness will have lower chances of reproducing. As we apply our templates only with a given probability, we still sample "regular" test cases. And, if those test cases will have higher fitness, they will reproduce more often and take over during the search. This could lead to a "slower" start, in which some search budget would be wasted in sampling tests with our templates. However, given enough search budget, those side-effects at the beginning of the search might become negligible.

5.2 Resource-Based Sampling

In this paper, we propose four methods to sample individuals as shown in Table 2. For each of the methods, we define an applicable precondition regarding exposed HTTP methods on resources in a SUT. In other words, the method can be applied to sample a test only if the precondition is satisfied.

One rationale behind the use of these methods is to distinguish related actions from independent actions (i.e., actions based on the independent template). By isolating actions with independent templates, we can reduce unnecessary invocations (i.e., HTTP calls) during the search. As the example in Fig. 4, the first action is to GET a warehouse by *warehouseName*. This is fully independent from the following two actions, as GET operations are not supposed to have side-effects. When applying a mutation on the parameter *warehouseName* of that action, it is highly possible that there is no improvement (e.g., covering new statements) achieved by the invocations of the remaining two actions. Besides, for testing two or more

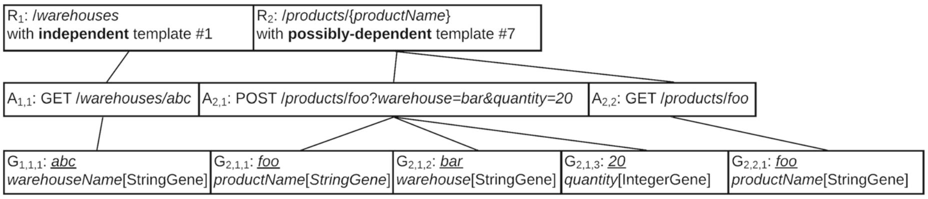

R1: /warehouses with independent template #1
R2: /products/{productName} with possibly-dependent template #7
A1,1: GET /warehouses/abc
A2,1: POST /products/foo?warehouse=bar&quantity=20
A2,2: GET /products/foo
G1,1,1: abc warehouseName[StringGene]
G2,1,1: foo productName[StringGene]
G2,1,2: bar warehouse[StringGene]
G2,1,3: 20 quantity[IntegerGene]
G2,2,1: foo productName[StringGene]

Fig. 5 An example of the test in Fig. 4 formulated with *resource-based* individual. An instance of $G_{i,j,k}$ is represented with its value, its *name* and its [type]

Table 2 Methods of resource-based sampling to select resources manipulated in a test

Method	Description	Precondition
S1iR	Sample one resource with independent template	At least one independent template exists in the SUT
S1dR	Sample one resource with possibly-dependent template	At least one possibly-dependent resource exists in the SUT
S2dR	Sample two resources, and only allows the last resource with non-parameter GET	At least two possibly-dependent resources exist in the SUT
SMdR	Sample more than two resources, and only allows the last resource with non-parameter GET	At least three possibly-dependent resources exist in the SUT

independent resources, it is better to have separate test cases (and thus, separate individuals). Shorter test cases are less costly to run, and easier to maintain. Therefore, for handling independent resources, we developed the *S1iR* strategy to sample one resource with one **independent** template in a test. Regarding handling of one resource, *S1dR* is defined to manipulate one resource with a **possibly-dependent** template in a test.

Another rationale is to explore dependency among resources. So we propose the *S2dR* strategy to sample the minimal dependent set by combining only two resources with **possibly-dependent** templates (e.g., the example in Fig. 6), and the *SMdR* for handling the possibility of complex, multi-resource dependencies. Note that when manipulating multiple resources in a test (i.e., *S2dR* and *SMdR* strategies), only an independent template with a non-parameter GET is allowed at the last position of the resources (e.g., *GET /warehouses* is allowed, but not *GET /warehouses/*{*warehouseName*}). As the non-parameter GET is often used to retrieve a collection of resources from the SUT, it may cover new statements due to new resources created by previous actions.

When a new individual is sampled with probability P_r (recall Algorithm 2), the individual is sampled with our resource-based sampling with probability P_s, or fully randomly with probability $1 - P_s$. In the former case, one of the four methods in Table 2 is chosen, with a probability denoted as P_m, and the probabilities of the four methods are $P_m(S1iR) + P_m(S1dR) + P_m(S2dR) + P_m(SMdR) = 1$.

Note that it is important to still be able to sample tests at random with no structure with probability $1 - P_s$. The templates we define in Table 1 should cover the most important cases, but likely not all. Which tests will then be most useful is left to the search to decide, based on the fitness function. Recall that in the evolutionary algorithms the most fit individuals have higher chances to reproduce.

The four methods enable us to sample tests with different considerations on resources. Since normally the resources involved vary from SUT to SUT, we designed five strategies to determine which method should be applied at the beginning of each sampling. For each applicable method, we set a probability P_m, which enables the selection process to be controlled by adjusting the appropriate selection probability. The five sample strategies are described as follows:

Equal-Probability: select methods at random with uniform probability, i.e., the probability for each applicable method is equal. It is calculated as $P_m = \frac{1.0}{n_m}$, where n_m is a number of applicable methods (i.e., the ones for which their preconditions are satisfied).

Action-Based: the probability for each applicable method is derived based on a number of independent or possible-dependent templates for all resources (this depends on which endpoints are available in the SUT, recall Table 1). It is calculated as:

$$P_m(S1iR) = \frac{n_{at} - n_{dt}}{(n_{at} + n_{dt} \times k)},$$

```
POST /warehouses/abc?capacity=40
```

actions with **possibly-dependent** #5 on */warehouses*

```
POST /products/foo?warehouse=bar&
      quantity=20
GET /products/foo
```

actions with **possibly-dependent** #7 on */products/*{*productName*}

Fig. 6 An example (Example 6) of a test sampled by S2dR method

where n_{at} is the sum of the number of applicable templates for all resources, n_{dt} is a sum of a number of possibly-dependent templates for all resources, and k is a configurable weight. Note that $k(\geq 0)$ indicates a degree to prioritize the *possibly-dependent* templates. We set it to 1 in this implementation. For example, there exist 5 *independent templates* and 15 *possibly-dependent* templates in a SUT, with $k = 1$, $P_m(S1iR) = \frac{15-10}{(15+10\times 1)} = 0.2$, thus, a sum of the methods with *possibly-dependent* templates is 0.8. For the probability of each of the methods,

$$P_m(s_i, w_i) = \frac{(1 - P_m(S1iR)) \times w_i}{\sum_{j=1}^{n_{dm}} w_j},$$

where $(s_i, w_i) \in S = \{(s_i, w_i)|\ s_i$ is an applicable method of *S1dR*, *S2dR* and *SMdR*, w_i is a weight of the method, $i = 1...n_{dm}$, and n_{dm} is the number of applicable methods except *S1iR*$\}$. The weight w_i for the method can be decided in various ways, e.g., constant 1. In this implementation, considering that a test involved more resources might take more budget to execute, the weight is defined based on the number of resources sampled with the method, i.e., $w(S1dR) = 3$, $w(S2dR) = 2$, and $w(SMdR) = 1$.

Used-Budget-Based: the probability for each applicable method is adaptive to the used budget (i.e., time or number of fitness evaluations) during search. The strategy samples an individual with one resource with a high probability (i.e., 0.8) at the beginning of the sampling (i.e., the used time budget for sampling is less than 50%), and then at later stages of the search it turns to sample a test with multiple resource methods. This approach allows the search to explore test cases with one resource first, and only spend effort on multiple resources if there is still enough available budget to allow for that. The reasoning is that test cases with multiple resources are harder to develop and more costly to run. They will be considered after the simpler test cases have been tried, and only if they provide a fitness improvement over those simpler test cases.

Archive-Based: the probability for each applicable method is adaptively determined by its performance during the search. The performance is evaluated based on the number of times that the method has helped to improve the fitness values (i.e., *improved times*) during the search. It is calculated as:

$$P_m(s_i, r_i) = P_m(s_i, r_i) \times (1 - \delta) + \delta \times \frac{r_i}{\sum_{j=1}^{n_m} r_j},$$

where $\delta = 0.1$, $(s_i, r_i) \in S = \{(s_i, r_i)|s_i$ is an applicable method, r_i is a rank for the applicable methods that is computed based on *improved times*, $i = 1...n_m$, and n_m is a number of the applicable methods$\}$.

ConArchive-Based (Controlled Archived Based): Distinguished from *Archive-Based* by a preparation phase. At the beginning of the sampling, the strategy samples an individual with one resource with a high probability. After a certain amount of search budget is used, the strategy starts to apply the same mechanism as *Archive-Based*. The strategy attempts to distinguish between improvements obtained by a combination of multiple resources from improvements obtained by different values on parameters of a resource. For instance, in Fig. 6, if a referred *bar* resource by actions at lines 2-3 is created by an action at line 1, the test could achieve an improvement due to the combination of the two resources (i.e., when the second resource depends on the first). But, during search, it would not be known whether the improvement is due to the actions on the first resource, actions on the second resource, or the combination of the two resources. If we first used some of the budget to sample the first resource and second resource separately in different tests, then later we may improve the chances of identifying whether the improvement is due to the combination (i.e., improve the chances to get the right *improved times* value for the strategies on multiple resources).

5.3 Resource-Based Mutation

When we sample a new individual during the search, we use our templates with some predefined structures. To improve the search, we need novel mutation operators that are aware of such structures. Therefore, we propose resource-based mutation that follows the same mechanism with MIO for mutating an individual: mutate values on parameters (i.e., the content of the HTTP calls, including headers, body payloads and path elements in the URIs), and mutate the structure of a test (i.e., adding or removing HTTP calls in the test).

Regarding value mutation, we apply the mutation on the parameters of the resources. The parameters of the resources are represented by the parameters of an action with longest path among the actions for the same resource. Once the value of any of the parameters is mutated, we update the other actions on the same resource with the same value. Considering a test shown in Fig. 6, in terms of */products/*{*productName*} resource, parameters of an action at line 2 is selected to represent the resource. When a value of the parameter *productName* is mutated, e.g., from *foo* to *ack*, then the same parameter *productName* on the action at line 3 is also required to be updated with the same value, i.e., from *foo* to *ack* as shown in Example 7 in Fig. 7.

In addition, we propose five operators to mutate the structure of the individuals, for exploiting relationships among resources and different templates on resources: ***DELETE***: delete a resource together with all its associated actions on that resource; ***SWAP***: swap the position of two resources together with all their associated actions; ***ADD***: add a new set of actions with a template for a new resource in the test; ***REPLACE***: replace a set of actions constrained with a template on a resource with another set of actions constrained with a template on a new different resource; ***MODIFY***: modify a set of actions on a resource with another template. For instance, a test applied with the *MODIFY* mutator is represented as Example 8 in Fig. 7.

6 Resource Dependency Heuristic Handling

In a RESTful web service, there typically exist some dependencies among the resources (recall Section 4). Then, a proper handling of resources according to their dependencies may help the generated tests to achieve a better coverage. In some cases, such dependencies might be partially identified by the *resource path*, i.e., hierarchical dependencies on the URIs, by just analyzing the OpenAPI/Swagger schemas. For example, the */products/*{*id*} resource is hierarchically related to the resource */products*. However, there might still exist

```
1 POST /warehouses/abc?
      capacity=40

2 POST /products/ack?warehouse
      =bar&quantity=20
3 GET /products/ack
```

Example 7. mutate values of parameters of */products/*{*productName*} resource (from *foo* to *ack*)

```
1 POST /warehouses/abc?capacity=40

2 POST /products/foo?warehouse=bar
      &quantity=20
3 DELETE /products/foo
```

Example 8. mutate a structure of the individual with MODIFY (a template of actions on */products/* {*productName*} are modified from POST-GET to POST-DELETE)

Fig. 7 Examples of applying resource-based mutations on a test in Fig. 6

some dependencies that are not exposed directly in the path elements of the URIs. For example, those could be HTTP query parameters or fields in body payload objects referencing IDs of other resources.

Therefore, we developed *dependency heuristic handling* to identify possible dependencies (Section 6.1). In addition, we enhance sampling (Section 6.2) and mutation (Section 6.3) by utilizing such identified dependencies among multiple resources.

6.1 Resource Dependency Detection

To identify dependencies among resources, we developed several solutions to derive them based on *REST API Schema*, *Accessed SQL Tables* and *Fitness Feedback* at runtime. Since the dependencies are heuristically inferred, each of the derived dependencies has an estimated probability ($\in$[0.0, 1.0]) for representing the confidence on the correctness of this inference. If a probability of a derived dependency is 1.0, then this indicates that we strongly consider this dependency to be correct. On the other hand, if the probability is 0.0, then this indicates that we strongly assume that there is no dependency between the two considered resources.

6.1.1 REST API Schema

Based on REST guidelines, a *resource path* and its *parameters* are typically designed with names of related resources. Thus, by identifying similar names, dependencies among resources might be derived directly based on the API Schema. As the snippet example of API schema shown in Fig. 1, there exists a dependency between *products* and *warehouses*, e.g., a *product* should always refer to an existing *warehouse*. In order to create a *product*, its creation method (e.g., POST) should be specified with the id/name of an existing *warehouse*. In this example, the dependent *warehouse* is identified by a query parameter in the POST */products/*{*productName*}, named *warehouse*.

By detecting matched names among query parameters and path elements in the URIs, it is possible to identify potential dependencies among resources. EvoMaster requires API schemas specified with OpenAPI.[7] Therefore, we identified possible components of the OpenAPI specification for inferring possible dependencies. The four components in the OpenAPI specification are: *path*, *parameter* (including *defined object type* of the parameter and *attribute* of the defined object type), *operation description*, and *operation summary*.

A *path* defines an endpoint, and it is composed of a sequence of tokens separated by / and { } symbols. The tokens between { and } are recognized as path parameters. For example, both */products/foo* and */products/42* would match the same endpoint path */products/*{*productName*}. Since there might exist multiple resources in a *path* to represent their hierarchical structure separated by /, to represent this resource, we take the last non-parameter token as a *representative token* for the resource. For instance, */products/*{*productName*} can be decomposed into two tokens, *products* and *productName*, and the text *products* can be used to represent the resource. Given a *representative token* x for a resource A, then to check if another resource B has a dependency on A, we analyze if B has any reference to x in its schema definition (e.g., a query parameter name, or text description).

For *parameter*, *defined object type* and its *attribute*, we keep their names as tokens. Besides, we consider some commonly used words (e.g., *id*, *name* and *value*) relevant for deriving alternative possibilities for the identifying tokens. For example, an alternative token for *warehouseName* query parameter is simply *warehouse*, as the token ends with the word *name* (ignoring the case). When matching dependent resources, both alternative tokens are used.

Regarding *operation description* and *operation summary*, since they are free-text sentences, we employed *Stanford Natural Language Parser*[8] (SNLP) to analyze them for identifying *noun*s as tokens. In addition, with SNLP we may get base forms (i.e., lemmas) of tokens analyzed from parameters and path elements, e.g., *products* has *product* as its lemma. Therefore each of the tokens is designed with a *lemma* property that is used as an alternative for token matching.

For a resource, one representative token and/or more related tokens can be parsed with such pre-processing on the four components of its resource path and its available operations. To infer possible related resources, we match tokens of the resource with the representative tokens of other resources using the Trigram Algorithm (Martin et al. 1998) to calculate a degree of similarity between any token of the resource and other representative tokens. In our current implementation, if the degree of at least one token satisfies our specified requirement (i.e., >0.6), we create a possible undirected dependency between the two resources. The probability of the derived dependency is initialized based on that degree.

6.1.2 Accessed SQL Tables

RESTful APIs are supposed to be stateless. This helps with horizontal scalability (e.g., a service can easily be replicated in several running instances on different processes, as there is no internal state to keep in sync), and to avoid issues with the restarting of the processes. This means that the state is usually handled externally, typically in a database, such as *Postgres*, *MySQL* and *MongoDB*. In these cases, the *resources* handled by the API are an abstraction of the data in such databases.

EvoMaster does analyze all the interactions of the SUT with SQL databases (Arcuri and Galeotti 2019; 2020) (for NoSQL ones such as *MongoDB*, it is work-in-progess), as it computes heuristics based on the results of such operations (e.g., to create the right input data for which *SELECT* operations return non-empty sets by satisfying their *WHERE* clauses). By exploiting this current feature in EvoMaster, we can record the information about all accessed tables in the databases for each executed HTTP call made in the tests.

Theoretically, if different API endpoints operate on the same tables, then there exist some relationships among them. During the search, we keep a global track in all test cases of all SQL tables accessed by actions on resources, at each single fitness evaluation. Thus, after each evaluation, dependencies among resources can be newly derived or updated if actions on different resources accessed the same tables in the database. Note that, if a dependency is newly derived (i.e., not currently existing in the graph), the added dependency in the graph is undirected. The probability of the dependency is then initialized/updated with the maximum 1.0.

6.1.3 Fitness Feedback

Dependency handling is developed for achieving a proper combination of dependent resources in a test. Besides heuristics on name matchings and database accesses, we also consider the feedback on the test fitness (e.g., line coverage), by analyzing its

[8]https://nlp.stanford.edu/

variations when different resources are manipulated. With resource-based structure mutation (recall Section 5.3), the combination of resources can be manipulated. Thus, by analyzing changes of fitness after a mutation operation, the dependency of resources in the test can be identified.

Assume that a test $T = (R_1, ..., R_n)$ is a sequence of R_k $(1 \leq k \leq n)$, and each R_k represents a sequence of actions (i.e., HTTP calls following one of the templates in Table 1) on one resource. Table 3 represents a set of heuristics to identify dependencies based on different changes of fitness for each of resource-based structure mutation operators. For each of the operators:

ADD a resource R_x at index i: fitness of all resources after i index is required to be examined. For each resource R_m $(i \leq m \leq n)$, if there exists any change, a dependency (i.e., R_m depends on R_x) can be identified, and its probability will be updated with 1.0. Note that the update of the probability with $update(DD_{R_m \rightarrow R_x}, p)$ is to take a larger value between the probability and p. For example (also represented in *add* row and *Example* column in Table 3), a test is a sequence of resource handling actions denoted as (A, B, C, D, E), e.g., A can be regarded as a sequence of actions on the A resource. With *add* mutation operator, a resource handling action on the resource F is added to the test at index 2, then the mutated individual becomes (A, B, F, C, D, E). For instance, if the fitness values contributed by actions in the C resource are changed in the mutated individual, i.e., with the newly involved *actions* on the F resource. This might imply that there exists a dependency, i.e., $C \rightarrow F$.

DELETE/MODIFY a resource R_i at index i: fitness of all resources after i index is required to be examined. For each of the resource R_m $(i < m \leq n)$, if there exist any change, a dependency (i.e., R_m depends on R_i) can be identified with a probability of 1.0.

REPLACE a resource R_i with R_x at index i: fitness of all resources after i index is required to be examined. For each of the resource R_m $(i < m \leq n)$, if the fitness becomes better, it means that R_m depends on R_x. If the fitness becomes worse, it means that R_m depends on R_i.

SWAP a resource R_i and a resource R_j $(1 \leq i < j \leq n)$: first, fitness of R_j is required to be examined, if it is changed, it means that R_j might depend on any resource between i and j (j is excluded). In addition, the fitness of all resources between i and j is required to be examined. For each of the resource R_m $(i < m < n)$, if the fitness becomes better, it means that R_m depends on R_j. If the fitness becomes worse, it means that R_m depends on R_i. Moreover, fitness of R_i is required to be examined, if it is changed, it means that R_i might depend on any resource between i (i is excluded) and j.

6.1.4 Summarize the Resource Dependency Detection

If the pre-match processing based on REST API Schema (Section 6.1.1) is enabled, the search will start with a set of undirected derived dependencies. Over the course of the search, the graph of dependencies will be evolved by adding new derived dependencies, updating directions of undirected derived dependencies, and updating probabilities of the derived dependencies. After each fitness evaluation, by further identifying dependencies based on *Accessed SQL Tables* (Section 6.1.2), the graph will be expanded and the probabilities of the dependencies might be updated to 1.0. Besides, by analyzing the changes in fitness scores (Section 6.1.3), newly directed dependencies can be further detected and the direction of the undirected derived dependencies might be updated.

Table 3 Heuristic to identify dependency among resources with resource-based structure mutation

Mutation operator	Heuristic$^{(1)}$	After/Before Fitness$^{(2)}$		Example, $T = (A, B, C, D, E)$	
Add	add R_x at index i $(1 \leq i \leq n)$			add F at index 2, i.e., $T' = (A, B, F, C, D, E)$ $R_m \in \{C, D, E\}$	
	$\forall R_m \in \{R_s \| i \leq s \leq n\}$	**IF**	$F_a(R_m) \neq F_b(R_m)$	**IF**	$F_a(C) \neq F_b(C)$
	$\rightarrow R_x$		update($DD_{R_m \rightarrow R_x}$, 1.0)		update($DD_{C \rightarrow F}$, 1.0),
		ELSE	update($DD_{R_m \rightarrow R_x}$, 0)	**ELSE**	update($DD_{C \rightarrow F}$, 0)
Delete	delete R_i $(1 \leq i \leq n)$			delete C, i.e., $T' = (A, B, D, E)$ $R_m \in \{D, E\}$	
	$\forall R_m \in \{R_s \| i < s \leq n\}$	**IF**	$F_a(R_m) \neq F_b(R_m)$	**IF**	$F_a(D) \neq F_b(D)$
	$\rightarrow R_i$		update($DD_{R_m \rightarrow R_i}$, 1.0)		update($DD_{D \rightarrow C}$, 1.0),
		ELSE	update($DD_{R_m \rightarrow R_i}$, 0)	**ELSE**	update($DD_{D \rightarrow C}$, 0)
Modify	modify R_i $(1 \leq i \leq n)$			modify C i.e., $T' = (A, B, C, D, E)$ $R_m \in \{D, E\}$	
	$\forall R_m \in \{R_s \| i < s \leq n\}$	**IF**	$F_a(R_m) \neq F_b(R_m)$	**IF**	$F_a(D) \neq F_b(D)$
	$\rightarrow R_i$		update($DD_{R_m \rightarrow R_i}$, 1.0)		update($DD_{D \rightarrow C}$, 1.0),
		ELSE	update($DD_{R_m \rightarrow R_i}$, 0)	**ELSE**	update($DD_{D \rightarrow C}$, 0)
Replace	replace R_i with R_x $(1 \leq i \leq n)$			replace C with F, i.e., $T' = (A, B, F, D, E)$ $R_m \in \{D, E\}$	
	$\forall R_m \in \{R_s \| i < s \leq n\}$	**IF**	$F_a(R_m) < F_b(R_m)$	**IF**	$F_a(D) < F_b(D)$
	$\rightarrow R_i, R_x$		update($DD_{R_m \rightarrow R_i}$, 1.0)		update($DD_{D \rightarrow C}$, 1.0)
		EIF	$F_a(R_m) > F_b(R_m)$	**EIF**	$F_a(D) > F_b(D)$

Table 3 (continued)

Mutation operator	Heuristic$^{(1)}$	After/Before Fitness$^{(2)}$		Example, $T = (A, B, C, D, E)$	
			update($DD_{R_m \to R_x}$, 1.0)		update($DD_{D \to F}$, 1.0)
		ELSE	update($DD_{R_m \to R_i}$, 0)	**ELSE**	update($DD_{D \to C}$, 0)
			update($DD_{R_m \to R_x}$, 0)		update($DD_{D \to F}$, 0)
Swap	swap R_i and R_j $(1 \le i < j \le n)$			swap B and D, i.e., $T' = (A, D, C, B, E)$ $R_m \in \{C\}$, $R_{im} \in \{B, C\}$, $R_{mj} \in \{C, D\}$	
	R_j	**IF**	$F_a(R_j) \neq F_b(R_j)$	**IF**	$F_a(D) \neq F_b(D)$
	$\to \forall R_{im} \in \{R_s \vert i \le s < j\}$		update($DD_{R_j \to R_{im}}$, $\frac{1.0}{j-i-1}$)		update($DD_{D \to B}$, 0.5), update($DD_{D \to C}$, 0.5)
		ELSE	update($DD_{R_j \to R_{im}}$, 0)	**ELSE**	update($DD_{D \to B}$, 0), update($DD_{D \to C}$, 0.0)
	$\forall R_m \in \{R_s \vert i < s < j\}$	**IF**	$F_a(R_m) < F_b(R_m)$	**IF**	$F_a(C) < F_b(C)$
	$\to R_i, R_j$		update($DD_{R_m \to R_i}$, 1.0)		update($DD_{C \to B}$, 1.0)
		EIF	$F_a(R_m) > F_b(R_m)$	**EIF**	$F_a(C) > F_b(C)$
			update($DD_{R_m \to R_j}$, 1.0)		update($DD_{C \to D}$, 1.0)
		ELSE	update($DD_{R_m \to R_i}$, 0)	**ELSE**	update($DD_{C \to B}$, 0)
			update($DD_{R_m \to R_j}$, 0)		update($DD_{C \to D}$, 0)
	R_i	**IF**	$F_a(R_i) \neq F_b(R_i)$	**IF**	$F_a(B) \neq F_b(B)$
	$\to \forall R_{mj} \in \{R_s \vert i < s \le j\}$		update($DD_{R_i \to R_{mj}}$, $\frac{1.0}{j-i-1}$)		update($DD_{B \to D}$, 0.5), update($DD_{B \to C}$, 0.5)
		ELSE	update($DD_{R_i \to R_{mj}}$, 0)	**ELSE**	update($DD_{B \to D}$, 0), update($DD_{B \to C}$, 0.0)

Note that $^{(1)}$ R_i is a sequence of actions (with the template) on one resource where i is an index (before the test is mutated) of the grouped actions on the resource. $^{(2)}$ F_a presents a fitness after the mutation, F_b presents a fitness before the mutation, $F_a > F_b$ indicates new targets are covered or any fitness of target is improved after the mutation, $F_a < F_b$ indicates fitness of any target is decreased after the mutation, $F_a \neq F_b$ indicates covered targets or their fitness are changed after the mutation, *update*($DD_{R_m \to R_x}$, p) is to update a probability of a derived dependency ($DD_{R_m \to R_x}$ means R_m depends on R_x) with p, and the *update* is implemented to take a larger value of the probability

6.2 Smart Sampling with Dependency

Information on derived dependencies can be exploited by sampling methods involving multiple resources, i.e., *S2dR* and *SMdR*, as smart sampling with dependencies. Enabling such smart sampling with dependencies is controlled with a probability, i.e., P_d, as described in Section 4. When enabled, for *S2dR*, a test is sampled with two resources which are linked with a derived dependency (if any exist). If the dependency is undirected, the order of handling the two resources will be determined randomly. Otherwise, the test starts with handling the dependent resource first followed by the other. *SMdR* does sample a test with more than two resources, and, in the test, there exists at least one derived dependency if the set of dependencies is not empty (i.e., there exists at least one dependency whose probability is more than 0.0). In case of no derived dependencies, then *SMdR* simply chooses resources at random.

As discussed in Section 5.1, values of parameters of actions within a template will be *bounded* for handling one specific resource. For example, all the path variables in a series of actions on a resource will have the same values (to guarantee to work on the same resource), and a mutation operation that modifies any of them will propagate its changes to all the other actions on such resource. To make a dependent resource available to the following related resources, it is also required to bind values of parameters across resources according to their dependencies.

Recall the example in Fig. 6, a test is sampled by *S2dR* that is composed of an action (at line 1) for handling a *warehouse* resource and two actions (at lines 2-3) for handling a *product* resource, and parameters of the actions at lines 2-3 are bounded for handing same resource, i.e., a **foo** product. If the dependency (i.e., *product* depends on *warehouse*) is derived, when processing actions on *product*, we also need to consider a **bar** *warehouse* in order to make a **foo** *product* related to the **bar** *warehouse*, i.e., the *warehouse* parameter of POST */products/*{*productName*} is bounded with the **bar** *warehouse* performed by the action at line 1 as shown in Fig. 8.

6.3 Smart Mutation with Dependency

As discussed in Section 5.3, one aim of resource-based structure mutation is to exploit relationships among resources. Thus, we enhance the mutation with a consideration of derived dependencies to boost its performance. Regarding each of the mutation operators:

DELETE. Delete a resource that is marked as unrelated to any other resource in the test.

SWAP. We propose three strategies to swap two resources with dependency relations: (1) *adjust* the order of a resource and its likely dependent resource, i.e., if there exist a derived dependency with a high probability (i.e., ≥ 0.6) in the test, but the order is incorrect according to the dependency relation, then we apply the strategy to swap their order in the test; (2) *attempt* to adjust the order of a resource and its possibly dependent resource, i.e., if

```
1 POST /warehouses/bar?capability=40
```

actions with **possibly-dependent** #5 on */warehouses*

```
2 POST /products/foo?warehouse=bar&
      quantity=20
3 GET /products/foo
```

actions with **possibly-dependent** #7 on */products/*{*productName*}

Fig. 8 An example (Example 9) of a test sampled by S2dR method with dependency

there exist a derived dependency with a modest probability (i.e., < 0.6 but still > 0.0) in the test, but the order is incorrect according the dependency, then we applied the strategy to fix their order; (3) *explore* new order of two resources. Similarly to *adjust* and *attempt* but *apply* on two resources with no dependency information, to evaluate a possible new dependency. Based on a set of derived dependencies, the applicable strategies are selected at random. If none of the strategies can be applied, we select two positions randomly as *SWAP* without dependency handling.

ADD. Add a resource that is related to one of the resources in the test based on the inferred dependencies. If none of the resources is related to any other resource, add a new resource as *ADD* without dependency handling.

REPLACE. Replace is handled as a combination of *DELETE* and *ADD* with a fixed position, i.e., remove an unrelated resource at a specific position, and then add a new related resource at that same position.

MODIFY. No further treatment with a consideration of the dependency graph.

Note that, if a resource is mutated with the above operators based on derived dependencies, any other resource bounded to it will need to be updated (e.g., query parameters and fields in body payloads) as well to maintain the dependency.

7 Case Studies

7.1 Open Source Case Studies

In our empirical study, we employed seven RESTful APIs (available on GitHub[4]) which we used in our previous work (Arcuri 2018b; 2019). These APIs consist of three artificial and four real-world web services. Table 4 shows detailed descriptive statistics on these web services, including their number of Java/Kotlin class files, lines of code, number of endpoints (where with the term *endpoint* we mean the combination of resource paths and HTTP methods applicable on them, ignoring any query parameter), and number of accessible resources (ignoring HTTP verbs) with the number of independent resources among them.

Regarding these services, REST Numerical Case Study (*rest-ncs*) and REST String Case Study (*rest-scs*) are artificial APIs based on functions that were previously used in unit testing for experiments on solving numerical (Arcuri and Briand 2011) and string (Alshraideh and Bottaci 2006) problems. These APIs simply put each of these stateless functions behind a different GET endpoint. *rest-news* is also an artificial API, which was developed for educational purposes on enterprise development in a university course of one of the authors.[9] The APIs *features-service*, *proxyprint*, *scout-api* and *catwatch* are real RESTful web service projects, which were selected by analyzing projects on the popular open-source repository GitHub. More details of the selection can be found in Arcuri (2018b) and Arcuri (2019).

Note: in this article we use the term "*real*" (for 4 SUTs downloaded from GitHub) just as opposed to, and differentiate from, the terms "*artificial*" (used for 3 SUTs) and "*synthetic*" (used for 12 SUTs). We do not claim that these systems are widely used in industry, or that they are representative of industrial systems in general. Furthermore, in this article we use the term "*artificial*" to represent the SUTs that were written by hand just for experiment purposes (e.g., based on software used in the evaluation of unit test generation techniques) or didactic reasons, in contrast to the "*synthetic*" SUTs that were automatically generated

[9] https://github.com/arcuri82/testing_security_development_enterprise_systems

Table 4 Descriptive statistics of the open source case studies

Name	#Classes	LOCs	#Endpoints	#Resource	#Independent resource
rest-ncs	9	602	6	6	6
rest-scs	13	859	11	11	11
rest-news	10	718	7	4	1
catwatch	69	5442	23	13	11
feature-service	23	2347	18	11	1
proxyprint	68	7534	74	56	26
scout-api	75	7479	49	21	2

with a tool. But other terminologies could have been used to distinguish those groups, like Group A, B and C.

7.2 Automatically Generated Synthetic RESTful APIs

To achieve sound results in an empirical study, a large and varied selection of SUTs is required (Fraser and Arcuri 2012). However, system testing is very time consuming. Furthermore, although open-source repositories such as GitHub do host plenty of software projects (especially libraries), enterprise applications are not so common among them. This poses challenges in carrying out empirical studies in this problem domain.

To address this problem, we integrated our empirical study with a set of synthetically generated APIs, with different characteristics when it comes to intra-resource dependencies. This also enables us to clearly identify how our techniques perform, i.e., pinpoint the conditions in which they perform well or struggle. However, whether real APIs would have the same characteristics remains to be seen. This is why it is important to still do empirical studies on real APIs and not just synthetic ones. In other words, the experiments on these synthetic APIs are only done to provide more insight, and possibly explain differences in performance among the real APIs.

To experiment the proposed techniques with various RESTful web applications in terms of resource and their dependency, we implemented a synthetic REST API generator (Fig. 9) for automatically producing such applications. In Fig. 9, we propose a model (i.e., *Synthetic REST API Graph*) that is composed of elements (denoted as white boxes) for defining the application with respect to resources and their dependencies. With such a model, a RESTful web service can be automatically generated with elements (denoted as grey boxes). Note that the elements with grey boxes are for representing a mapping between the model and an instantiation that could be carried out with any available tool/framework. In our work, we used SpringBoot[10] JPA[11] and automated inference of OpenAPI schemas[7] from source code.[12]

In our implementation, a Synthetic RESTful API is defined with a set of resources (*ResourceClass*), which can be connected with dependencies (*Dependency*).

ResourceClass represents a type of the resource, and its instances can be considered as an actual resource. For example, *GET /products/foo* can be regarded as "to retrieve an instance

[10]https://spring.io/projects/spring-boot

[11]https://spring.io/projects/spring-data-jpa

[12]https://github.com/springdoc/springdoc-openapi

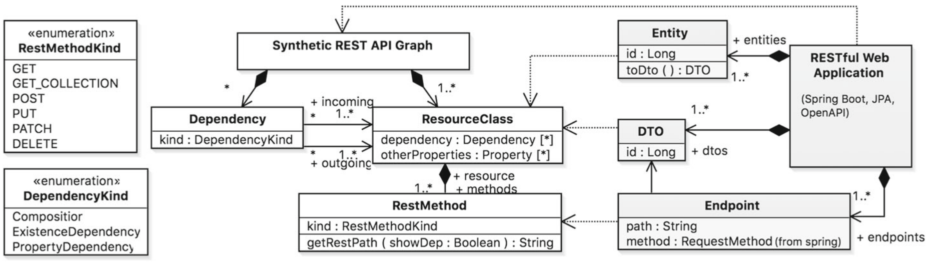


Fig. 9 Synthetic REST API Design and Realization

of *product*, and the identifying name of the instance should be *foo*". Based on the specified *ResourceClass*, we generate an *Entity* (for resource persistence and access in the database) and its corresponding *Data Transfer Object* (*DTO*) (when the resource needs to marshaled into a JSON object representation for resource transfer through a network).

Dependency is designed to describe "one or more resources depend on one or more of the other resources". The dependencies among resources are varied. To generate RESTful APIs with various dependencies, we designed three kinds of dependency, i.e., *Composition*, *ExistenceDependency*, and *PropertyDependency*. These kinds are also designed with different level of complexity in *dependency handling*, i.e., *Composition* is the easiest one, and *PropertyDependency* is the most challenging one. Definitions of each of these kinds are presented in Table 5, together with the corresponding constraints on resources and their dependent resources.

Moreover, we introduced *RestMethod* for specifying *Endpoint* generations for a RESTful API. The *RestMethod* is associated with a *RestMethodKind*, which is composed of all the different types of HTTP methods (e.g., GET and POST), for specifying which HTTP methods should be provided to access the related resources. As dependencies among resources might be represented in the resource path, we proposed two strategies to generate resource paths for each *Endpoint*, i.e., showing dependency and hidden dependency. Table 6 presents our strategies to generate endpoints based on different HTTP methods following REST guidelines.

In our context, in terms of an endpoint, a successful status code (i.e., 2xx) should be given only if all related dependency constraints are satisfied and an action (according to specified HTTP method) is performed properly. If any constraint is not satisfied, execution of the endpoint will exit and return a 4xx status code indicating that there are user errors in

Table 5 Definitions and Constraints of *Dependency* in the Synthetic REST APIs

#	Dependency	Definitions	Constraints
DR1	Composition	A resource is composed of resource(s).	Owned resources cannot exist independently of the resources.
DR2	ExistenceDependency	Resource(s) depends on existing resource(s).	Dependent resources must exist before performing any action on the resources.
DR3	PropertyDependency	Resource(s) depends on existing resource(s) with additionally some constraints on their property(ies).	Dependent resources must exist, and properties of the dependent resources must satisfy some specified constraints.

Table 6 Implementation of Generating *Endpoints* with *RestMethod*

Method	Description	Constraint$^{(1)}$ and status code	Path$^{(2)}$
GET COLLECTION	get all resources	return with 2xx	**showing dependency**: /DRCs/{rid}/RCs, e.g., /warehouses/{warehouseName}/products hidden dependency: /RCs e.g., /products

POST	create a resource	**IF** R, return with 4xx
		IF not DR1, return with 4xx
		IF not DR2, return with 4xx
		IF not DR3, return with 4xx
		return with 2xx	

PUT	create or update a resource with its id	**IF not** DR1, return with 4xx
		IF not DR2, return with 4xx
		IF not DR3, return with 4xx
		return with 2xx
			showing dependency: /DRCs/{rid}/RCs/{id} e.g., warehouses/{warehouseName}/products/{id} **hidden dependency**: /RCs/{id} e.g., /products/{id}

PATCH	partially update a resource with its id	**IF not** R, return with 4xx
		IF not DR2, return with 4xx
		IF not DR3, return with 4xx
		return with 2xx	

DELETE	delete a resource with its id	**IF not** R, return with 4xx
		return with 2xx	

GET	get a resource with its id	**IF not** R, return with 4xx
		return with 2xx	

Note that $^{(1)}$ by *Constraints* column: R is a condition that the requested resource exists; DR1-3 indicate the corresponding constraint in Table 5 is satisfied; **not** indicates the condition or constraint is not satisfied. $^{(2)}$ by *Path* column: DRC represents dependent resources; {*rid*} represents an identifier of DRC; RC represents a resource to be performed by HTTP methods; {*id*} represents an identifier of RC

this HTTP request. The order and type of the dependency constraint checking is based on the complexity in the dependency handling setting, i.e., from easy to challenging. Thus, a test with proper data to handle dependencies can cover more code (as all those constraint checks are `if` statements in sequence, checking one condition at a time). Note that, if all of the constraints are satisfied but the action is still performed improperly (e.g., an exception is thrown for some reason), then the execution will exit and a server error status code (i.e., 500) will be returned (this is handled by default in Spring).

To generate the applications with various resource-dependency settings, we designed three resource graph settings, i.e., *Dense-Central*, *Medium-Deep* and *Sparse-Straight*, as shown in Fig. 10 (the names of the resources are generated at random). Note that all of the three settings consist of 5 *ResourceClass*es with 6 methods. Thus, there are 30 endpoints for all of the settings.

Regarding *Dense-Central*, there are 4 dependencies connecting all 5 resource classes through *UEear*. Regarding *Medium-Deep*, 3 dependencies connect 4 out of the 5 resource classes, but there exist a deep chained dependency from *VIL0S* to *U1rA1* through *HErqD* and *XpOCt*. Regarding *Sparse-Straight*, 2 out of the 5 resources classes are connected with 1 dependency. Note that in Fig. 10 *<is composed of>* can only be specified with *DR1*, but *<depends on>* can be specified with either *DR2* or *DR3*, which result into easy or complex

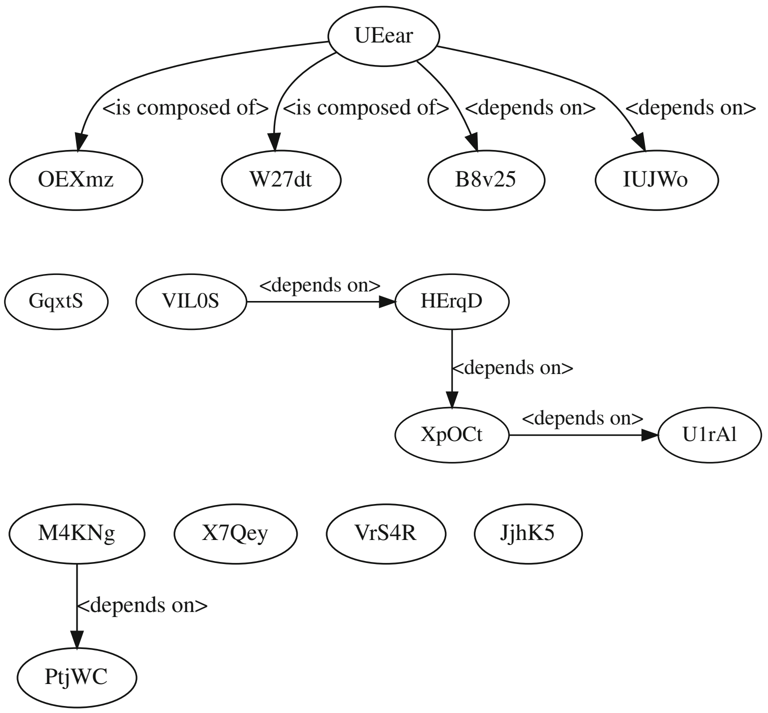


Fig. 10 Dependencies of Dense, *Medium and Sparse* settings. *Dense-Central* setting is defined with 4 one-to-one dependencies from $UEear$ that forms 1 two-to-two dependency relationship, i.e., $OEXmz$ and $W27dt$ depend on $B8v25$ and $IUJWo$. *Medium-Deep* setting is defined with 3 connected one-to-one dependency relationships. The deepest derived dependency is from $VIL0S$ to $U1rA1$ through $HErqD$ and $XpOCt$. *Sparse-Straight* setting is defined with 1 one-to-one dependency relationship

dependency constraints. To control for the difficulty in solving dependency constraints, we, therefore, generated APIs with different choices (i.e., *DR2* or *DR3*) on *<depends on>* in the three settings. Moreover, we configured resource paths generation with showing or hidden dependencies in paths, since this configuration might impact the handling of resources. Thus, in total, 12 (3 resource-dependency settings $\times 2$ kinds of dependency constraints $\times 2$ strategies of resource path generation) synthetic RESTful APIs are generated for our empirical study.

An example is shown in Fig. 11, illustrating different endpoint generations according to the different configurations. The endpoint in Fig. 11 is for the POST action on *UEear* resource shown in **Dense-Central** setting of Fig. 10. In the setting, *UEear* owns *OEXmz* and *W27dt*, and has two dependent resources, i.e., *B8v25* and *IUJwo*. Snippets 1-2 present different paths regarding showing and hidden dependency in paths, respectively. In Snippet 1, two dependent resources are shown in the path, i.e., */iUJWos/iUJWoId/b8v25s/b8v25Id/uEears*, while not in Snippet 2, i.e., */uEears*. Regarding an implementation of POST, when creating an instance of *UEear*, it is the first to check whether the instance exists, denoted as *R* check (see line 2 in Snippet 3). Once passing the *R* check, create owned resources (i.e., an instance of *OEXmz* and an instance of *W27dt*) if the resources do not belong to any others (*DR1* checks), and the code for processing *OEXmx* is at lines 6-13 in Snippet 3.

```
@RequestMapping(
value = "/iUJWos/{iUJWoId}/b8v25s/{
      b8v25Id}/uEears",
method = RequestMethod.POST,
consumes = MediaType.APPLICATION_JSON)
```

Snippet 1. showing dependency

```
@RequestMapping(
value = "/uEears",
method = RequestMethod.POST,
consumes = MediaType.APPLICATION_JSON)
```

Snippet 2. hidden dependency

```
1  // an entity with id uEear.id should not exist
2  if (uEearRepository.findById(uEear.id).isPresent()) return ResponseEntity.status
        (400).build(); //IF R
3  UEearEntity node = new UEearEntity();
4  node.setId(uEear.id);
5  // create owned entity
6  OEXmz ownedDto0 = new OEXmz();
7  ownedDto0.value = uEear.oEXmzValue;
8  ownedDto0.id = uEear.oEXmzId;
9  int ownedDto0Code = oEXmzRestAPI.createOEXmz(ownedDto0).getStatusCode().value();
10 if (!oEXmzRepository.findById(uEear.oEXmzId).isPresent())
11   return ResponseEntity.status(ownedDto0Code).build();
12 OEXmzEntity ownedEntityOEXmzEntity = oEXmzRepository.findById(uEear.oEXmzId).get();
13 node.setOwnedOEXmz(ownedEntityOEXmzEntity);
14 ...
```

Snippet 3. codes for DR1

```
15 // related entities should exist
16 if (!b8v25Repository.findById(uEear.
        b8v25Id).isPresent())
17   return ResponseEntity.status(404).
        build();
18 B8v25Entity referVarToB8v25Entity =
        b8v25Repository.findById(uEear.
        b8v25Id).get();
19 node.setB8v25(referVarToB8v25Entity);
20 if (!iUJWoRepository.findById(uEear.
        iUJWoId).isPresent())
21   return ResponseEntity.status(404).
        build();
22 IUJWoEntity referVarToIUJWoEntity =
        iUJWoRepository.findById(uEear.
        iUJWoId).get();
23 node.setIUJWo(referVarToIUJWoEntity);
24 node.setName(uEear.name);
25 node.setValue(uEear.value);
```

Snippet 4. codes for DR2 and DR3

```
26 // additional codes for DR3
27 if (!(Util.median(new double[] {
        ownedDto0.value * 1.0,
        ownedDto1.value * 1.0}) < Util.
        median(new double[] {
        referVarToB8v25Entity.getValue
        () * 1.0, referVarToIUJWoEntity
        .getValue() * 1.0})
28 && Util.average(new double[] {
        ownedDto0.value * 1.0,
        ownedDto1.value * 1.0}) < Util.
        average(new double[] {
        referVarToB8v25Entity.getValue
        () * 1.0, referVarToIUJWoEntity
        .getValue() * 1.0})))
29 return ResponseEntity.status(400).
        build();
```

Snippet 5. additional codes for DR3

```
30 uEearRepository.save(node);
31 return ResponseEntity.status(201).build();
```

Snippet 6. codes to return successful status code

Fig. 11 Snippet examples of implementations of POST on *UEear* (see **Dense-Central** in Fig. 10) with different configurations

As discussed, *<depends on>* can be applied with *DR2* or *DR3*. If *DR2* is applied, following by the creation of owned resources, we check the existence of dependent resources, as shown in Snippet 4, i.e., line 16 is for *B8v25* and line 20 is for *IUJwo*. If *DR3* is applied, there are not only the same checks as in Snippet 4, but also additional checks on *property condition* in Snippet 5. In this implementation, the *property condition* is designed based on *value* properties of the resource (denoted as RC) and its dependent resources (denoted as DRC_i) as

not $(median(SRC) < median(SDRC)$ **and** $average(SRC) < average(SDRC))$

where if RC is composed of more than one other resources (denoted as ORC_k), $SRC = \{ORC_k | k > 1\}$, otherwise $SRC = \{RC\}$; $SDRC = \{DRC_i | i > 0\}$; $median(S)$ is a median of values on *value* property of a set of resources S; $average(S)$ is an average of values on *value* property of a set of resources S. For instance, the additional check of *UEear* is shown at lines 27-28 in Snippet 5. Last, if all checks are passed, we save the created instance of *UEear* in the database and return 201 status code (Snippet 6).

Note that our synthetic REST API generator is not limited for evaluation just in this work. As we released it as open-source, it could be also useful to setup experiments for studying other RESTful APIs-related approaches. With our generator, a REST API can be automatically generated by configuring a set of parameters (e.g., a number of resource nodes, a number of dependencies, applied HTTP methods), or a specific resource dependency graph. Those configurable parameters capture different basic characteristics of a REST API, and this flexible configuration would be helpful to customize different synthetic case studies (e.g., the number of resources might be used to study the scalability of an approach). In addition, as the generator is open-source, that presents a possibility to be further customized by different researchers, e.g., extend *PropertyDependency* with different implementations. Moreover, the generator is designed by following the REST API guidelines, and the generated REST APIs come with a schema using OpenAPI/Swagger. Thus, the generator might offer an opportunity to assess OpenAPI-based approaches with synthetic case studies. However, currently the generator only supports the creation of REST APIs with SpringBoot and JPA. This may limit the study regarding implementations with different frameworks/libraries.

8 Empirical Study

In this paper, we have carried out an empirical study aimed at answering the following research questions.

RQ1: How does *resource-based MIO* perform? Among the different settings, which one gives the best results in terms of covered targets, line coverage and branch coverage?
RQ2: How does *dependency heuristic handling* work with resource-based MIO? Among the different settings, which one gives the best results?
RQ3: Do our novel techniques achieve any improvement compared to existing work? Among the different techniques we proposed, which one gives the best results?

8.1 Experiment Design

To assess our novel approach, we conducted an empirical study. The experiment settings and their design are as shown in Tables 7 and 8, respectively.

Table 7 reports the configurations of our experiments for each investigated technique. In the experiments, we selected two baselines: *Base1* is an implementation of default MIO (Arcuri 2018b); the other (i.e., *Base2*) is also based on MIO, but it was integrated with smart sampling techniques (that can be regarded as resource-based solutions) specialized for sampling test data for RESTful APIs (Arcuri 2019) (recall Section 2.3). With MIO integrated with smart sampling (i.e., *Base2*), we used its default setting on the probability of applying smart sampling at the sampling phase of MIO, i.e., $P_s = 0.5$. *Base2* is the current default technique in EvoMaster, where smart sampling was empirically evaluated to provide better results (Arcuri 2019). *Base1* is simply EvoMaster with the smart sampling deactivated. In this paper, we still compare with *Base1* to get a better insight on what results can be achieved compared with a more basic approach. Note that, in the past, we have compared MIO-based EvoMaster with other search algorithms, e.g., random search and MOSA (Arcuri 2018b). We do not repeat such comparisons here in this paper, and just use the current default version of EvoMaster as baseline, as that is the one that has given best results in our previous work.

For the proposed approaches, we classified them into two configurations, by distinguishing whether we enable or not the handling of dependencies (Sections 5 and 6). This is

Table 7 Description of experiment settings

Technique	Sampling		Mutation	Dependency		Count[2]
	Strategy	P_s		P_d	Pre-Match	
Base1	Random	0	Default	0.0	F	1
Base2	Smart Sampling	0.5	Default	0.0	F	1
R-MIO	R-Sampling[1]	{0.5, 1.0}	R-Mutation	0.0	F	10
Rd-MIO	R-Sampling[1]	{0.5, 1.0}	R-Mutation	{0.5, 1.0}	{F, T}	40

Note that [1] Resource-based Sampling, *R-Sampling*∈ {*EqualProbability, Actions, TimeBudgets, Archive, ConArchive*}; [2] Count represents a number of configurations for the technique

controlled by the P_d parameter (see Algorithm 2), i.e., Resource-based MIO (denoted as *R-MIO*) with $P_d = 0$, and Resource-based MIO with dependency handling (denoted as *Rd-MIO*) with $P_d > 0$.

Regarding the settings for sampling and mutation, all five sampling strategies ($S \in$ {*Action, Archive, ConArchive, Used-Budget, Equal*}), combined with the proposed resource-based mutation, are evaluated in the experiments with two different probabilities ($P_s \in \{0.5, 1.0\}$) of using the proposed sampling. For example, if $P_s = 0.5$, MIO applies our novel sampling to sample an individual with 50% probability, or applies random sampling otherwise.

These sampling and mutations settings for *R-MIO* and *Rd-MIO* are different. Because, when dependency handling is enabled, the derived dependencies might be utilized to guide the sampling and mutation (recall Section 6). Regarding the settings of dependency handling of *Rd-MIO*, we set two different probabilities on enabling dependency handling (i.e., $P_d \in \{0.5, 1.0\}$), combined with two different values on whether enabling the inference of possible dependencies based on the schema API (recall Section 6.1.1) (i.e., Pre-Match, $PM \in \{F, T\}$). For example, if $P_d = 0.5$, then at the sampling phase, when resource-based sampling is enabled with P_s, MIO applies resource-based sampling with dependencies to sample an individual with 50% probability (e.g., sample an individual with two resources that might be dependent), or applies resource-based sampling without dependency otherwise. During the mutation phase, MIO will then apply resource-based mutation with dependency handling to mutate an individual with 50% probability (e.g., switch actions of an individual based on the dependency of their resources), or applies resource-based sampling without dependency otherwise.

For each setting, we ran EvoMaster using the same fixed value for the search budget (i.e., 100,000 HTTP calls). All the other settings are left as their defaults in EvoMaster, like for example the population size (i.e., 10 per target), maximum length of a test (i.e., 10), probability of sampling (i.e., 0.5), and start of focused search (i.e., after 50% of the budget is used).

The design of the experiments is illustrated in Table 8. The table presents, for each research question, which settings are used, which tasks we performed, which case studies are used, how many times the experiments are repeated, which statistical tests are applied, and which metrics are used.

In these experiments, we selected seven open-source RESTful web services (three artificial RESTful APIs and four real RESTful APIs, recall Section 7.1) and generated twelve synthetic RESTful APIs covering various resources settings (recall Section 7.2).

Table 8 Description of experiment design regarding research questions

RQs	Conf.$^{(1)}$	Tasks	CS	Times	Statistical tests	Metrics
RQ1	R-MIO	- Analyze effectiveness of R-MIO - Identify the best from 10 settings	7$^{(2)}$	10	- Effect analysis of factors and interactions: Aligned Ranks Transformation ANOVA and Partial eta-squared effect size - Variance analysis: Friedman test - Pair comparsion:Mann-Whitney-Wilcoxon U-tests at a significant level $\alpha = 0.05$ and Vargha-Delaney effect sizes	#Targets %Lines %Branches
RQ2	Rd-MIO	- Analyze effectiveness of Rd-MIO - Identify the best from 40 settings				
RQ3	Base1 Base2 R-MIO Rd-MIO	- Compare best configurations of R-MIO and Rd-MIO with baselines - Analyze effectiveness of R-MIO and Rd-MIO regarding different case studies	7$^{(2)}$ + 12$^{(3)}$	30		

Note that $^{(1)}$ detailed configurations specified in **Conf.** can be found in Table 7; $^{(2)}$ 7 open source case studies; $^{(3)}$ 12 synthetic case studies

To take into account the randomness of the employed search algorithms, each settings of each technique should be repeated several times, and 30 times is a typically recommended value (Arcuri and Briand 2014). However, with 52 configurations and 19 case studies, it is impractical to run each of the configurations with a search budget of 100,000 HTTP calls on all case studies 30 times, i.e., $52 \times 19 \times 30 \times 100k = 2964M$ HTTP calls.

Therefore, we conducted our experiments to answer our RQs as follows:

– For RQ1 and RQ2, we executed all configurations of *R-MIO* and *Rd-MIO* 10 times, just on the 7 open-source RESTful APIs, to study the overall performance of the two techniques and identify their best settings.
– For RQ3, we applied the best configurations identified by RQ1 and RQ2 to represent *R-MIO* and *Rd-MIO*, respectively, and executed the two baseline techniques and the two identified configurations 30 times with all 19 (i.e., 7 open-source and 12 synthetic) case studies.

Experiment results were analyzed with following statistical tests: (1) Factorial data analysis is conducted with Aligned Ranks Transformation ANOVA (ART) and Partial eta-squared effect size (${\eta_p}^2$) (Wobbrock et al. 2011; Kay and Wobbrock 2019). In these experiments, the configured parameters (e.g., Sampling Strategy *R-Sampling*) can be regarded as factors, then we applied the test to study effects of parameters or interaction to response value; (2) Variance analysis is performed with Friedman test, e.g., variance analysis on ranks of different settings. But the Friedman test might be inadequate if there exist multiple factors (Wobbrock et al. 2011). In this case, ART can be conducted first to reduce the number of factors; (3) Pair comparisons are made with Mann-Whitney-Wilcoxon U-test at a significant level $\alpha = 0.05$ and Vargha-Delaney effect size.

In the context of white-box testing, we considered three metrics as response values for the experiments, i.e., a number of covered targets (#Targets), line coverage (%Lines) and branch coverage (%Branches), to evaluate the effectiveness of the tests generated by the different techniques. #Targets is the default coverage criterion that EvoMaster optimizes for by default. It is the aggregated value of all the other coverage metrics, including as well test targets related to the HTTP status codes for each different endpoint (e.g., status codes such as 500 can be used to detect potential faults). Note that, in the analyses, we mainly focus on line coverage (i.e., %Line), since it is typically the most used metric to evaluate test cases in practice. Branch coverage (i.e., %Branches) and covered targets (#Targets) are reported to provide additional insight on the results.

8.2 Experiment Results

8.2.1 Results of RQ1 (Resource-based MIO)

In Table 9, we report the effectiveness of *R-MIO*, measured by the average number of covered targets, line coverage and branch coverage, of the tests generated by each of the 10 settings of *R-MIO*, when run on the 7 open-source APIs. From the table, we can see that the tests generated by *R-MIO* are capable of covering up to 87.3% of lines and 66.2% of branches for the artificial REST APIs, i.e., *rest-ncs*, *rest-scs* and *rest-news*. For the other RESTful APIs, *R-MIO* achieves up to 53.4% line coverage and 21.0% branch coverage.

R-MIO is configured by two main parameters, i.e., *R-Sampling* with five resource-based sampling strategies, and P_s with two probabilities of applying resource-based sampling. To investigate the best configuration among all of these settings, we first conducted an effect analysis of the parameters, and their interactions, with Aligned Ranks Transformation

Table 9 Average, minimum and maximum values (represented as Avg.[min., max.]) of #Targets, %Lines, and %Branches covered by tests generated by all 10 settings of R-MIO

SUT	#Targets	%Lines	%Branches
rest-ncs	535.5[525.0,543.0]	87.3%[85.0%,88.1%]	66.2%[64.7%,68.1%]
rest-scs	855.2[854.0,860.0]	82.1%[82.0%,82.4%]	51.4%[51.3%,51.7%]
catwatch	1015.9[982.0,1163.0]	26.4%[25.7%,30.5%]	14.5%[13.5%,16.2%]
features-service	596.4[585.0,662.0]	53.4%[52.8%,59.2%]	12.8%[12.7%,16.9%]
proxyprint	1296.2[1280.0,1349.0]	18.7%[18.5%,19.4%]	5.4%[5.3%,6.2%]
rest-news	265.6[250.0,276.0]	40.5%[39.1%,42.4%]	23.3%[20.8%,25.4%]
scout-api	1794.3[1603.0,2141.0]	38.1%[33.8%,45.0%]	21.0%[19.1%,27.3%]

Table 10 Average, minimum and maximum values (represented as Avg.[min., max.]) of #Targets, %Lines, and %Branches covered by tests generated by all 40 settings of Rd-MIO

SUT	#Targets	%Lines	%Branches
rest-ncs	535.5[525.0,543.0]	87.3%[85.0%,88.1%]	66.2%[64.7%,68.1%]
rest-scs	855.2[854.0,860.0]	82.1%[82.0%,82.4%]	51.4%[51.3%,51.7%]
catwatch	1014.1[982.0,1159.0]	26.3%[25.7%,30.4%]	14.5%[13.5%,16.2%]
features-service	701.7[662.0,721.0]	64.2%[59.6%,65.7%]	18.9%[16.9%,21.2%]
proxyprint	1293.9[1279.0,1351.0]	18.6%[18.5%,19.4%]	5.4%[5.3%,6.2%]
rest-news	266.4[252.0,278.0]	40.7%[39.1%,42.4%]	23.2%[20.8%,25.4%]
scout-api	1765.5[1587.0,2145.0]	37.7%[33.6%,45.5%]	21.0%[18.9%,30.1%]

Table 11 Pair comparison of $PreMatch \in \{T, F\}$ using Mann-Whitney-Wilcoxon U-tests (p-value) and Vargha-Delaney effect sizes ($\hat{A}_{12}$)

SUT	PreMatch	#Targets		%Lines		%Branches	
		$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
catwatch	T vs. F	0.47	0.144	0.49	0.593	0.47	0.105
features-service	T vs. F	0.52	0.311	0.51	0.758	0.47	0.228
proxyprint	T vs. F	0.46	0.077	0.47	0.128	0.47	0.151
rest-news	T vs. F	**0.58**	**<0.001**	**0.63**	**<0.001**	0.48	0.411
scout-api	T vs. F	0.53	0.155	**0.56**	**0.010**	0.50	0.871

Values in bold means T is statistical significant better than F, whereas values in red means F is statistical significant better than T

Table 12 Pair comparison of $P_d \in \{0.5, 1.0\}$ using Mann-Whitney-Wilcoxon U-tests (p-value) and Vargha-Delaney effect sizes ($\hat{A}_{12}$)

SUT	P_d	#Targets		%Lines		%Branches	
		$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
catwatch	1 vs. 0.5	0.51	0.815	0.49	0.729	0.53	0.371
features-service	1 vs. 0.5	**0.69**	**<0.001**	**0.67**	**<0.001**	**0.60**	**0.003**
proxyprint	1 vs. 0.5	0.51	0.647	0.50	0.992	0.50	0.884
rest-news	1 vs. 0.5	0.46	0.230	0.47	0.337	0.47	0.476
scout-api	1 vs. 0.5	0.57	0.069	0.56	0.107	**0.57**	**0.049**

Values in bold means $P_d = 1.0$ is statistical significant better than $P_d = 0.5$, whereas values in red means $P_d = 0.5$ is statistical significant better than $P_d = 1.0$

ANOVA and Partial eta-squared ${\eta_p}^2$. Results are reported in Table 17, in the Appendix. From that table, we can see that different configurations of *R-Sampling* have a greater effect than P_s and their interaction on all three response values for most cases. The exceptions are *feature-service* in terms of %Lines, and *rest-news* with *scout-api* in terms of %Branches.

Based on this observation, we then conducted the Friedman test on ranks of average of three metrics among all case studies to identify the best setting on *R-Sampling* for *R-MIO*. Average of ranks among case studies and results of Friedman test are reported in Table 18, in Appendix. The results are statistically significant in terms of #Targets, but not %Lines and %Branches. When considering the average ranks, *ConArchive* achieves the best performance on #Targets and %Lines, and the second best on %Branches. Therefore, *ConArchive* is selected as the default setting on *R-Sampling* for *R-MIO*.

We further studied the P_s parameter to configure a probability of applying *ConArchive* at the sampling phase. We applied the Friedman test on ranks, and results are reported in Table 19, in Appendix. The results show that there is no statistically significant difference between the two settings (i.e, 0.5 and 1.0) regarding all three metrics, and the average ranks are close as well. Therefore, a reasonable choice is to set $P_s = 0.5$ as the default setting for P_s in *R-MIO*.

According to above results, we can conclude that:

RQ1: Resource-based MIO (i.e., MIO enhanced with resource-based technique) is capable of automatically generating tests that cover up to 53.4% lines in real REST APIs and 87.3% lines in artificial REST APIs. Our recommended configuration for applying *R-MIO* is ConArchive strategy with a 50% probability.

8.2.2 Results of RQ2 (Resource-Based MIO with Dependency Heuristic Handling)

The overall performance of *Rd-MIO* is reported in Table 10. In that table, for each case study, we report the average number of covered targets, lines coverage and branches coverage, of the tests generated by each of the 40 settings of *Rd-MIO*. Since all resources in *rest-ncs* and *rest-scs* are identified as ***independent***, there is no benefit to handle resource dependency. Thus, *Rd-MIO* and *R-MIO* perform in a similar manner on the two case studies. This leads to similar results with Table 9 for *rest-ncs* and *rest-scs*. Regarding the other RESTful APIs, tests generated by *Rd-MIO* achieve up to 64.2% line coverage and 23.2% branch coverage.

Table 13 Average of #Targets, %Lines and %Branches covered by tests generated by four techniques and their rank

SUT	Techniques	#Targets	%Lines	%Branches
rest-ncs	Base1	531.0(4)	86.4%(4)	65.7%(4)
	Base2	535.9(2)	87.3%(2)	**66.5%(1)**
	R-MIO	**536.2(1)**	**87.5%(1)**	66.3%(2)
	Rd-MIO	534.8(3)	87.2%(3)	66.1%(3)
rest-scs	Base1	591.2(4)	58.8%(4)	35.2%(4)
	Base2	687.7(3)	68.0%(3)	41.6%(3)
	R-MIO	**855.3(1)**	**82.1%(1)**	**51.4%(1)**
	Rd-MIO	855.0(2)	82.1%(2)	**51.4%(1)**
catwatch	Base1	1000.9(3)	26.2%(3)	14.6%(2)
	Base2	998.4(4)	26.8%(4)	14.9%(3)
	R-MIO	1013.0(2)	26.6%(2)	14.5%(4)
	Rd-MIO	**1022.9(1)**	**26.8%(1)**	**14.6%(1)**
features-service	Base1	316.1(4)	27.9%(4)	5.0%(4)
	Base2	505.1(3)	45.2%(3)	12.4%(3)
	R-MIO	596.5(2)	53.3%(2)	12.8%(2)
	Rd-MIO	**701.7(1)**	**64.3%(1)**	**18.6%(1)**
proxyprint	Base1	1300.0(3)	18.8%(3)	5.5%(3)
	Base2	1302.1(2)	18.7%(2)	5.5%(2)
	R-MIO	**1303.6(1)**	**18.8%(1)**	**5.6%(1)**
	Rd-MIO	1298.8(4)	18.8%(4)	5.5%(4)
rest-news	Base1	**269.3(1)**	**41.5%(1)**	**23.6%(1)**
	Base2	266.9(4)	41.0%(3)	23.1%(4)
	R-MIO	267.9(2)	41.3%(2)	23.2%(3)
	Rd-MIO	267.7(3)	41.0%(4)	23.2%(2)
scout-api	Base1	1519.3(4)	32.9%(4)	19.3%(4)
	Base2	1577.7(3)	34.0%(3)	19.3%(3)
	R-MIO	1794.1(2)	38.4%(2)	20.7%(2)
	Rd-MIO	**1809.8(1)**	**38.6%(1)**	**21.7%(1)**
Average rank	Base1	3.3	3.3	3.1
	Base2	3.0	2.9	2.7
	R-MIO	**1.6**	**1.6**	2.2
	Rd-MIO	2.1	2.3	**1.9**
Friedman test (χ^2, p-value)		7.8, 0.050	6.9, 0.074	3.7, 0.296

Rank value with 1 represents the highest achievement, and values in bold are the highest in the case study

There exist four parameters (i.e., *R-Sampling*, P_s, P_d and *PreMatch*) that produce 40 combinations to configure *Rd-MIO*. To identify the best configuration of *Rd-MIO*, we started from *PreMatch* parameter, because the parameter deals with a static process to derive possible dependencies before the search starts. For a given case study, dependencies derived by the process are inferred. But the derived dependencies might be incorrect, and that may negatively affect the performance of *Rd-MIO*. So we first studied whether *PreMatch* should

Table 14 Pair comparison of our approaches with baselines in terms of #Targets, %Lines and %Branches using Mann-Whitney-Wilcoxon U-tests (p-value) and Vargha-Delaney effect sizes ($\hat{A}_{12}$)

SUT	A vs. B	#Targets			%Lines			%Branches		
		$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
rest-ncs	R-MIO vs. Base1	**0.77**	**<0.001**	**+1.0%**	**0.79**	**<0.001**	**+1.3%**	**0.71**	**<0.001**	**+1.0%**
	R-MIO vs. Base2	0.46	0.873	+0.1%	0.53	0.594	+0.2%	0.42	0.524	-0.2%
	Rd-MIO vs. Base1	**0.70**	**0.024**	**+0.7%**	**0.73**	**0.012**	**+0.9%**	0.64	0.096	+0.7%
	Rd-MIO vs. Base2	0.40	0.181	-0.2%	0.47	0.589	-0.1%	0.36	0.103	-0.5%
	Rd-MIO vs. R-MIO	0.41	0.298	-0.3%	0.43	0.212	-0.3%	0.42	0.448	-0.3%
rest-scs	R-MIO vs. Base1	**1.00**	**<0.001**	**+44.7%**	**1.00**	**<0.001**	**+39.7%**	**1.00**	**<0.001**	**+46.0%**
	R-MIO vs. Base2	**1.00**	**<0.001**	**+24.4%**	**1.00**	**<0.001**	**+20.7%**	**1.00**	**<0.001**	**+23.6%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+44.6%**	**1.00**	**<0.001**	**+39.6%**	**1.00**	**<0.001**	**+46.0%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+24.3%**	**1.00**	**<0.001**	**+20.7%**	**1.00**	**<0.001**	**+23.6%**
	Rd-MIO vs. R-MIO	0.45	0.341	-0.0%	0.47	0.484	-0.0%	0.50	1.000	+0.0%
catwatch	R-MIO vs. Base1	**0.65**	**0.013**	**+1.2%**	**0.69**	**0.001**	**+1.5%**	0.43	0.237	-0.9%
	R-MIO vs. Base2	0.62	0.050	+1.5%	**0.67**	**0.006**	**+1.8%**	0.45	0.401	-0.4%
	Rd-MIO vs. Base1	0.60	0.292	+2.2%	0.62	0.222	+2.4%	0.54	0.695	+0.2%
	Rd-MIO vs. Base2	0.57	0.450	+2.5%	0.56	0.550	+2.6%	0.61	0.279	+0.7%
	Rd-MIO vs. R-MIO	0.49	0.910	+1.0%	0.47	0.732	+0.8%	0.63	0.123	+1.1%
features-service	R-MIO vs. Base1	**1.00**	**<0.001**	**+88.7%**	**1.00**	**<0.001**	**+91.2%**	**1.00**	**<0.001**	**+156.8%**
	R-MIO vs. Base2	**1.00**	**<0.001**	**+18.1%**	**1.00**	**<0.001**	**+18.1%**	**0.74**	**<0.001**	**+3.7%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+122.0%**	**1.00**	**<0.001**	**+130.7%**	**1.00**	**<0.001**	**+272.7%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+38.9%**	**1.00**	**<0.001**	**+42.4%**	**1.00**	**<0.001**	**+50.6%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+17.6%**	**1.00**	**<0.001**	**+20.6%**	**1.00**	**<0.001**	**+45.1%**
proxyprint	R-MIO vs. Base1	0.54	0.623	+0.3%	0.57	0.289	+0.3%	0.55	0.414	+1.0%
	R-MIO vs. Base2	0.55	0.496	+0.1%	0.59	0.161	+0.2%	0.58	0.199	+0.9%
	Rd-MIO vs. Base1	0.49	0.895	-0.1%	0.45	0.587	-0.1%	0.46	0.598	-0.4%
	Rd-MIO vs. Base2	0.50	0.991	-0.3%	0.49	0.950	-0.2%	0.50	0.968	-0.5%
	Rd-MIO vs. R-MIO	0.48	0.809	-0.4%	0.38	0.129	-0.4%	0.41	0.201	-1.4%
rest-news	R-MIO vs. Base1	0.41	0.181	-0.5%	0.47	0.593	-0.4%	0.40	0.104	-1.4%
	R-MIO vs. Base2	0.54	0.648	+0.4%	0.58	0.295	+0.8%	0.56	0.466	+0.8%
	Rd-MIO vs. Base1	0.38	0.238	-0.6%	0.36	0.123	-1.2%	0.40	0.290	-1.4%
	Rd-MIO vs. Base2	0.53	0.817	+0.3%	0.52	0.883	-0.0%	0.57	0.561	+0.8%
	Rd-MIO vs. R-MIO	0.47	0.752	-0.1%	0.42	0.334	-0.8%	0.51	0.902	+0.0%
scout-api	R-MIO vs. Base1	**1.00**	**<0.001**	**+18.1%**	**1.00**	**<0.001**	**+16.9%**	**0.97**	**<0.001**	**+7.2%**
	R-MIO vs. Base2	**1.00**	**<0.001**	**+13.7%**	**1.00**	**<0.001**	**+12.9%**	**0.92**	**<0.001**	**+7.1%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+19.1%**	**1.00**	**<0.001**	**+17.3%**	**1.00**	**<0.001**	**+12.3%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+14.7%**	**1.00**	**<0.001**	**+13.3%**	**0.98**	**<0.001**	**+12.3%**
	Rd-MIO vs. R-MIO	0.52	0.786	+0.9%	0.46	0.637	+0.4%	**0.74**	**0.008**	**+4.8%**

Values in bold means A is statistical significant better than B, and Values in red means B is statistical significant better than A. *relative* is calculated based on average of the metric of A and B as $\frac{A-B}{B}$

be enabled by making a pair comparison analysis with Mann-Whitney-Wilcoxon U-tests and Vargha-Delaney effect sizes. Results for each case study are shown in Table 11. With the results, there is no case showing negative side effects when enabling the prematch process, while performance improvements are observed in *rest-news* and *scout-api*. Therefore, *PreMatch* is now enabled by default for *Rd-MIO*. Note that the inference mainly depends on the API schema, e.g., explicit descriptions on actions, names of parameters and resources.

Table 15 Average of #Targets, %Lines and %Branches covered by tests generated by four techniques and their rank

Resource Graph	*Showing*		*ExistenceDependency*			*PropertyDependency*		
Settings	*Dependency*	Techniques	#Targets	%Lines	%Branches	#Targets	%Lines	%Branches
Dense-Central	*N*	Base1	592.9(3)	51.5%(3)	26.5%(2)	584.3(4)	48.4%(3)	23.8%(3)
		Base2	589.2(4)	50.9%(4)	25.2%(4)	587.1(3)	48.3%(4)	23.2%(4)
		R-MIO	621.2(2)	53.0%(2)	26.3%(3)	624.1(2)	50.8%(2)	24.3%(2)
		Rd-MIO	**758.7(1)**	**66.1%(1)**	**29.1%(1)**	**809.4(1)**	**65.0%(1)**	**29.6%(1)**
	Y	Base1	588.1(4)	50.7%(3)	24.9%(3)	589.2(3)	48.5%(3)	23.1%(3)
		Base2	588.9(3)	50.4%(4)	24.3%(4)	588.9(4)	48.1%(4)	22.4%(4)
		R-MIO	619.7(2)	52.5%(2)	25.2%(2)	624.3(2)	50.5%(2)	23.6%(2)
		Rd-MIO	**723.3(1)**	**62.3%(1)**	**27.5%(1)**	**758.7(1)**	**60.5%(1)**	**27.6%(1)**
Medium-Deep	*N*	Base1	445.4(3)	38.9%(3)	28.4%(3)	439.0(4)	36.4%(4)	24.8%(3)
		Base2	444.5(4)	38.3%(4)	27.9%(4)	443.9(3)	36.5%(3)	24.6%(4)
		R-MIO	466.4(2)	40.3%(2)	28.6%(2)	463.5(2)	38.2%(2)	25.2%(2)
		Rd-MIO	**636.6(1)**	**60.9%(1)**	**29.5%(1)**	**514.3(1)**	**43.3%(1)**	**27.0%(1)**
	Y	Base1	446.3(4)	36.9%(4)	25.1%(4)	445.7(4)	35.3%(4)	22.4%(4)
		Base2	464.6(3)	39.1%(3)	25.1%(3)	457.9(3)	36.4%(3)	22.7%(3)
		R-MIO	497.5(2)	42.5%(2)	25.2%(2)	482.5(2)	38.8%(2)	23.3%(2)
		Rd-MIO	**539.4(1)**	**45.9%(1)**	**28.2%(1)**	**512.1(1)**	**41.2%(1)**	**24.2%(1)**

Table 15 (continued)

Resource Graph	*Showing*		*ExistenceDependency*			*PropertyDependency*		
Settings	*Dependency*	Techniques	#Targets	%Lines	%Branches	#Targets	%Lines	%Branches
Sparse-Straight	*N*	Base1	505.0(4)	53.9%(3)	30.4%(3)	506.2(4)	53.0%(3)	29.2%(3)
		Base2	510.0(3)	53.8%(4)	30.2%(4)	510.0(3)	52.9%(4)	28.6%(4)
		R-MIO	557.2(2)	58.3%(2)	31.7%(2)	550.7(2)	56.6%(2)	30.0%(2)
		Rd-MIO	**633.9(1)**	**67.4%(1)**	**32.7%(1)**	**612.4(1)**	**63.3%(1)**	**32.9%(1)**
	Y	Base1	511.3(4)	54.2%(4)	29.2%(3)	506.0(4)	52.5%(4)	27.6%(3)
		Base2	530.2(3)	56.2%(3)	28.6%(4)	512.6(3)	52.6%(3)	27.1%(4)
		R-MIO	572.6(2)	60.8%(2)	30.1%(2)	562.4(2)	57.8%(2)	28.8%(2)
		Rd-MIO	**636.5(1)**	**67.1%(1)**	**31.7%(1)**	**609.2(1)**	**62.4%(1)**	**31.3%(1)**
				#Targets		%Line		%Branches
Average rank (all 12 synthetic case studies)		Base1		3.8		3.4		3.1
		Base2		3.2		3.6		3.8
		R-MIO		2.0		2.0		2.1
		Rd-MIO		**1.0**		**1.0**		**1.0**
Friedman test (χ^2, p-value)				33.3, <**0.001**		32.5, <**0.001**		32.7, <**0.001**

Rank value with 1 represents the highest achievement, and values in bold are the highest in the case study

Table 16 Pair comparison of our approaches with baselines in terms of #Targets, %Lines and %Branches using Mann-Whitney-Wilcoxon U-tests (p-value) and Vargha-Delaney effect sizes ($\hat{A}_{12}$) for all of 12 synthetic case studies

SUT	A vs. B	#Targets			%Lines			%Branches		
		$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
Dense-Central HideDependency ExistenceDependency	R-MIO vs. Base1	**0.86**	**<0.001**	**+4.8%**	**0.76**	**<0.001**	**+2.9%**	0.44	0.390	-1.0%
	R-MIO vs. Base2	**0.91**	**<0.001**	**+5.4%**	**0.86**	**<0.001**	**+4.2%**	**0.75**	**0.002**	**+4.1%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+28.0%**	**1.00**	**<0.001**	**+28.2%**	**0.87**	**<0.001**	**+9.5%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+28.8%**	**1.00**	**<0.001**	**+29.9%**	**0.97**	**<0.001**	**+15.1%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+22.1%**	**1.00**	**<0.001**	**+24.6%**	**0.90**	**<0.001**	**+10.7%**
Dense-Central HideDependency PropertyDependency	R-MIO vs. Base1	**0.92**	**<0.001**	**+6.8%**	**0.87**	**<0.001**	**+4.8%**	**0.67**	**0.019**	**+2.4%**
	R-MIO vs. Base2	**0.92**	**<0.001**	**+6.3%**	**0.90**	**<0.001**	**+5.1%**	**0.80**	**<0.001**	**+5.0%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+38.5%**	**1.00**	**<0.001**	**+37.5%**	**1.00**	**<0.001**	**+30.8%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+37.9%**	**1.00**	**<0.001**	**+37.8%**	**1.00**	**<0.001**	**+34.2%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+29.7%**	**1.00**	**<0.001**	**+31.2%**	**1.00**	**<0.001**	**+27.8%**
Dense-Central ShowingDependency PropertyDependency	R-MIO vs. Base1	**0.92**	**<0.001**	**+6.0%**	**0.86**	**<0.001**	**+4.2%**	0.62	0.052	+2.1%
	R-MIO vs. Base2	**0.92**	**<0.001**	**+6.0%**	**0.88**	**<0.001**	**+4.9%**	**0.79**	**<0.001**	**+5.2%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+28.8%**	**1.00**	**<0.001**	**+28.0%**	**1.00**	**<0.001**	**+25.3%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+28.8%**	**1.00**	**<0.001**	**+28.8%**	**1.00**	**<0.001**	**+29.2%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+21.5%**	**1.00**	**<0.001**	**+22.8%**	**1.00**	**<0.001**	**+22.8%**

Table 16 (continued)

SUT	A vs. B	#Targets			%Lines			%Branches		
		$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
Dense-Central ShowingDependency ExistenceDependency	R-MIO vs. Base1	**0.88**	**<0.001**	**+5.4%**	**0.80**	**<0.001**	**+3.4%**	0.57	0.261	+1.2%
	R-MIO vs. Base2	**0.90**	**<0.001**	**+5.2%**	**0.85**	**<0.001**	**+4.1%**	**0.73**	**<0.001**	**+4.0%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+23.0%**	**1.00**	**<0.001**	**+22.8%**	**0.93**	**<0.001**	**+10.3%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+22.8%**	**1.00**	**<0.001**	**+23.5%**	**0.98**	**<0.001**	**+13.4%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+16.7%**	**1.00**	**<0.001**	**+18.7%**	**0.90**	**<0.001**	**+9.0%**
Medium-Deep HideDependency ExistenceDependency	R-MIO vs. Base1	**0.79**	**<0.001**	**+4.7%**	**0.68**	**0.010**	**+3.5%**	0.54	0.292	+0.7%
	R-MIO vs. Base2	**0.83**	**<0.001**	**+4.9%**	**0.78**	**0.002**	**+5.2%**	**0.69**	**0.001**	**+2.5%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+42.9%**	**1.00**	**<0.001**	**+56.6%**	**0.70**	**<0.001**	**+4.0%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+43.2%**	**1.00**	**<0.001**	**+59.2%**	**0.81**	**<0.001**	**+5.9%**
	Rd-MIO vs. R-MIO	**1.00**	**<0.001**	**+36.5%**	**1.00**	**<0.001**	**+51.3%**	**0.66**	**0.010**	**+3.3%**
Medium-Deep HideDependency PropertyDependency	R-MIO vs. Base1	**0.90**	**<0.001**	**+5.6%**	**0.84**	**<0.001**	**+5.0%**	0.57	0.083	+1.6%
	R-MIO vs. Base2	**0.85**	**<0.001**	**+4.4%**	**0.80**	**<0.001**	**+4.7%**	**0.67**	**0.007**	**+2.6%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+17.1%**	**1.00**	**<0.001**	**+19.0%**	**0.97**	**<0.001**	**+8.6%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+15.9%**	**1.00**	**<0.001**	**+18.6%**	**0.98**	**<0.001**	**+9.7%**
	Rd-MIO vs. R-MIO	**0.98**	**<0.001**	**+11.0%**	**0.98**	**<0.001**	**+13.3%**	**0.88**	**<0.001**	**+7.0%**
Medium-Deep ShowingDependency PropertyDependency	R-MIO vs. Base1	**0.94**	**<0.001**	**+8.3%**	**0.97**	**<0.001**	**+10.0%**	**0.79**	**<0.001**	**+3.8%**
	R-MIO vs. Base2	**0.85**	**<0.001**	**+5.4%**	**0.88**	**<0.001**	**+6.6%**	**0.72**	**0.004**	**+2.7%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+14.9%**	**1.00**	**<0.001**	**+16.7%**	**0.89**	**<0.001**	**+8.2%**
	Rd-MIO vs. Base2	**0.96**	**<0.001**	**+11.8%**	**0.98**	**<0.001**	**+13.1%**	**0.85**	**<0.001**	**+7.0%**
	Rd-MIO vs. R-MIO	**0.89**	**<0.001**	**+6.1%**	**0.92**	**<0.001**	**+6.1%**	**0.77**	**<0.001**	**+4.2%**

Table 16 (continued)

SUT	A vs. B	#Targets			%Lines			%Branches		
		$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
Medium-Deep ShowingDependency ExistenceDependency	R-MIO vs. Base1	**0.94**	**<0.001**	**+11.5%**	**0.96**	**<0.001**	**+15.2%**	0.52	0.728	+0.7%
	R-MIO vs. Base2	**0.82**	**<0.001**	**+7.1%**	**0.81**	**<0.001**	**+8.7%**	0.57	0.228	+0.6%
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+20.9%**	**1.00**	**<0.001**	**+24.5%**	**0.72**	**<0.001**	**+12.7%**
	Rd-MIO vs. Base2	**0.94**	**<0.001**	**+16.1%**	**0.97**	**<0.001**	**+17.5%**	**0.73**	**<0.001**	**+12.7%**
	Rd-MIO vs. R-MIO	**0.90**	**<0.001**	**+8.4%**	**0.91**	**<0.001**	**+8.1%**	**0.68**	**0.007**	**+11.9%**
Sparse-Straight HideDependency ExistenceDependency	R-MIO vs. Base1	**0.94**	**<0.001**	**+10.4%**	**0.88**	**<0.001**	**+8.1%**	**0.76**	**0.003**	**+4.5%**
	R-MIO vs. Base2	**0.95**	**<0.001**	**+9.3%**	**0.90**	**<0.001**	**+8.2%**	**0.75**	**0.006**	**+5.1%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+25.5%**	**1.00**	**<0.001**	**+25.2%**	**0.86**	**<0.001**	**+7.5%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+24.3%**	**1.00**	**<0.001**	**+25.2%**	**0.84**	**<0.001**	**+8.2%**
	Rd-MIO vs. R-MIO	**0.99**	**<0.001**	**+13.8%**	**0.99**	**<0.001**	**+15.8%**	0.62	0.074	+2.9%

Table 16 (continued)

SUT	A vs. B	#Targets			%Lines			%Branches		
		$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
Sparse-Straight HideDependency PropertyDependency	R-MIO vs. Base1	**0.94**	**<0.001**	**+8.8%**	**0.85**	**<0.001**	**+6.7%**	**0.65**	**0.036**	**+3.0%**
	R-MIO vs. Base2	**0.92**	**<0.001**	**+8.0%**	**0.86**	**<0.001**	**+6.9%**	**0.75**	**<0.001**	**+5.0%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+21.0%**	**1.00**	**<0.001**	**+19.5%**	**0.97**	**<0.001**	**+12.9%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+20.1%**	**1.00**	**<0.001**	**+19.8%**	**0.98**	**<0.001**	**+15.2%**
	Rd-MIO vs. R-MIO	**0.98**	**<0.001**	**+11.2%**	**0.99**	**<0.001**	**+12.0%**	**0.90**	**<0.001**	**+9.7%**
Sparse-Straight ShowingDependency PropertyDependency	R-MIO vs. Base1	**0.99**	**<0.001**	**+11.1%**	**0.97**	**<0.001**	**+10.1%**	**0.75**	**<0.001**	**+4.5%**
	R-MIO vs. Base2	**0.96**	**<0.001**	**+9.7%**	**0.95**	**<0.001**	**+9.9%**	**0.83**	**<0.001**	**+6.4%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+20.4%**	**1.00**	**<0.001**	**+18.8%**	**0.97**	**<0.001**	**+13.3%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+18.8%**	**1.00**	**<0.001**	**+18.6%**	**0.98**	**<0.001**	**+15.4%**
	Rd-MIO vs. R-MIO	**0.96**	**<0.001**	**+8.3%**	**0.95**	**<0.001**	**+7.9%**	**0.87**	**<0.001**	**+8.4%**
Sparse-Straight ShowingDependency ExistenceDependency	R-MIO vs. Base1	**0.93**	**0.003**	**+12.0%**	**0.89**	**<0.001**	**+12.3%**	0.64	0.100	+3.2%
	R-MIO vs. Base2	**0.85**	**0.011**	**+8.0%**	**0.83**	**0.013**	**+8.3%**	**0.74**	**0.030**	**+5.3%**
	Rd-MIO vs. Base1	**1.00**	**<0.001**	**+24.5%**	**1.00**	**<0.001**	**+23.9%**	**0.89**	**0.002**	**+8.8%**
	Rd-MIO vs. Base2	**1.00**	**<0.001**	**+20.0%**	**1.00**	**<0.001**	**+19.5%**	**0.93**	**<0.001**	**+11.0%**
	Rd-MIO vs. R-MIO	**0.95**	**<0.001**	**+11.2%**	**0.95**	**<0.001**	**+10.4%**	**0.74**	**0.048**	**+5.4%**

SUT column represents the synthetic case studies specified with their generation settings. Values in bold means A is statistical significant better than B, and Values in red means B is statistical significant better than A. *relative* is calculated based on average of the metric of A and B as $\frac{A-B}{B}$

As shown in Algorithm 2, to configure *Rd-MIO*, P_d is a main parameter that controls enabling of *dependency heuristic handling*, and the probability of applying resource-dependency smart sampling and resource-dependency mutation. Therefore, we studied settings for P_d before *R-Sampling* and P_s. Mann-Whitney-Wilcoxon U-test and Vargha-Delaney effect size were applied to compare two settings of P_d, and results are presented in Table 12. Based on the results, we found that $P_d = 1.0$ outperform $P_d = 0.5$ in *feature-service* regarding all three metrics ($p\text{-}value \leq 0.03$ and $\hat{A}_{12} > 0.6$) and in *scout-api* regarding %Branches ($p\text{-}value = 0.049$ and $\hat{A}_{12} = 0.57$), and there is no statistical significant difference for the rest. As such, we selected $P_d = 1.0$ for *Rd-MIO*.

By following the same statistical tests and criteria as in RQ1, settings for *R-Sampling* and P_s were decided, i.e., $R\text{-}Sampling = ConArchive$ and $P_s = 1.0$. Detailed results are represented in Appendix, in Tables 20, 21 and 22.

According to above results, we can conclude that:

RQ2: Resource-dependency MIO (i.e., MIO enhanced with resource-based technique and dependency handling) is capable of automatically generating tests that cover up to 64.2% lines in real RESTful APIs and 87.3% lines in artificial RESTful APIs. Our recommended configuration for applying *Rd-MIO* is with ConArchive strategy with a 100% probability, a 100% probability of applying dependency handling, and enabling of prematch process.

8.2.3 Results of RQ3 (Comparison among Different Techniques)

To compare our novel approaches with the baselines, we selected the best configurations to represent them, i.e., *R-MIO* with ($S = ConArchive,\ P_s = 0.5$), and *Rd-MIO* with ($S = ConArchive,\ P_s = 1.0,\ P_d = 1.0,\ PM = T$), based on the results of RQ1 and RQ2.

Results on Open-Source Case Studies Regarding the seven open-source case studies, results of the four applied techniques (i.e., *Base1*, *Base2*, *R-MIO* and *Rd-MIO*) are reported in Table 13 and in Table 14. As can be seen from those tables, our approaches (i.e., *R-MIO* and *Rd-MIO*) have the best overall results. In Table 13, for each of the case studies, the best average number of #Targets and %Lines are obtained by our approaches in six out of the seven case studies, except *rest-news*, and the best of %Branches are obtained in five out of the seven case studies, except *rest-ncs* and *rest-news*. For these case studies, by comparing the worst metrics value from our approaches with the best results from *Base1* and *Base2*, the differences are minimal, i.e., for *rest-news* it is less than 2 targets, less than 0.5% line coverage and less than 0.4% branch coverage; for *rest-ncs* it is just less than 0.4% branch coverage.

Regarding average ranks, our approaches (i.e., *R-MIO* and *Rd-MIO*) are consistently better than the baselines for all metrics, i.e., *R-MIO* are best for #Targets and %Lines, and *Rd-MIO* are best for %Branches. In addition, from Table 14, in three out of the seven case studies (i.e., *rest-scs*, *feature-service*, and *scout-api*), our approaches achieve a clear improvement over the baselines based on the low p-value and high effect size, i.e., #Target: p-value < 0.001, $\hat{A}_{12} = 1.0$, $relative \in [13.7\%, 122.0\%]$; %Lines: p-value < 0.001, $\hat{A}_{12} = 1.0$, $relative \in [12.9\%, 130.7\%]$; %Branches: p-value < 0.001, $\hat{A}_{12} > 0.74$,

$relative \in [3.7\%, 272.7\%]$. For the rest, in *rest-ncs* and *catwatch*, by comparing with *Base1*, *R-MIO* achieves a slight but statistically significant improvement over the baselines (i.e., p-value < 0.024, and $\hat{A}_{12} \in [0.65, 0.77]$); and in *proxyprint* and *rest-news*, there is no statistically significant difference between our approaches and the baselines.

Results on Synthetic Case Studies Regarding the 12 synthetic case studies, Table 15 reports the average of #Targets, %Lines and %Branches, with a relative rank for all the four techniques. Table 16 reports results of the pair comparisons with relative improvement among the four techniques. With the results, our proposed (i.e., *R-MIO* and *Rd-MIO*) techniques significantly outperformed baselines techniques (i.e., *Base1* and *Base2*). More specifically, in Table 15, for all metrics, *Rd-MIO* is consistently ranked as the best, and *R-MIO* is ranked as the second best with one exception (denoted with blue text in the table). Besides, variance on the ranks in the techniques is significant with the Friedman test, i.e., p-value < 0.01. Moreover, as seen from Table 16, *Rd-MIO* performed significantly better than baselines with all of the 12 case studies for all of the three metrics based on low p-value (i.e., < 0.001), high effect size (i.e., > 0.94) and over 11% relative improvement.

For *R-MIO*, as shown in Table 16, compared with baselines, it has better overall performance, i.e., *R-MIO* is significantly better than baselines on all case studies for #Targets and %Lines metrics, significantly better than *Base1* in 5 out of the 12 case studies for %Branches, and significantly better than *Base2* in 11 out of the 12 case studies for %Branches. Additionally, there is no downside, i.e., neither of baseline techniques performs significantly better than *R-MIO* on any case study.

Based on the experiment results on the 7 open-source and 12 synthetic case studies, we can conclude that:

RQ3: Our proposed techniques (i.e., *R-MIO* and *Rd-MIO*) statistically significantly outperformed the two selected baseline techniques in 2 out of the 3 artificial case studies, 3 out of the 4 real case studies, and all of the 12 synthetic case studies (i.e., relative improvements of line coverage are up to 39.7% for artificial case studies, up to 130.7% for real case studies and up to 59.2% for synthetic case studies).

8.3 Result Discussion

We now discuss the results on each case study in more detail, starting from the ones which have high percentage of independent resources (Table 4), i.e., *rest-scs* ($11/11 = 100\%$), *rest-ncs* ($6/6 = 100\%$), and *catwatch* ($11/13 = 85\%$).

For *rest-scs*, both of *R-MIO* and *Rd-MIO* achieve a high coverage (i.e., 82.1% line coverage), as reported in Table 13. They have a significant improvement, considering the low p-value (i.e.,< 0.001) and high effect size (i.e.,> 0.82) compared with baselines.

For *rest-ncs*, results in Table 13 show a high achievement on line coverage with *R-MIO* and *Rd-MIO* (i.e., 87.5% and 87.2% respectively). But, by comparing with the baselines, improvements (reported in Table 14) are modest. The results show that only the comparison with *Base1* (and not *Base2*) is significant with a low p-value (i.e., < 0.024) and a high effect size (i.e.,> 0.70). However, the relative improvements are modest. One reason for the relative low improvement may be that a number of accessible resources is relatively low (i.e., 6). Thus, given the search budget (i.e., $100k$ HTTP calls), it might be that the baselines are already very close to get the maximum achievable coverage on this case study.

For *catwatch*, the achievements of line coverage (Table 13) are modest, i.e., 26.6% by *R-MIO* and 26.8% by *Rd-MIO*. Regarding comparisons with baselines, the results (Table 14) still indicate an improvement, but there are few statistically significant cases (i.e., p-value $<$ 0.05), modest effect size and relative improvements. By going into the details of the implementation of *catwatch*, we found that all the endpoints on **possibly-dependent** resources (i.e., */config/scoring.project* with POST, */import* with POST) are deactivated in the SUT.[13] This results in no accessible endpoints for manipulating the existence of resources that may limit the effectiveness of our novel resource-based solutions. In addition, retrieve actions (i.e., endpoints with GET) are complex in *catwatch*. For instance, GET */contributors* is to "return all information like name, url, commits count, projects count of all the Contributors for the selected filter", and the filter is specified with seven query parameters: (required) *organizations* with String type for "List of github.com organizations to scan(comma separated)"; *limit* with Integer type for "Number of items to retrieve. Default is 5"; *offset* with Integer type to "Offset the list of returned results by this amount. Default is zero"; *startDate* with String type to "Date from which to start fetching records from database(default = current_date)"; *endDate* with String type to "Date till which records will be fetched from database (default = current_date)"; *sortBy* with String type to "parameter by which result should be sorted. '-' means descending order (default is count of commit)"; *q* with String type to "query parameter for search query (this will be contributor names prefix)". In the implementation of this endpoint, it first needs to validate the values of all of the parameters according to their format constraints. This means that following business logic of the endpoint can be executed if and only if all those inputs are valid. However, such constraints are not formally defined in the OpenAPI/Swagger schema (it is just a comment in the description field), and so our technique cannot handle it yet. A possible item for future work would be to do byte-code instrumentation on the source code of the SUT to analyze and handle these further constraints. This issue may result in the modest improvements on line coverage for all techniques (i.e., $<$ 26.8% in Table 13).

For the remaining four case studies, based on the results reported in Tables 13 and 14, we can identify that, regarding improvements by comparing with baselines, our resource-based techniques are the most effective for *feature-service*, quite effective for *scout-api*, but less effective on *proxyprint* and *rest-news*. As descriptive statistics show in Table 4, it is likely that there exist dependent resources in the four case studies. The different effectiveness in our results might be due to the different dependencies among resources. Therefore, we took a detailed look at APIs and source code of all case studies, to manually identify their resources and dependencies, and compare experiment results with the identified real dependencies. The four case studies are discussed based on the effectiveness of resource-based techniques, from most effective to least, i.e., *feature-service*, *scout-api*, *proxyprint* and *rest-news*.

Regarding *feature-service*, results showed that performance of the four techniques can be ranked as *Base1* (worst), *Base2*, *R-MIO* and *Rd-MIO* (best). Figure 12 represents the real resources and their dependencies identified manually. As seen from the figure, all resources are closely and directly connected, and *Product* is the most frequently depended on. In addition, by checking resource URIs, they are in accord with those dependencies, e.g., *ProductConfiguration* can be manipulated with */products/{productName}/configurations/{configurationName}* and the dependent *Product* is

[13]https://github.com/EMResearch/EMB/blob/master/cs/rest/original/catwatch/catwatch-backend/src/main/java/org/zalando/catwatch/backend/web/admin/AdminController.java

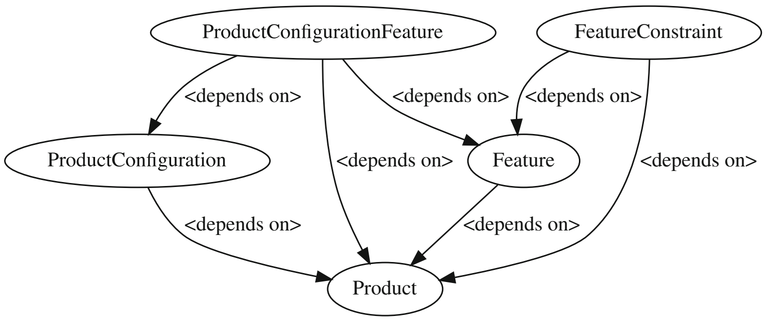


Fig. 12 Resource and their dependencies in *feature-service* case study

shown in the URI. Based on such resource dependencies and URIs, the results might be explained as:

- *Base2* is better than *Base1*, because a set of test templates were developed in *Base2* and the templates are partially handling resources.
- *R-MIO* is better than *Base2*, because *R-MIO* enables handling of an individual based on resources which might help to seek a better combination of multiple resources.
- *Rd-MIO* is better than *R-MIO*, because real dependencies might be identified by *dependency heuristic handling* that might accelerate the process of seeking the better combination.

Regarding line coverage on *feature-service*, *Rd-MIO* strongly improved the line coverage to 64.3% from 27.9% by *Base1* as reported in Table 13.

Regarding *scout-api*, ranks of all techniques are similar with *feature-service*, but *Rd-MIO*, i.e., there is no significant difference between *R-MIO* and *Rd-MIO*. As shown in Fig. 13, in *scout-api*, five out of the seven resources are connected with six dependencies. Two out of the six are *Composition*. The three dependencies of the remaining four, i.e., *Category(v1)/Tag(v2)-User*, *Rating-User* and *User-User*, were implemented with authorization. For instance, by POST */v2/activities/*{*id*}*/rating*, the system only allows a valid *User* to rate an existing *Activity* (i.e., *Rating-User* dependency). The *Activity* is specified with the *id* path parameter, and the *User* is carried by HTTP authorization header of the request. Note that authorization is handled by EvoMaster whereby sampling HTTP actions with a probability of applying one of the valid authentication credentials. The valid authentication credentials are pre-specified in a driver[14] needed when using EvoMaster on a SUT (recall Section 2.3). Thus, the three dependencies can be solved when a valid authentication credential is applied on the related HTTP actions. The remaining dependency, i.e., *Rating-Activity*, is manifested by the path */v2/activities/*{*id*}*/rating*.

Based on such dependencies, the results are explained as:

- with such connected resources in *scout-api*, improvements compared with baselines might be achieved by the manipulation of multiple resources in *R-MIO* and *Rd-MIO*.
- in *scout-api*, dependencies are simple and most of them are exposed to *R-MIO*, so there is no further improvement made by *Rd-MIO*.

[14] https://github.com/EMResearch/EvoMaster/blob/master/docs/write_driver.md

Moreover, in *scout-api*, *R-MIO* and *Rd-MIO* performed better than baselines, but the overall line coverage is still not high, i.e., best is 38.6% by *Rd-MIO*. Therefore, by checking their implementation, we found the same issue as in *catwatch*, i.e., complex input validation where the constraints are not formalized in the schema. For instance, the GET endpoint */v1/activities/*{*id*} in *scout-api* is about *"Read a specific activity"* with one query parameter called *attrs*. This is used to specify what attributes should be included in the response by following specific defined rules, i.e., a *"Comma-separated list"*. However, given a budget of 100*k* HTTP calls, currently EvoMaster does not properly handle such constraints.

Based on the results of *proxyprint*, all techniques achieved modest line coverage, i.e., 18.8% in Table 13, and there is no statistically significant difference reported in the pair comparisons of the four techniques in Table 13. In terms of dependencies among resources, there are many, as shown in Fig. 14, that should result in a good performance of *R-MIO* and *Rd-MIO*. First, by checking its API specification, we found that the SUT does not follow the REST best practices, i.e., 15 resource URIs start with */consumer*, but a creation of a consumer is under */consumer/register*. Furthermore, 22 resource URIs start with */printshops*, but there is no endpoint for creating a printshop, and a collection of printshop examples can be initialized by POST */seed* with *Admin* permission. In addition, by further investigating the implementation, *Printshop* is, to some extent, the most important resource that is depended on by many resources, and has an impact on 65 out of 74 endpoints in *proxyprint*. For the 65 endpoints, it is difficult to cover code for successful requests if there is no valid *PrintShop*.

As mentioned above, there does not exist an endpoint for creating a specific *PrintShop*. In our proposed resource-based techniques, resource-creation endpoints (i.e., POST/PUT) are employed for preparing dependent resources. Therefore, for *proxyprint*, *R-MIO* and *Rd-MIO* probably failed to prepare a *Printshop* for endpoints that require a reference to an existing *Printshop*. This might be the main reason for the modest achieved line coverage and no improvement made by *R-MIO* and *Rd-MIO*. To handle this issue, a possible solution could be to provide more manners for preparing resources, e.g., collect existing resources with GET methods, or create new resources directly in the database.

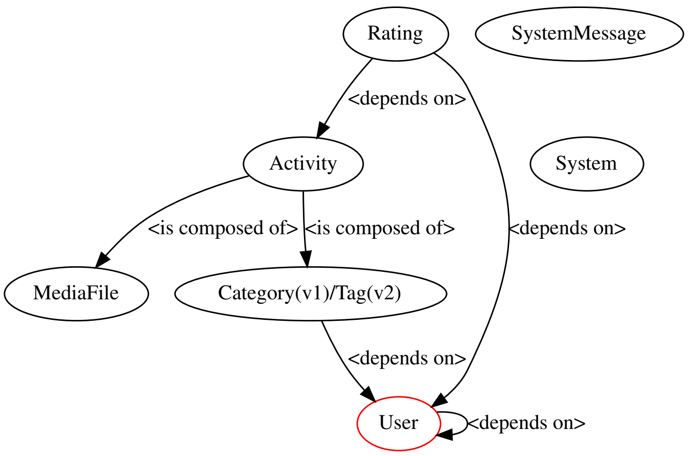


Fig. 13 Resource and their dependency in *scout-api* case study. Resource node with red line means that authorization is required to access the resource

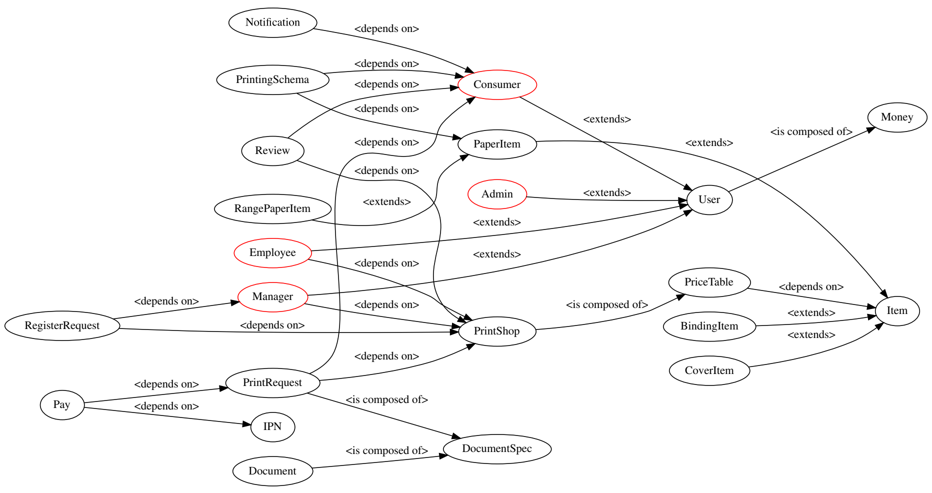


Fig. 14 Resource and their dependency in *proxyprint* case study. - Resource node with red line means that authorization is required to access the resource

Regarding *rest-news*, we made no improvement compared with baselines. But, as can be seen in Table 13, the results of #Targets, %Lines and %Branches do not vary much among the different techniques (i.e., for #Targets, %Lines and %Branches, difference between the best and the worst are just 3, 0.5% and 0.5% respectively). In addition, all techniques achieved around 41% line coverage. In *rest-news*, there are only two resources (Fig. 15), i.e., *News* and *Country*, and *News* depends on *Country* for indicating which country the news are from. But the collection of *Country* is fixed, i.e., there is not endpoint for creating/deleting/updating a *Country*. The collection is used to check whether a HTTP request on *News* is specified with a valid country. With such an implementation, the slight difference among techniques might be explained by considering small size of resources, and simple dependency and implementation. Regarding the results of line coverage, any uncovered lines are probably due to the issue of resources preparation (as discussed in *proxyprint*), i.e., prepare a specific *Country* for *News* with POST/PUT.

Regarding the synthetic SUTs, to study the performance of the different techniques with respect to each case study, we first analyze the results based on the different aspects/properties of the generation of the synthetic case studies. As discussed in Section 7.2, these aspects are three resource graph settings (i.e., *Dense-Central*, *Medium-Deep* and *Sparse-Straight*), two dependency constraints (i.e., *ExistenceDependency* and *PropertyDependency*), and two resource path generation strategies (i.e., *ShowingDependency* and *HideDependency*). By checking results (reported in Table 15) regarding those aspects, we found that there exists a consistent trend for all the techniques that can be visualized with Fig. 16.

Regarding %Lines, in terms of resource graph settings, all of the four techniques have the best performance on *Sparse-Straight*, then on *Dense-Central*, and worst on *Medium-Deep*.

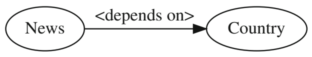


Fig. 15 Resource and their dependency in *rest-news* case study

This may point a precise ranking of difficulty of the problems to solve, i.e., from easy to challenging: *Sparse-Straight*, *Dense-Central* and *Medium-Deep*.

In terms of dependency constraints, all of the four techniques are more effective on case studies with *ExistenceDependency* than case studies with *PropertyDependency*. This confirms our expectations. Besides, in terms of path generation strategies, the trend for the techniques is different for %Lines. One possible explanation of the results might be that the one path generation strategy (i.e., *ShowingDependency*) is designed to provide additional information, however, the additional information might lead to side effects in some of the techniques.

Regarding *Base1*, since it does not employ resource-based solutions, it is reasonable that there is not much difference in its performance on these two configurations. For the other techniques, as they apply strategies based on exploiting the URIs of the resources, one would expect that their performances on *ShowingDependency* would be better than on *HideDependency*. However, this is not the case in the obtained results. Recall the *Dense-Central* graph setting, there exists a one-to-two dependency, e.g., *UEear* depends on *B8v25* and *IUJWo* as shown in Fig. 10. With the *ShowingDependency* strategy, the path would be */iUJWos/*{*iUJWoId*}*/b8v25s/*{*b8v25Id*}*/uEears*. Based on hierarchical structures of such paths, *B8v25* should depend on *IUJWo*. However, such a dependency does not exist. This might limit the effectiveness of *Base2*, *R-MIO* and *Rd-MIO* to case studies with *ShowingDependency* in *Dense-Central*.

In terms of *Medium-Deep* (Fig. 10), there exists an indirect dependency from *VIL0S* to *U1rA1* through *HErqD* and *XpOCt*, thus, with *ShowingDependency*, the path of *VIL0S* is */u1rA1s/*{*u1rA1Id*}*/xpOCts/*{*xpOCtId*}*/hErqDs/*{*hErqDId*} */vIL0Ss/*{*vIL0SId*}. Based on the hierarchical structure, in order to prepare a resource for example like a GET on the path, at least another 4 actions are required, as follows. Note that these 5 actions are regarded as actions with template #7 CREATE-GET on one resource, as discussed in Section 5.1.

```
POST /u1rA1s
POST /u1rA1s/ack/xpOCts
POST /u1rA1s/ack/xpOCts/baz/hErqDs
POST /u1rA1s/ack/xpOCts/baz/hErqDs/bar/vIL0Ss
GET /u1rA1s/ack/xpOCts/baz/hErqDs/bar/vIL0Ss/foo
```

With the maximum length of a test (i.e., 10), this deep hierarchical structure might limit the number of resources in a test, and accelerate consumption of the search budget for each evaluation (i.e., recall that the cost of each fitness evaluation depends on the length of the test cases).

In addition, in the synthetic case studies, the POST methods implemented for creating new resources were using only a *Body* payload, which has an attribute named *id* for representing its identifier, e.g., POST */u1rA1s*. However, for all settings with *ShowingDependency*, the name of the parameter for referring to the resource is *<name of the resource>Id*, e.g., a path parameter *u1rA1Id* of POST */u1rA1s/*{*u1rA1Id*}*/xpOCts*. This is different from just using *id* (which does not contain any reference to the name of the resource). In this case, without exploiting text information, our technique probably fails to match that parameter to the resource just created. This limits the effectiveness of resource preparation by our techniques, e.g., *Base2*, *R-MIO* and *Rd-MIO*. Moreover, since *Rd-MIO* is capable of binding values among resources for making resource correlate to its dependent resource, such additional actions with possible failure of resource preparation may cause a side-effect in *Rd-MIO*. With these results, *Base2* and *R-MIO* performed overall better

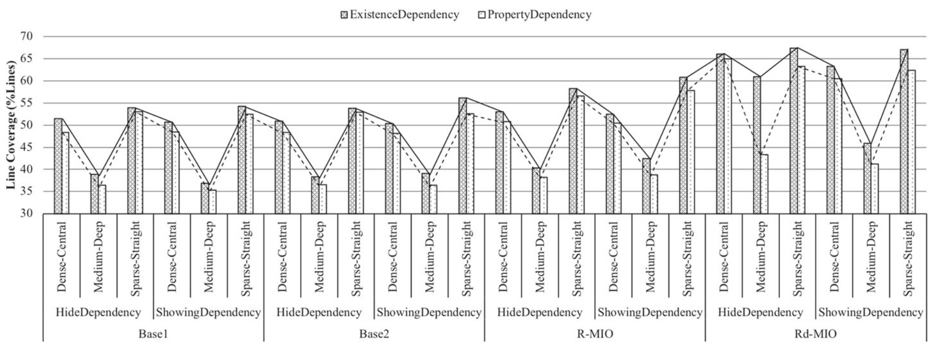


Fig. 16 Visualized trends of average %Lines of each of the techniques by case studies. Note that detailed number is reported in Table 15

in *ShowingDependency* case studies than in *HideDependency*, while *Rd-MIO* consistently showed less effectiveness in *ShowingDependency* than *HideDependency*.

To get more insight on the effectiveness of *R-MIO* and *Rd-MIO*, we analyzed as well a direct comparison between *R-MIO* and *Rd-MIO*. Detailed results (i.e., p-value with Mann-Wilcoxon U-tests, Vargha-Delaney effect sizes and relative improvements) of comparison among techniques can be found in Table 16. We also report in Fig. 17 in Appendix the visualization of the differences among techniques, with bar and line charts, in the categorized synthetic case studies.

Compared with *R-MIO*, we found that *Rd-MIO* achieved more improvements in *Dense-Central* and *Medium-Deep* settings than *Sparse-Straight*. This indicates that *Dependency Handling* helped to generate effective test cases for SUT which handles a number of connected resources, e.g., high-density dependencies, or deep connected dependencies. Besides, by comparing with *R-MIO* in terms of *HideDependency*, there exist notable improvements by *Rd-MIO*, showing that *Dependency Handling* is capable of identifying dependencies even when they are not explicit in the URIs. However, with *PropertyDependency*, the improvements of *Rd-MIO* on *HideDependency* are decreased. For instance, in terms of %Lines, the biggest improvement (i.e., 51.3% which is the highest point of Relative (*Rd-MIO* vs. *R-MIO*) line chart in Fig. 17) by *Rd-MIO* from *R-MIO* is achieved in *Medium-Deep-HideDependency-ExistenceDependenecy*, while the improvement is 13.3% in *Medium-Deep-HideDependency-PropertyDependenecy*. This might imply that dependency relationships can be identified by *Rd-MIO*, but the test data for solving the dependency constraints is not evolved yet within 100k action evaluations. Thus, the search might require more budget for generating suitable data. Regarding *ShowingDependency*, *Rd-MIO* still outperformed other techniques, but its efficiency is limited.

Based on the analysis, we can summarize that:

Our proposed techniques (i.e., *R-MIO* and *Rd-MIO*) statistically significantly outperformed the two selected baselines, especially on SUTs that handle fully independent, or clearly connected, resources. Besides, *Rd-MIO* is particularly effective on SUTs that handle closely connected resources and their dependencies are not exposed in the URIs.

9 Threats to Validity

Conclusion validity. Due to the randomness in search algorithms, results may be significantly affected by chance. To handle this threat, especially in the context of techniques with multiple settings, we first repeated all settings of the techniques 10 times to select the best performing settings. Then, we repeated these representative settings of the techniques 30 times for further technique comparisons. Based on the standard guideline to report search-based software engineering experiments (e.g., Arcuri and Briand 2014), we chose the Friedman test for variance analysis by ranks, the Mann-Whitney U-test to calculate p-value for pair comparisons at the significance level $\alpha = 0.05$ and the Vargha-Delaney $\hat{A}_{12}$, to determine the practical and statistical significance of results.

Construct validity. As suggested in Ali et al. (2010), the same stopping criterion must be applied for the algorithms to avoid any potential bias in results. In the experiments, we set the same search budget (i.e., $100k$ HTTP calls) to deal with this type of validity threat. Using the number of HTTP calls instead of passed time also help in replicating these experiments, as the used stopping criterion is independent of the employed hardware. The execution time strongly depends on the used hardware and what the SUT executes in its business logic. When running EvoMaster on the SUTs in our empirical study, using a laptop with 16GB RAM and an Intel i7 Processor, $100k$ HTTP calls take roughly 1 hour on average (i.e., each HTTP call take around 35ms-40ms), but there is quite a bit of variety among the SUTs. Note, this also includes all the extra network calls that EvoMaster requires to operate, e.g., to extract coverage information from the SUT each time the fitness function is computed, and to tell the driver that the SUT must reset its state at the end of a test evaluation, like deleting all the new data that was added to the database, if any. Whether this amount of time is something that practitioners would run tools such EvoMaster, or if it will be for more or for less time, is something we do not know at the moment. Interviews and surveys from practitioners would be required to answer questions like “for how long would you typically run EvoMaster to generate tests for your APIs?”.

Internal validity. A threat to the internal validity is that we used our prototype implementation to conduct our experiments. Although we carefully tested it, we cannot guarantee that our implementation is bug free. To cope with this threat, our implementation is open-source, so anyone can review the code and replicate this study.

External validity. An external validity threat typical to any empirical study is about the generalization of the results. Our results were obtained from conducting experiments on three artificial REST APIs, four real REST APIs and twelve synthetic REST APIs. The fact that only four real case studies were used in the empirical experiment is due to (1) such enterprise-level REST APIs are normally not open-source, and (2) executions of such experiments on system testing are very time-consuming. Note that only one run with the best configuration is required when applying the approach in practice, e.g., when software engineers need to automatically generate system tests for their RESTful APIs. When we release new versions of EvoMaster, we provide default parameter settings which gave the best results on average on the SUTs in our empirical studies. However, on any specific SUT of a user, some other settings could lead to better results.

10 Conclusions

In recent years, application of REST for building web services has been growing in industry. It is particularly useful to companies to provide public APIs of their services (e.g., on the cloud) over the Internet. However, testing RESTful web services is challenging. In this paper, we propose a resource-based approach to improve search-based test case generation for white-box testing of RESTful web services. Our approach takes advantage of the MIO algorithm and EvoMaster. We design resource-based sampling with five smart sampling strategies, resource-based mutation, and resource dependency handling, to exploit domain knowledge of REST on the handling of HTTP resources and their dependency.

We compared our approach with the default version of EvoMaster on seven open-source RESTful APIs (used in our previous work) and twelve synthetic REST API generated with various resource-based settings for conducting experiments on automated system testing for web/enterprise applications. Based on our results, compared with existing work, our best strategy has an overall best performance among the case studies, and achieves significant improvements on SUTs that handle fully independent, or clearly connected, resources. Relative improvements of up to +130.7% line coverage were achieved.

In the future, we plan to conduct additional experiments with more case studies, and further study the generalization of our approach. Besides, with synthetic RESTful API generator, we plan to extend it by introducing small changes (e.g., bugs) of programs for evaluating our approaches regarding fault finding. Moreover, to reduce failures in handling resources (e.g., create operations), we plan to investigate novel solutions for doing a more intelligent analysis of resources in web services that are not fully following the REST guidelines. Furthermore, we would like to further involve database operations to directly manipulate resources in the RESTful APIs.

EvoMaster and the employed case studies are freely available online on GitHub. To learn more about EvoMaster, visit our webpage at `www.evomaster.org`.

Appendix

This appendix contains several tables and figures for the empirical analyses carried out in this paper. They are not essential for the understanding of the paper, but they provide valuable data to get more insight in the results.

Table 17 We applied nonparametric Aligned Ranks Transformation ANOVA for analyzing effects of different configurations of *RS* and P_s on all three response values for *R-MIO*

SUT	A=*R-Sampling*,	#Targets				%Lines				%Branches			
	B=P_s	Df	F	$Pr(>F)$	$\eta_p{}^2$	Df	F	$Pr(>F)$	$\eta_p{}^2$	Df	F	$Pr(>F)$	$\eta_p{}^2$
catwatch	A	4	28.96	<**0.001**	0.13	4	28.40	<**0.001**	0.13	4	8.37	<**0.001**	0.04
	B	1	7.31	**0.007**	0.01	1	8.56	**0.004**	0.01	1	0.15	0.697	0.00
	A:B	4	3.56	**0.007**	0.02	4	4.73	<**0.001**	0.02	4	0.40	0.809	0.00
features-service	A	4	4.44	**0.001**	0.02	4	1.96	0.100	0.01	4	12.43	<**0.001**	0.06
	B	1	4.72	**0.030**	0.01	1	4.53	**0.034**	0.01	1	0.01	0.919	0.00
	A:B	4	0.62	0.648	0.00	4	1.45	0.215	0.01	4	8.17	<**0.001**	0.04
proxyprint	A	4	17.03	<**0.001**	0.16	4	21.74	<**0.001**	0.20	4	26.77	<**0.001**	0.24
	B	1	0.02	0.886	0.00	1	1.05	0.306	0.00	1	2.33	0.128	0.01
	A:B	4	3.21	**0.013**	0.04	4	2.91	**0.021**	0.03	4	3.58	**0.007**	0.04
rest-news	A	4	18.16	<**0.001**	0.09	4	54.74	<**0.001**	0.24	4	0.49	0.742	0.00
	B	1	0.13	0.722	0.00	1	38.33	<**0.001**	0.05	1	5.11	**0.024**	0.01
	A:B	4	2.63	**0.033**	0.01	4	11.54	<**0.001**	0.06	4	2.40	**0.049**	0.01

Table 17 (continued)

SUT	A=*R-Sampling*,	#Targets				%Lines				%Branches			
	B=P_s	Df	F value	$Pr(> F)$	$\eta_p{}^2$	Df	F value	$Pr(> F)$	$\eta_p{}^2$	Df	F value	$Pr(> F)$	$\eta_p{}^2$
scout-api	A	4	4.83	<**0.001**	0.02	4	15.10	<**0.001**	0.07	4	0.88	0.474	0.00
	B	1	21.81	<**0.001**	0.02	1	11.45	<**0.001**	0.01	1	14.01	<**0.001**	0.02
	A:B	4	1.69	0.150	0.01	4	2.22	0.065	0.01	4	0.89	0.468	0.00

Values in bold means the effect of the factor on the response is significant, i.e., $Pr(> F) > 0.05$, and the effect size is measured by Partial eta-squared $\eta_p{}^2$. Interpretation of $\eta_p{}^2$ is that a value is [0.01, 0.06) for Small, [0.06, 0.14) for Medium, and [0.14, ∞) for Large

Table 18 We applied Friedman test for analyzing variance of different sampling strategies (i.e., *RS*) among case studies for *R-MIO*

R-Sampling	#Targets	%Lines	%Branches
EqualProbability	3.4	3.2	3.8
Actions	3.2	3.2	2.6
TimeBudgets	2.6	2.2	**2.2**
Archive	4.4	4.4	4.0
ConArchive	**1.4**	**2.0**	2.4
Friedman test	χ^2=9.76, p-value=0.045	χ^2=7.36, p-value=0.118	χ^2=5.6, p-value=0.231

Average ranks, χ^2 and p-value are reported. Rank with a small value represents higher achieved coverage, and values in bold are the highest

Table 19 We applied Friedman test for analyzing variance of different probabilities (i.e., P_s) with a fixed RS (i.e., *ConArchive*) among case studies

P_s	#Targets	%Lines	%Branches
0.5	**1.4**	1.6	**1.4**
1	1.6	**1.4**	1.6
Friedman test	χ^2=0.2, p-value=0.655	χ^2=0.2, p-value=0.655	χ^2=0.2, p-value=0.655

Average ranks, χ^2 and p-value are reported. Rank with a small value represents higher achieved coverage, and values in bold are the highest

Table 20 We applied nonparametric Aligned Ranks Transformation ANOVA for analyzing effects of different configurations of *RS* and P_s on all three response values for *Rd-MIO*

SUT	A=*R-Sampling*	#Targets				%Lines				%Branches			
	B=P_s	Df	F	$Pr(>F)$	$\eta_p{}^2$	Df	F	$Pr(>F)$	$\eta_p{}^2$	Df	F	$Pr(>F)$	$\eta_p{}^2$
catwatch	A	4	1.04	0.392	0.03	4	0.98	0.423	0.03	4	0.52	0.723	0.02
	B	1	0.12	0.734	0.00	1	0.17	0.679	0.00	1	0.07	0.797	0.00
	A:B	4	0.26	0.902	0.01	4	0.35	0.845	0.01	4	1.74	0.145	0.06
features-service	A	4	14.25	<**0.001**	0.34	4	18.66	<**0.001**	0.40	4	5.92	<**0.001**	0.17
	B	1	39.24	<**0.001**	0.26	1	62.70	<**0.001**	0.36	1	14.17	<**0.001**	0.11
	A:B	4	1.13	0.348	0.04	4	3.41	**0.011**	0.11	4	1.61	0.177	0.05
proxyprint	A	4	1.12	0.349	0.03	4	1.00	0.411	0.03	4	0.81	0.524	0.03
	B	1	5.54	**0.020**	0.04	1	3.39	0.068	0.03	1	7.34	**0.008**	0.06
	A:B	4	16.97	<**0.001**	0.35	4	16.99	<**0.001**	0.35	4	17.86	<**0.001**	0.37
rest-news	A	4	0.96	0.431	0.03	4	0.09	0.984	0.00	4	0.72	0.581	0.02
	B	1	1.30	0.257	0.01	1	0.05	0.821	0.00	1	2.64	0.107	0.02
	A:B	4	0.19	0.945	0.01	4	0.21	0.931	0.01	4	0.48	0.747	0.01
scout-api	A	4	0.77	0.546	0.03	4	0.97	0.425	0.03	4	0.70	0.591	0.03
	B	1	11.66	<**0.001**	0.10	1	11.05	**0.001**	0.09	1	3.00	0.086	0.03
	A:B	4	0.96	0.435	0.03	4	0.55	0.697	0.02	4	0.45	0.773	0.02

Values in bold means the effect of the factor on the response is significant, i.e., $Pr(> F) > 0.05$, and the effect size is measured by Partial eta-squared $\eta_p{}^2$

Table 21 We applied Friedman test for analyzing variance of different sampling strategies (i.e., *RS*) among case studies for *Rd-MIO*

R-Sampling	#Targets	%Lines	%Branches
EqualProbability	3.8	3.8	3.6
Actions	3.8	3.4	3.9
TimeBudgets	2.6	2.8	2.5
Archive	**2.4**	2.6	2.6
ConArchive	**2.4**	**2.4**	**2.4**
Friedman test	χ^2=4.32, p-value=0.364	χ^2=2.72, p-value=0.606	χ^2=3.92, p-value=0.417

Average ranks, χ^2 and p-value are reported. Rank with a small value represents higher achieved coverage, and values in bold are the highest

Table 22 We applied Friedman test for analyzing variance of different probabilities (i.e., P_s) with a fixed RS (i.e., *ConArchive*) for *Rd-MIO*

P_s	#Targets	%Lines	%Branches
0.5	1.8	1.8	1.8
1	**1.2**	**1.2**	**1.2**
Friedman test	χ^2=1.8, p-value=0.180	χ^2=1.8, p-value=0.180	χ^2=1.8, p-value=0.180

Average ranks, χ^2 and p-value are reported. Rank with a small value represents higher achieved coverage, and values in bold are the highest

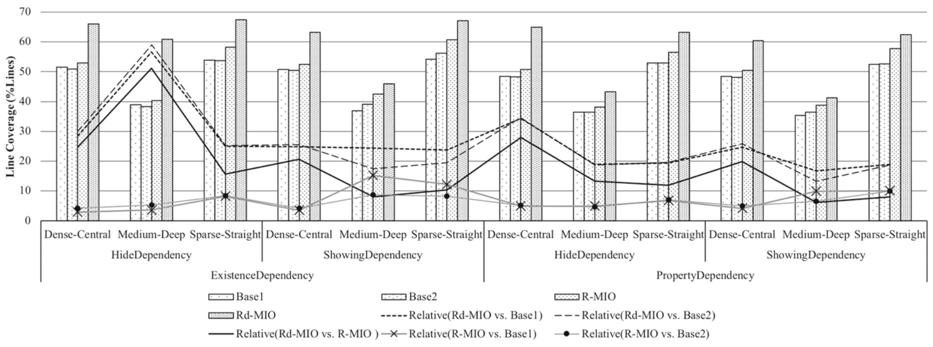


Fig. 17 Visualized difference of average %Lines among techniques for each of the case studies. Note that detailed number of average %Lines is reported in Table 15 and relative improvement of %Lines (e.g., Relative(Rd-MIO vs. R-MIO)) is reported in Table 16

Acknowledgements This work is supported by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385).

Funding Open Access funding provided by Kristiania University College.

References

Ali S, Briand L, Hemmati H, Panesar-Walawege R (2010) A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Trans Softw Eng (TSE) 36(6):742–762

Allamaraju S (2010) Restful web services cookbook: solutions for improving scalability and simplicity. O'Reilly Media Inc.

Alshraideh M, Bottaci L (2006) Search-based software test data generation for string data using program-specific search operators. Softw Test Verification Reliab 16(3):175–203. https://doi.org/10.1002/stvr.v16:3

Arcuri A (2017) RESTful API Automated Test Case Generation. In: IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 9–20. IEEE

Arcuri A (2018) Evomaster: evolutionary multi-context automated system test generation. In: IEEE International conference on software testing, verification and validation (ICST). IEEE

Arcuri A (2018) Test suite generation with the Many Independent Objective (MIO) algorithm. Inform Softw Technol (IST) 104:195–206

Arcuri A (2019) Restful api automated test case generation with evomaster. ACM Trans Softw Eng Methodol (TOSEM) 28(1):3

Arcuri A, Briand L (2011) Adaptive random testing: an illusion of effectiveness? In: ACM Int. Symposium on software testing and analysis (ISSTA), pp 265–275

Arcuri A, Briand L (2014) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw Testing Verif Reliab (STVR) 24(3):219–250

Arcuri A, Galeotti JP (2019) Sql data generation to enhance search-based system testing. In: Proceedings of the genetic and evolutionary computation conference, GECCO '19. Association for Computing Machinery, New York, pp 1390–1398, https://doi.org/10.1145/3321707.3321732

Arcuri A, Galeotti JP (2020) Handling sql databases in automated system test generation. ACM Trans Softw Eng Methodol (TOSEM) 29(4):1–31

Atlidakis V, Godefroid P, Polishchuk M (2019) Restler: Stateful rest api fuzzing. In: Proceedings of the 41st international conference on software engineering, ICSE '19, p 748–758. IEEE Press. https://doi.org/10.1109/ICSE.2019.00083

Bozkurt M, Harman M, Hassoun Y (2013) Testing and verification in service-oriented architecture: a survey. Softw Test Verif Reliab (STVR) 23(4):261–313

Canfora G, Di Penta M (2009) Service-oriented architectures testing: a survey. In: Software engineering. Springer, pp 78–105

Chakrabarti SK, Kumar P (2009) Test-the-rest: an approach to testing restful web-services. In: Future computing, service computation, cognitive, adaptive, content, patterns, 2009. COMPUTATIONWORLD'09. Computation World. IEEE, pp 302–308

Chakrabarti SK, Rodriquez R (2010) Connectedness testing of restful web-services. In: Proceedings of the 3rd India software engineering conference. ACM, pp 143–152

Droste S, Jansen T, Wegener I (1998) On the optimization of unimodal functions with the (1 + 1) evolutionary algorithm. In: Proceedings of the international conference on parallel problem solving from nature, pp 13–22

Ed-douibi H, Cánovas Izquierdo JL, Cabot J (2018) 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

Fertig T, Braun P (2015) Model-driven testing of restful apis. In: Proceedings of the 24th international conference on World Wide Web. ACM, pp 1497–1502

Fielding RT (2000) Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine

Fraser G, Arcuri A (2012) Sound empirical evidence in software testing. In: ACM/IEEE International conference on software engineering (ICSE), pp 178–188

Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Comput Surv (CSUR) 45(1):11

Karlsson S, Causevic A, Sundmark D (2020) Quickrest: property-based test generation of openapi described restful apis. In: IEEE International conference on software testing, verification and validation (ICST). IEEE

Kay M, Wobbrock J (2019) Artool: aligned rank transform for nonparametric factorial anovas. R package version 0.10.6.9000. https://doi.org/10.5281/zenodo.594511. https://github.com/mjskay/ARTool

Lamela Seijas P, Li H, Thompson S (2013) Towards property-based testing of restful web services. In: Proceedings of the twelfth ACM SIGPLAN workshop on Erlang. ACM, pp 77–78

Martin S, Liermann J, Ney H (1998) Algorithms for bigram and trigram word clustering. Speech Commun 24(1):19–37

Pinheiro PVP, Endo AT, Simao A (2013) Model-based testing of restful web services using uml protocol state machines. In: Brazilian workshop on systematic and automated software testing

Rojas JM, Vivanti M, Arcuri A, Fraser G (2017) A detailed investigation of the effectiveness of whole test suite generation. Empir Softw Eng (EMSE) 22(2):852–893

Segura S, Parejo JA, Troya J, Ruiz-Cortés A (2017) Metamorphic testing of RESTful web APIs. IEEE Transactions on Software Engineering (TSE)

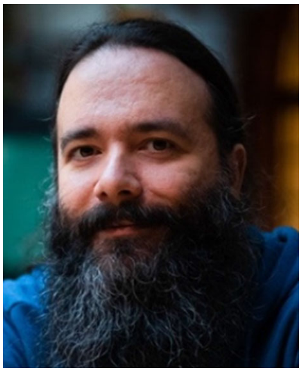
Viglianisi E, Dallago M, Ceccato M (2020) Resttestgen: automated black-box testing of restful apis. In: IEEE International conference on software testing, verification and validation (ICST). IEEE

Wobbrock JO, Findlater L, Gergle D, Higgins JJ (2011) The aligned rank transform for nonparametric factorial analyses using only anova procedures. In: Proceedings of the SIGCHI conference on human factors in computing systems. ACM Press, pp 143–146. https://doi.org/10.1145/1978942.1978963, http://depts.washington.edu/aimgroup/proj/art/

Zhang M, Marculescu B, Arcuri A (2019) Resource-based test case generation for restful web services. In: Proceedings of the genetic and evolutionary computation conference, pp 1426–1434

Man Zhang is a Postdoctoral Researcher at the Artificial Intelligence in Software Engineering (AISE) Lab, Kristiania University College, Norway. Previously, she obtained her PhD in Computer Science at Simula Research Laboratory and University of Oslo, Norway (2015 - 2018). Her main research focuses on developing novel methods with search techniques for automated test generation for enterprise systems.

Bogdan Marculescu is a post-doctoral researcher at Kristiania University College, where he is part of the Artificial Intelligence in Software Engineering (AISE) lab. His research focuses on software testing and applying metaheuristic techniques to improve software testing. He received his PhD in software engineering from Blekinge Institute of Technology, Sweden, in 2017.



Dr. Andrea Arcuri is a Professor of Software Engineering at Kristiania University College and Oslo Metropolitan University, Oslo, Norway. His main research interests are in software testing, especially test case generation using evolutionary algorithms. Having worked 5 years in industry as a senior engineer, a main focus of his research is to design novel research solutions that can actually be used in practice. Dr. Arcuri is the main-author of EvoMaster and a co-author of EvoSuite, which are open-source tools that can automatically generate test cases using evolutionary algorithms. He received his PhD in software testing from the University of Birmingham, UK, in 2009.

Affiliations

Man Zhang[1] · **Bogdan Marculescu**[1] · **Andrea Arcuri**[1,2]

Bogdan Marculescu
bogdan.marculescu@kristiania.no

Andrea Arcuri
andrea.arcuri@kristiania.no

[1] Kristiania University College, Oslo, Norway

[2] Oslo Metropolitan University, Oslo, Norway