

# Resource-based Test Case Generation for RESTful Web Services

Man Zhang  
Kristiania University College  
Oslo, Norway  
man.zhang@kristiania.no

Bogdan Marculescu  
Kristiania University College  
Oslo, Norway  
bogdan.marculescu@kristiania.no

Andrea Arcuri  
Kristiania University College  
Oslo, Norway  
andrea.arcuri@kristiania.no

## ABSTRACT

Nowadays, RESTful web services are widely used for building enterprise applications. In this paper, we propose an enhanced search-based method for automated system test generation for RESTful web services. This method exploits domain knowledge on the handling of HTTP resources, and it is integrated in the Many Independent Objectives (MIO) search algorithm. MIO is an evolutionary algorithm specialized for system test case generation with the aim of maximizing code coverage and fault finding. Our approach builds on top of the MIO by implementing a set of effective templates to structure test actions, based on the semantics of HTTP methods, used to manipulate the web services' resources. We propose four novel sampling strategies for the test cases that can use one or more of these test actions. The strategies are further supported with a set of new, specialized mutation operators that take into account the use of these resources in the generated test cases. We implemented our approach as an extension to the EvoMASTER tool, and evaluated it on seven open-source RESTful web services. The results of our empirical study show that our novel, resource-based sampling strategies obtain a significant improvement in performance over the baseline MIO (up to +42% coverage).

## CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Search-based software engineering**;

## KEYWORDS

Search-based Test Case Generation, RESTful Web Service Testing

### ACM Reference Format:

Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based Test Case Generation for RESTful Web Services. In *Genetic and Evolutionary Computation Conference (GECCO '19)*, July 13–17, 2019, Prague, Czech Republic. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3321707.3321815>

## 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 the HTTP protocol [2, 15]. It is very useful for developing web services with public APIs over a network.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

GECCO '19, July 13–17, 2019, Prague, Czech Republic

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6111-8/19/07...\$15.00

<https://doi.org/10.1145/3321707.3321815>

Currently, REST has been applied by many companies for providing their services over the Internet, e.g., Google<sup>1</sup>, Amazon<sup>2</sup>, and Twitter<sup>3</sup>. However, in spite of their widespread use, testing such RESTful web services is quite challenging [9, 10].

In this paper, we propose a novel approach to enhance the automated generation of systems tests for RESTful web services using search-based methods [16]. To generate tests using search-based methods, an evolutionary algorithm, the Many Independent Objectives algorithm (MIO) [5], is employed. The 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 [13], so that an individual is mainly manipulated by sampling and mutation (no crossover). We implemented our approach as an extension of an existing test case generation tool: EvoMASTER [4, 6]. During the search, EvoMASTER assesses the fitness of individual test cases using runtime code-coverage metrics and fault finding ability. The 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 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 not*) 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 kind of individuals, we propose a resource-based smart sampling operator and a resource-based mutation operator in our approach. For the smart sampling operator, we define four sample methods. At each sampling of a new random individual, one of these methods is applied to sample a test. These methods are proposed 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 a 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 propose five novel operators to mutate the structure of the individuals, with respect to their use of the resources.

We conducted an empirical study on our novel approach by comparing it with existing work. Experiments were carried out

<sup>1</sup><https://developers.google.com/drive/v2/reference/>

<sup>2</sup><http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html>

<sup>3</sup><https://dev.twitter.com/rest/public>

on seven open-source case studies, obtained from an open-source benchmark<sup>4</sup>, made for experimentation in automated system testing of web/enterprise applications. Results of our empirical study show that our novel techniques can significantly improve the performance of the test generation (up to 42%) on SUTs that use independent, or clearly connected, resources.

## 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.
- *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 [15] in 2000, and it is now widely applied in industry, e.g., Google<sup>1</sup>, Amazon<sup>2</sup>, and Twitter<sup>3</sup>. 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: 1) resources should be identified in the requests by using Uniform Resource Identifiers (URIs); 2) resources should be separated from their representation, i.e., the machine-readable data describing the current state of a resource; 3) 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 the resource-related ones.

### 2.2 The MIO Algorithm

The Many Independent Objective (MIO) algorithm [5] 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 [13] 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 an empty population for each optimization target. 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 uncovered targets. Next, 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 [5]. Moreover, to make a trade-off between exploration and exploitation of the search landscape, MIO is integrated with an 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 the exploitation by reducing the probability of random sampling  $P_r$ .

In this paper, we improve the performance of the sampling (i.e., *RandomIndividual()*) and mutation (i.e., *Mutate(T)*) operators of the MIO algorithm, and assess these improvements on the problem of testing RESTful web services.

### 2.3 RESTful API Test Case Generation

In [6], we proposed a search-based approach for automatically generating systems tests for RESTful web services, using the MIO algorithm (see 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., 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.
- *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 they are doing partial updates instead of a full resource replacement.

<sup>4</sup><https://github.com/EMResearch/EMB>

**Algorithm 1:** Pseudo-code of the MIO Algorithm [5]

---

**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      $p \leftarrow Mutate(T)$ 
9   end
10  foreach element  $k$  of  $ReachedTargets(p)$  do
11    if  $IsTargetCovered(k)$  then
12       $UpdateArchive(A, p)$ 
13       $T \leftarrow T \setminus T_k$ 
14    else
15       $T_k \leftarrow T_k \cup \{p\}$ 
16      if  $|T_k| > n$  then
17         $RemoveWorstTest(T_k, \delta)$ 
18      end
19    end
20  end
21   $UpdateParameters(F, P_r, n)$ 
22 end

```

---

- **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<sup>5</sup>. It has two components [4]: *Core* which mainly implements a set of search algorithms for test case generation (e.g., WTS [19]); 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 and fault finding ability.

### 3 RELATED WORK

In this section, we summarize existing work regarding the automated testing of RESTful web services.

Most approaches to test REST APIs rely on formal models or specifications [11, 12, 14, 17, 18, 20]. The models or specifications often describe test inputs, exposed methods of SUTs, behaviors of SUTs, specific characteristics of REST or testing requirements. For instance, Chakrabarti *et al.* [12] defined a specification to formalize connections among resources of a RESTful service, and further focus on testing such “connectedness”. In our approach, we also consider relationships among resources, and we used this information to improve the search by enhancing how test cases are sampled. However, our testing approach aims at maximizing code coverage and fault finding, and not only examining the connections.

Fertig *et al.* [14] developed a Domain Specific Language to describe APIs, including HTTP methods, authentication and resource model. In addition, the model also requires additional constraints

<sup>5</sup>[www.evomaster.org](http://www.evomaster.org)

```

1 "/products":{
2   "get":{"operationId":"getAllProducts"...},
3 "/products/{productName}":{
4   "get":{"operationId":"getProductByName"...},
5   "post":{"operationId":"addProduct"...},
6   "delete":{"operationId":"deleteProductByName"...},

```

**Figure 1:** Snippet example of Swagger JSON definitions for a RESTful API

**Table 1:** Definitions of resource-based templates used to generate tests on a resource.

#	Description	IsIndep.	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	POST-POST
7	To retrieve an existing resource	No	POST-GET
8	To replace an existing resource	No	POST-PUT
9	To (partially) update an existing resource	No	POST-PATCH(-PATCH)
10	To delete an existing resource	No	POST-DELETE

1) IsIndep.→ IsIndependent; 2) Note that #6-10 are only applicable if there exists POST or PUT on the resource or one of its ancestors' resource

on attributes of the resource, e.g., an attribute with integer type should be between 1 and 10. A set of test cases can be generated from such a model. Seijas *et al.* [17] proposed an approach to generate test cases based on property-based test models, and UML state machines are applied [18] to construct behavior models for test case generation.

Segura *et al.* [20] 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.

As discussed, in [6] we proposed a means of generating test cases for RESTful APIs by using search-based methods to create sequences of HTTP calls. The approach is automatic, relying only on information obtained from the API specifications and code instrumentation to generate cases. It does not, however, consider relationships between resources when generating these test cases. We build upon the approach in [6] and enhance it by adding resource handling methods and strategies, as well as developing tailored sampling and mutation operators.

## 4 PROPOSED APPROACH

### 4.1 Resource and Individual

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. More details can be found in Section 2.1. Figure 1 shows a snippet example of a specification of REST APIs using Swagger<sup>6</sup>. As shown in the figure, APIs are structured with resource URIs, and relevant HTTP methods are defined for each resource.

<sup>6</sup><https://swagger.io/>

Search-based techniques use random sampling to create new individuals. However, by sampling resources for HTTP calls randomly, it is unlikely we will be able to generate the same resource for several HTTP calls. If relationships between resources and actions exist, random sampling is very unlikely to produce tests that result in good coverage. Therefore, we define a set of templates that list meaningful combinations of actions on one resource based on the semantics of the HTTP methods. Thus, we use these templates to sample new individuals, instead of sampling them completely at random. However, search is still needed for example to find the right query parameters for the URIs, the content of the body payloads (e.g., JSON objects), and the HTTP headers.

The templates are defined in Table 1. Each template is defined based on the types of HTTP methods, along with whether the related resource exists. Note that we intentionally make the template short (i.e., at most combine two different types of HTTP methods). Because, in terms of one action, code coverage is normally attributed to the action on a resource (also including whether the resource exists), rather than if any other action is executed before it. For instance, if PATCH-DELETE improves the code coverage, which is normally because of the resource updated by PATCH, then we can use POST-DELETE instead. To handle the cases where the existence of a specific resource might help improve fitness when other actions use it afterwards, we use mutation to replace different templates but keep same values on the resource (i.e., *MODIFY* mutation operator discussed later). In this paper, we only use POST to create resources. Because the search usually chooses new values for the parameters of the actions, this means that it is unlikely that the newly sampled values had been previously applied on a POST for creating that corresponding resource. Thus, DELETE is almost the same as the situation when no POST is used.

In the context of testing, we also capitalize on invalid sequences of actions. For example, templates #2 and #3 generate a test that aims to update or delete a resource that does not exist. We consider template #4 independent. However, when making a PUT on a resource that does not exist, the PUT may create it or return an 4xx status. Note that templates #5, #7-#10 (Table 1) are the same as the templates from our previous work [6] (Section 2.3). These templates were applied on sampling a test. But, 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. In addition, we define properties (i.e., **independent**, **non-independent**) for all templates.

Internal relations among resources in the SUT are usually unknown. So, it is not clear, based on the URIs alone, if 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). Therefore, we identify **independent** templates that, when actions with the template are executed, do not have any effect on follow-up actions on any resource. We further define a **non-independent** template as a template for which independence cannot be assumed. Note that a **non-independent** 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. Furthermore,

we defined an individual based on such a template as *a sequence of resource(s) with HTTP calls following the defined template*.

Consider an API that deals with Products and Users. There can be operations to retrieve specific products and users by *id* as shown in Example 1. Each line represents an action which follows the format `<a method on a resource path with/without parameter> : <the method on the path with values of the parameters>`. In the example, actions in lines 1-2 follow template #7, and actions in lines 3-4 follow template #9. Furthermore, the action on line 1 is to prepare a resource for the action on line 2, and the value on the property *id* is bounded.

```
1 POST /products : POST /products/ #json object
2 GET /products/{id} : GET /products/P001
3 POST /users/{id} : POST /users/U001
4 PATCH /users/{id} : PATCH /users/U001
```

#### Example 1. A test with template #7 and #9

In the following subsections, we explain how to sample (Section 4.2) and mutate (Section 4.3) such an individual during the search.

## 4.2 Resource-based Smart Sampling

In this paper, we propose four methods to sample individuals as shown in Table 2. One rationale behind the use of these methods is to distinguish related actions from independent actions (i.e., actions followed with dependent template). By isolating actions with independent templates, we can reduce useless invocations during the search. For example, a test is sampled at random as below:

```
1 GET /products/{id} : GET /products/P001
2 POST /users/{id} : GET /users/U001
3 GET /products/{id} : GET /products/P007
```

The first action is to GET a product by *id*, which is fully independent from following two actions. Then, when applying a mutation on the parameter *id* 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. Therefore, we developed the *S1iR* strategy to sample actions with independent templates in a test. For testing two or more 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. When manipulating multiple resources in a test (i.e., *S2dR* and *SMdR* strategies), only a non-parameter GET is allowed at the last position of the resources (e.g., *GET /products* is allowed, but not *GET /products/{id}*). 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. 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, and the *SMdR* for handling the possibility of complex, multi-resource dependencies. In addition, we also define an applicable condition for each of the methods regarding exposed HTTP methods on resources in a SUT, i.e., the method can be applied to sample a test only if the condition is satisfied. To an extent, a resource structure may be derived with its applicable methods.

When a new individual is sampled with probability  $P_r$  (recall Algorithm 1), the individual is sampled with our resource-based smart 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

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

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

is chosen, with probabilities  $P_m(S1iR) + P_m(S1dR) + P_m(S2dR) + P_m(SMdR) = 1$ .

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, 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.

**Action-Based:** the probability for each applicable method is derived based on a number of independent or not templates for all resources. It is calculated as:

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

where  $n_{temp}$  is the sum of the number of applicable templates for all resources,  $n_{itemp}$  is a sum of a number of independent templates for all resources, and  $k$  is a configurable weight (we set it 1 in this implementation). Then,

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

where  $(s_i, w_i) \in S = \{(s_i, w_i) | s_i \text{ is an applicable method of } S1dR, S2dR \text{ and } SMdR, w_i \text{ is a weight of the method, } i = 1 \dots n_{dm}\}$ .

**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) in the beginning of the sampling (i.e., the used time budget for sampling is less than 50%), and then 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 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  $\delta = 0.1$ ,  $(s_i, r_i) \in S = \{(s_i, r_i) | s_i \text{ is an applicable method, } r_i \text{ is a rank for the applicable methods that is computed based on improved times, } i = 1 \dots n_m\}$ .

**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, the test shown in Example 1 could achieve an improvement, but it would not be known if the improvement is due to the actions on the first resource, actions on the second resource, or combination of the two resources. If we first used some of the budget to sample the first resource and second resource separately, 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).

### 4.3 Resource-based Mutation

When we sample a new individual during the search, we use our templates with some pre-defined 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 and body payloads), and mutate structure of a test (i.e., adding or removing HTTP calls in the test). The difference is that mutations on the values of an action might automatically update all previous actions related to that same resource. Considering the example (shown in Example 1), when we mutate a test by changing values on parameters of actions, first we select the longest path among actions on one resource (e.g., `/products/{id}` on line 2 for the first resource). Once the value is mutated, update other actions on the resource with the same value (e.g., update the value on *id* property of *json object* that is the value of a body parameter of *POST /products/* on line 1).

In addition, we propose five operators to mutate a structure of the individual for exploiting relationship among resources and different template on resources: 1) *DELETE*: delete a resource together with all associated actions; 2) *SWAP*: swap the position of two resources together with all associated actions; 3) *ADD*: add a set of actions on a new resource path; 4) *REPLACE*: replace a set of actions on a resource with another set of actions on a new resource path; 5) *MODIFY*: modify a set of actions on a resource with another template. For instance, if *SWAP* is applied, the test (shown in Example 1) will be



**Table 3: Descriptive statistics of the case studies.**

Name	#Classes	LOCs	Endp.	Res.	Indep. Res.
<i>rest-ncs</i>	9	602	6	6	6
<i>rest-scs</i>	13	859	11	11	11
<i>rest-news</i>	10	718	7	4	1
<i>catwatch</i>	69	5442	23	13	11
<i>feature-service</i>	23	2347	18	11	1
<i>proxyprint</i>	68	7534	74	56	26
<i>scout-api</i>	75	7479	49	21	2

Note that Endp. → Endpoints; Res. → Resource; Indep. → Independent

```

1 POST /users/{id} : POST /users/U001
2 PATCH /users/{id} : PATCH /users/U001
3 POST /products : POST /products/ #json object
4 GET /products/{id} : GET /products/P001

```

## 5 EMPIRICAL STUDY

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

**RQ1:** Do our novel techniques achieve any improvement compared to existing work?

**RQ2:** Among the different techniques we presented in this paper, which one gives the best results?

### 5.1 Experiment Setting

To assess our novel approach, we compared it with existing work [5, 6] (described in Section 2.3) by applying both to seven open-source RESTful web services. The seven case studies from a benchmark<sup>4</sup> were previously selected for experiments on RESTful software testing approach [5, 6], which consists of three artificial APIs and four real-world web services. Table 3 shows detailed descriptive statistics of the case studies, including their number of Java/Kotlin class files, lines of code, number of endpoints, and number of accessible resources with the number of independent resources among them. In terms of case studies, REST Numerical Case Study (*rest-ncs*) and REST String Case Study (*rest-scs*) are artificial APIs that were previously used in unit testing for experiments on solving numerical [7] and string [3] problems. *rest-news* is also an artificial API, which was developed for educational purposes on enterprise development using REST<sup>7</sup>. 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 [5, 6].

Experiment settings are reported in Table 4. In the experiment, we selected two baselines: one (i.e., Base1) is an implementation of default MIO for the REST problem; the other (i.e., Base2) is also based on the MIO, but it was integrated with smart sampling techniques specialized for sampling REST APIs [6]. 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.  $P_s = 0.5$ . In addition, for our approach, all five sampling strategies ( $S \in \{Action, Archive, ConArchive, Used-Budget, Equal\}$ ) combined with the proposed mutation are applied on the experiment with four different probabilities ( $P_s \in \{0.25, 0.5, 0.75, 1.0\}$ ) of using the proposed sampling at the sampling phase (for example, if

**Table 4: Description of experiment settings**

Configuration	Operator and Parameter Settings for MIO
Base1	Sample: Random, Mutation: MIO Default
Base2	Sample: Smart $P_s = 0.5$ , Mutation: MIO Default
Resource	Sample: Resource-based Smart $S \in \{Action, Archive, ConArchive, Used-Budget, Equal\}$ with $P_s \in \{0.25, 0.5, 0.75, 1.0\}$ Mutation: Resource-based Mutation

$P_s = 0.75$ , MIO applies our novel sampling to sample an individual with 75% probability, and applies random sampling with 25% probability). For each setting, we ran MIO using given values for search budget (i.e., 100k HTTP calls), population size (i.e., 10), maximum length of a test (i.e., 10), probability of sampling (i.e., 0.5), and start of focused search (i.e., after 50% budget used). In addition, to take into account the randomness of employed search algorithms, we repeated each experiment 40 times for all selected case studies. Note that, because there only exist independent resources in *rest-ncs* and *rest-scs* (Table 3), all sampling strategies perform equally, i.e. *S1iR* is always selected. Thus, we only ran one of those five strategies (e.g., *Equal*) on those two case studies.

### 5.2 Experiment Results

To evaluate the effectiveness of the generated test cases, we use the default coverage criteria that EvoMASTER optimizes for by default in its fitness function. This is based on code coverage metrics like statement and branch coverage, and returned status codes from the HTTP responses. EvoMASTER computes a total number of such testing targets that are covered. Table 5 reports an average covered targets with a relative rank for all selected techniques (see Table 4) for all case studies. In addition, we also compared each setting of our approach with baselines based on the covered targets from 40 independent runs. Following the guidelines [8] for reporting search-based software engineering experiments, we used Vargha-Delaney effect sizes  $\hat{A}_{12}$  and non-parametric Mann-Whitney-Wilcoxon U-tests at a significant level  $\alpha = 0.05$  to make the pair comparison analysis. These statistics are also reported in Table 5.  $BT_1$  denotes that the approach achieved a better performance than Base1 that is determined when  $\hat{A}_{12} > 0.5$  and  $p$ -values  $< 0.05$ . Similarly,  $BT_2$  is the result of a comparison with Base2.

As can be seen from Table 5, our approach has the best overall result. The best average number of covered targets are obtained by our approach in six out of the seven case studies, except *rest-news*. In addition, in five out of the seven case studies (i.e., *rest-ncs*, *rest-scs*, *catwatch*, *feature-service*, and *scout-api*), our approach achieves a clear improvement over the baseline with respect to ranks (i.e., Base1 is always ranked as last, and the best rank of Base2 is 15th from 22 candidates) and statistical test results (i.e.,  $BT_1 > 75\%$  and  $BT_2 > 55\%$ ). Regarding the *rest-news* and *proxyprint*, by comparing the worst average number of covered targets from our approach with the best result from Base1 and Base2, the difference is minimal, i.e., less than 2 targets for *rest-news* and less than 20 targets for *proxyprint*. According to the overall results in Table 5, we can conclude that:

<sup>7</sup>[https://github.com/arcu82/testing\\_security\\_development\\_enterprise\\_systems](https://github.com/arcu82/testing_security_development_enterprise_systems)

**Table 5: Average numbers of covered test targets and their rank. Rank 1 represents the highest achieved coverage, and values in bold are the highest in the case study. We also specify if better than baseline (i.e.,  $\hat{A}_{12} > 0.5$  and  $p$ -values  $< 0.05$ ), and the  $\chi^2$  and  $p$ -value of the Friedman test.**

SUT	Base1	Base2	Sample Strategy	Resource-based Sampling				%BT <sub>1</sub>	%BT <sub>2</sub>
				0.25	0.5	0.75	1.0		
<i>rest-ncs</i>	531.5(5)	530.4(6)		531.6(4):BT <sub>2</sub>	532.4(2):BT <sub>1</sub> ,BT <sub>2</sub>	<b>532.6(1):BT<sub>1</sub>,BT<sub>2</sub></b>	532.0(3):BT <sub>1</sub> ,BT <sub>2</sub>	75%	100%
<i>rest-scs</i>	538.4(6)	623.8(5)		766.5(4):BT <sub>1</sub> ,BT <sub>2</sub>	<b>766.8(1):BT<sub>1</sub>,BT<sub>2</sub></b>	766.6(2):BT <sub>1</sub> ,BT <sub>2</sub>	766.5(3):BT <sub>1</sub> ,BT <sub>2</sub>	100%	100%
<i>rest-news</i>	299.7(22)	<b>302.0(1)</b>	Actions	301.1(19):BT <sub>1</sub>	301.6(4):BT <sub>1</sub>	301.1(17):BT <sub>1</sub>	301.4(12):BT <sub>1</sub>	95%	0%
			Archive	301.3(14):BT <sub>1</sub>	301.5(8):BT <sub>1</sub>	301.5(6):BT <sub>1</sub>	301.4(10):BT <sub>1</sub>		
			ConArchive	301.4(12):BT <sub>1</sub>	301.9(2):BT <sub>1</sub>	301.4(12):BT <sub>1</sub>	300.9(20):BT <sub>1</sub>		
			Used-Budgets	301.4(9):BT <sub>1</sub>	301.5(6):BT <sub>1</sub>	301.1(17):BT <sub>1</sub>	300.4(21)		
			Equal	301.3(15):BT <sub>1</sub>	301.2(16):BT <sub>1</sub>	301.8(3):BT <sub>1</sub>	301.6(4):BT <sub>1</sub>		
<i>catwatch</i>	974.6(22)	974.7(21)	Actions	981.8(10):BT <sub>1</sub> ,BT <sub>2</sub>	982.4(8):BT <sub>1</sub> ,BT <sub>2</sub>	981.1(14)	977.4(19)	75%	70%
			Archive	981.5(12):BT <sub>1</sub> ,BT <sub>2</sub>	977.2(20):BT <sub>1</sub> ,BT <sub>2</sub>	981.9(9):BT <sub>1</sub> ,BT <sub>2</sub>	981.1(14):BT <sub>1</sub> ,BT <sub>2</sub>		
			ConArchive	1007.4(2):BT <sub>1</sub> ,BT <sub>2</sub>	1002.5(4):BT <sub>1</sub> ,BT <sub>2</sub>	<b>1014.1(1):BT<sub>1</sub>,BT<sub>2</sub></b>	1004.6(3):BT <sub>1</sub> ,BT <sub>2</sub>		
			Used-Budgets	981.1(13)	981.5(11):BT <sub>1</sub> ,BT <sub>2</sub>	979.2(18):BT <sub>1</sub>	979.6(17)		
			Equal	985.2(5):BT <sub>1</sub> ,BT <sub>2</sub>	984.3(6):BT <sub>1</sub> ,BT <sub>2</sub>	982.5(7):BT <sub>1</sub> ,BT <sub>2</sub>	979.8(16)		
<i>features-service</i>	539.6(22)	628.7(21)	Actions	704.5(3):BT <sub>1</sub> ,BT <sub>2</sub>	702.9(15):BT <sub>1</sub> ,BT <sub>2</sub>	703.1(14):BT <sub>1</sub> ,BT <sub>2</sub>	702.4(19):BT <sub>1</sub> ,BT <sub>2</sub>	100%	100%
			Archive	704.3(5):BT <sub>1</sub> ,BT <sub>2</sub>	703.7(9):BT <sub>1</sub> ,BT <sub>2</sub>	702.4(20):BT <sub>1</sub> ,BT <sub>2</sub>	703.5(10):BT <sub>1</sub> ,BT <sub>2</sub>		
			ConArchive	704.4(4):BT <sub>1</sub> ,BT <sub>2</sub>	703.9(7):BT <sub>1</sub> ,BT <sub>2</sub>	704.6(2):BT <sub>1</sub> ,BT <sub>2</sub>	703.7(8):BT <sub>1</sub> ,BT <sub>2</sub>		
			Used-Budgets	703.4(12):BT <sub>1</sub> ,BT <sub>2</sub>	703.3(13):BT <sub>1</sub> ,BT <sub>2</sub>	<b>705.1(1):BT<sub>1</sub>,BT<sub>2</sub></b>	702.8(17):BT <sub>1</sub> ,BT <sub>2</sub>		
			Equal	703.5(10):BT <sub>1</sub> ,BT <sub>2</sub>	704.2(6):BT <sub>1</sub> ,BT <sub>2</sub>	702.8(16):BT <sub>1</sub> ,BT <sub>2</sub>	702.6(18):BT <sub>1</sub> ,BT <sub>2</sub>		
<i>proxyprint</i>	1515.8(3)	1513.6(5)	Actions	1511.3(9)	1510.2(15)	1506.8(18)	1500.7(21)	0%	15%
			Archive	1510.8(13)	1512.3(7)	1505.3(19)	1500.8(20)		
			ConArchive	1515.6(4):BT <sub>2</sub>	<b>1519.0(1):BT<sub>2</sub></b>	1516.0(2):BT <sub>2</sub>	1512.5(6)		
			Used-Budgets	1512.0(8)	1511.1(11)	1510.9(12)	1498.6(22)		
			Equal	1511.2(10)	1510.3(14)	1507.3(17)	1507.3(16)		
<i>scout-api</i>	1431.7(22)	1869.8(15)	Actions	1865.7(18):BT <sub>1</sub>	1882.8(12):BT <sub>1</sub>	1898.2(7):BT <sub>1</sub> ,BT <sub>2</sub>	<b>1916.5(1):BT<sub>1</sub>,BT<sub>2</sub></b>	100%	55%
			Archive	1866.2(17):BT <sub>1</sub>	1882.2(13):BT <sub>1</sub> ,BT <sub>2</sub>	1908.5(6):BT <sub>1</sub> ,BT <sub>2</sub>	1911.0(4):BT <sub>1</sub> ,BT <sub>2</sub>		
			ConArchive	1855.5(21):BT <sub>1</sub>	1882.1(14):BT <sub>1</sub>	1885.9(8):BT <sub>1</sub> ,BT <sub>2</sub>	1910.1(5):BT <sub>1</sub> ,BT <sub>2</sub>		
			Used-Budgets	1860.8(20):BT <sub>1</sub>	1869.0(16):BT <sub>1</sub>	1884.0(10):BT <sub>1</sub>	1912.9(3):BT <sub>1</sub> ,BT <sub>2</sub>		
			Equal	1865.2(19):BT <sub>1</sub>	1883.9(11):BT <sub>1</sub> ,BT <sub>2</sub>	1885.1(9):BT <sub>1</sub> ,BT <sub>2</sub>	1913.3(2):BT <sub>1</sub> ,BT <sub>2</sub>		
Average rank	14	10	Actions	9	8	10	11	75%	51%
			Archive	9	8	9	9		
			ConArchive	7	<b>4</b>	<b>4</b>	6		
			Used-Budgets	10	8	8	12		
			Equal	9	8	7	8		

Friedmantest :  $\chi^2=36.66928$ ,  $p$ -value=0.01836624Note that BT<sub>1</sub> means better than Base1, and BT<sub>2</sub> means better than Base2.**RQ1: Our novel techniques significantly improve performance in 6 out of the 7 case studies.**

To investigate the best configuration among all settings, first, we conducted the Friedman test for variance analysis on ranks. The result is statistically significant as shown in Table 5. Then, we selected the two best configurations:  $S = \text{ConArchive}$  with  $P_s=0.5$  and  $S = \text{ConArchive}$  with  $P_s=0.75$ , based on the best average rank (i.e., 4) among all case studies. Then, by further checking the probability column,  $P_s=0.5$  has a better rank on average. Thus, we choose as best configuration:  $S = \text{ConArchive}$  with  $P_s=0.5$ .

To assess the improvement with our approach, Table 6 shows the results of comparing MIO using such configuration with the baselines. In the table, detailed results on Vargha-Delaney effect sizes,  $p$ -values of statistical tests, and relative improvement are reported. First, we analyze the case studies which have high percentage of independent resources (Table 3), we target *rest-scs* (11/11 = 100%), *catwatch* (11/13=85%), and *rest-ncs* (6/6 = 100%). For *rest-scs*

**Table 6: A detailed comparison of our approach (using  $S = \text{ConArchive}$ ,  $P_s = 0.5$ ) with the baselines. Values in bold means our approach is better than the baseline, i.e.,  $\hat{A}_{12} > 0.5$  and  $p$ -values  $< 0.05$ .**

SUT	Base1			Base2		
	$\hat{A}_{12}$	p-value	relative	$\hat{A}_{12}$	p-value	relative
<i>rest-ncs</i>	<b>0.60</b>	<b>&lt;0.01</b>	<b>+0.2%</b>	<b>0.66</b>	<b>&lt;0.01</b>	<b>+0.4%</b>
<i>rest-scs</i>	<b>1.00</b>	<b>&lt;0.01</b>	<b>+42.4%</b>	<b>1.00</b>	<b>&lt;0.01</b>	<b>+22.9%</b>
<i>rest-news</i>	<b>0.63</b>	<b>&lt;0.01</b>	<b>+0.7%</b>	0.48	0.36	-0.0%
<i>catwatch</i>	<b>0.82</b>	<b>&lt;0.01</b>	<b>+2.9%</b>	<b>0.83</b>	<b>&lt;0.01</b>	<b>+2.9%</b>
<i>features-service</i>	<b>1.00</b>	<b>&lt;0.01</b>	<b>+30.4%</b>	<b>1.00</b>	<b>&lt;0.01</b>	<b>+12.0%</b>
<i>proxyprint</i>	0.48	0.65	+0.2%	<b>0.59</b>	<b>&lt;0.01</b>	<b>+0.4%</b>
<i>scout-api</i>	<b>1.00</b>	<b>&lt;0.01</b>	<b>+31.5%</b>	0.53	0.11	+0.7%

and *catwatch*, MIO with the chosen configuration has a significant improvement based on the low  $p$ -value (i.e., less than 0.01) and high effect size (i.e., more than 0.82). Regarding *rest-ncs*, the results still indicate an improvement, but the effect size and relative improvement are modest.

Regarding the results of *rest-news* shown in Table 6, we made a small improvement compared with Base1, and no improvement compared with Base2. But, as can be seen in Table 5, the number of covered targets does not vary much among the different techniques (i.e., the best has just 1.6 more targets than the worst). These results might be explained by considering the small size of resources (see Table 3) and simple implementation of that case study.

Regarding *features-service*, the result indicates a clear and significant improvement compared with both Base1 and Base2. By checking APIs of the case study, we found that there exist strongly connected relationships among resource URIs. These strong improvements might depend on such relationships between resources by seeking a better combination of multiple resources.

Regarding *scout-api*, by comparing with Base1, our approach obtained a significantly better performance with low  $p$ -value (i.e., less than 0.01) and high effect size (i.e., 0.82). This gives a hint that manipulation of resources works for *scout-api*. But there does not exist any improvement by comparing with another resource-related solution, i.e., Base2. Therefore, we took a look at its detailed APIs. 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, such constraint is not formally defined in the 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. These issues may limit the current effectiveness of our novel resource-based solutions.

Regarding *proxyprint*, the performance of MIO with the selected configuration is statistically equal with Base1, and statistically better than Base2. This exposes that making required resources available for the following actions does not work well for this case study. By further checking its API specification, we found that such SUT does not follow the REST constraints: (1) 15 resource URIs start with `/consumer`, but a creation of a consumer is under `/consumer/register`, and (2) 22 resource URIs start with `/printshops`, but a creation of a printshop is under `/request/register`. In this case, our solution actually fails to handle resources on these 37 (15+22) endpoints. This may be a main reason that there is no improvement compared with random sampling. To handle this issue, it requires further semantic analysis on the resource URIs.

**RQ2: MIO, enhanced with the ConArchive-based strategy (with a 50% probability) and resource-based mutation, performs significantly better (up to 42%) on SUTs that use independent, or clearly connected, resources.**

## 6 THREATS TO VALIDITY

*Conclusion validity.* Due to randomness in search algorithms, results may be significantly affected by chance. We handled this threat

by repeating each experiments 40 times. Based on the standard guideline [8] to report search-based software engineering experiments, 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 [1], the same stopping criterion must be applied for the algorithms to avoid any potential bias in results. In the experiment, we set the same time budget (i.e., 100k HTTP calls) to deal with this type of validity threat.

*Internal validity.* The internal validity is that we used our implementation, that is a tool prototype, to conduct our experiments. We cannot guarantee that our implementation is bug free.

*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 and four real REST APIs. The fact that only seven 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 REST APIs.

## 7 CONCLUSIONS

In recent years, application of REST for building web services is 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 proposed a resource-based approach to improve search-based test case generation for testing RESTful web services. Our approach takes advantages of the MIO algorithm and EvoMASTER. We designed resource-based sampling with five smart sampling strategies, and resource-based mutation, to exploit domain knowledge of REST on the handling of HTTP resources. We compare our approach with existing work on seven open-source RESTful APIs from a benchmark for conducting experiments on automated system testing for web/enterprise applications. Based on our results, our best strategy has an overall best performance among all case studies. However, the improvement is not always significant.

In the future, we plan to conduct additional experiments with more case studies, and further study generalization of our approach. In addition, to reduce failures in handling resources (e.g., create), we plan to investigate novel solutions for doing a more intelligent analysis of resources of web services that are not fully following the REST guidelines. Furthermore, we would like to further investigate better strategies for more specialized and effective mutation operators.

EvoMASTER and the employed case study are freely available online on GitHub. To learn more about EvoMASTER, visit our webpage at [www.evomaster.org](http://www.evomaster.org)

## ACKNOWLEDGMENTS

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



## REFERENCES

- [1] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36, 6 (2010), 742--762.
- [2] Subbu Allamaraju. 2010. *Restful web services cookbook: solutions for improving scalability and simplicity*. " O'Reilly Media, Inc."
- [3] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175--203. <https://doi.org/10.1002/stvr.v16:3>
- [4] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
- [5] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology (IST)* 104 (2018), 195--206.
- [6] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [7] A. Arcuri and L. Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 265--275.
- [8] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.
- [9] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2013. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability (STVR)* 23, 4 (2013), 261--313.
- [10] Gerardo Canfora and Massimiliano Di Penta. 2009. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, 78--105.
- [11] Sujit Kumar Chakrabarti and Prashant Kumar. 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, 302--308.
- [12] Sujit Kumar Chakrabarti and Reswin Rodriguez. 2010. Connectedness testing of restful web-services. In *Proceedings of the 3rd India software engineering conference*. ACM, 143--152.
- [13] S. Droste, T. Jansen, and I. Wegener. 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*. 13--22.
- [14] Tobias Fertig and Peter Braun. 2015. Model-driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1497--1502.
- [15] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Ph.D. Dissertation. University of California, Irvine.
- [16] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
- [17] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*. ACM, 77--78.
- [18] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-Based Testing of RESTful Web Services Using UML Protocol State Machines. In *Brazilian Workshop on Systematic and Automated Software Testing*.
- [19] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering (EMSE)* 22, 2 (2017), 852--893.
- [20] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2017. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering (TSE)* (2017).