Improving Test Case Generation for REST APIs Through Hierarchical Clustering

Stallenberg, Dimitri; Olsthoorn, Mitchell; Panichella, Annibale

DOI
10.1109/ASE51524.2021.9678586

Publication date
2021

Document Version
Accepted author manuscript

Published in
2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Improving Test Case Generation for REST APIs Through Hierarchical Clustering

Dimitri Stallenberg
Delft University of Technology
Delft, The Netherlands
D.M.Stallenberg@tudelft.nl

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

***Abstract*—With the ever-increasing use of web APIs in modern-day applications, it is becoming more important to test the system as a whole. In the last decade, tools and approaches have been proposed to automate the creation of system-level test cases for these APIs using evolutionary algorithms (EAs). One of the limiting factors of EAs is that the genetic operators (crossover and mutation) are fully randomized, potentially breaking promising patterns in the sequences of API requests discovered during the search. Breaking these patterns has a negative impact on the effectiveness of the test case generation process. To address this limitation, this paper proposes a new approach that uses Agglomerative Hierarchical Clustering (AHC) to infer a linkage tree model, which captures, replicates, and preserves these patterns in new test cases. We evaluate our approach, called LT-MOSA, by performing an empirical study on 7 real-world benchmark applications *w.r.t.* branch coverage and real-fault detection capability. We also compare LT-MOSA with the two existing state-of-the-art white-box techniques (MIO, MOSA) for REST API testing. Our results show that LT-MOSA achieves a statistically significant increase in test target coverage (*i.e.,* lines and branches) compared to MIO and MOSA in 4 and 5 out of 7 applications, respectively. Furthermore, LT-MOSA discovers 27 and 18 unique real-faults that are left undetected by MIO and MOSA, respectively.**

***Index Terms*—system-level testing, test case generation, machine learning, search-based software engineering**

I. Introduction

Over the last decade, the software landscape has been characterized by the shift from large monolithic applications to component-based systems, such as microservices. These systems, together with their many diverse client applications, make heavy use of web APIs for communication. Web APIs are almost ubiquitous today and rely on well-established communication standards such as SOAP [1] and REST [2]. The shift towards component-based systems makes it ever-increasingly more important to test the system as a whole since many different components have to work together. Manually writing system-level test cases is, however, time-consuming and error-prone [3], [4].

For these reasons, researchers have come up with different techniques to automate the generation of test cases. One class of such techniques is search-based software testing. Recent advances have shown that search-based approaches can achieve a high code coverage [5], also compared to manually-written test cases [6], and are able to detect unknown bugs [7], [8], [9]. Search-based test case generation uses Evolutionary Algorithms (EAs) to evolve a pool of test cases through randomized *genetic operators*, namely *mutations* and *crossover/recombination*. More precisely, test cases are encoded as chromosomes, while statements (*i.e.,* method calls) and test data are encoded as the genes [10]. In the context of REST API testing, a test case is a sequence of API requests (*i.e.,* HTTP requests and SQL commands) on specific resources [11], [9].

Listing 1: Motivating example of patterns in API requests.

```
1 POST   authenticate?user=admin&password=pwd
2 POST   product?id=1&price=10.99&token={key}
3 UPDATE product/1?price=8.99&token={key}
4 GET    product/1?token={key}
```

REpresentational State Transfer (REST) APIs deal with states. Each individual request changes the state of the API, and therefore, its execution result depends on the state of the application (*i.e.,* the previously executed requests). Listing 1 shows an example of HTTP requests made to a REST API that manages `products`. In the example, the first request authenticates the client to the API with the given `username` and `password`. In return, the client receives a `token` that can be used to make subsequent requests. The second request creates a new product by specifying the `id`, `price`, and the `token`. The `price` is then updated in the third request and the changes are retrieved in the last request.

The example above contains patterns of HTTP requests that strongly depend on the previous ones. The `GET` request can not retrieve a product that does not exist, and therefore, can not be successfully executed without request 2. Similarly, the `UPDATE` request can not be executed before the product is created. Lastly, request 2, 3, and 4, all depend on request 1 for the authentication token. Hence, HTTP requests should not be executed in any random order [12].

Test generation tools rely on EAs to build up sequences of HTTP requests iteratively through genetic operators, *i.e.,* crossover and mutation [9], [11], [13]. While these operators can successfully create promising sequences of HTTP requests, they do not directly recognize and preserve them when creating new test cases [14]. For example, the genetic operators may remove request 2 from the test case in Listing 1, breaking requests 3 and 4 unintentionally.

In this paper, we argue that detecting and preserving patterns

of HTTP requests, hereafter referred to as *linkage structures*, improves the effectiveness of the test case generation process. We propose a new approach that uses Agglomerative Hierarchical Clustering (AHC) to infer these linkage structures from automatically generated test cases in the context of REST APIs testing. In particular, AHC generates a *Linkage Tree* (LT) model from the test cases that are the closest to reach uncovered test targets (*i.e.,* lines and branches). This model is used by the genetic operators to determine which sequences of HTTP requests should not be broken up and should be replicated in new tests.

To evaluate the feasibility and effectiveness of our approach, we implemented a prototype based on EvoMaster, the state-of-the-art test case generation tool for Java-based REST APIs. We performed an empirical study with 7 benchmark web/enterprise applications from the EvoMaster Benchmark (EMB) dataset. We compared our approach against the two state-of-the-art algorithms for system-level test generations implemented in EvoMaster, namely Many Independent Objective (MIO) and Many Objective Search Algorithm (MOSA).

Our results show that LT-MOSA covers significantly more test targets in 4 and 5 out of the 7 applications compared to MIO and MOSA, respectively. On average, LT-MOSA, covered $11.7\,\%$ more test targets than MIO (with a max improvement of $66.5\,\%$) and $8.5\,\%$ more than MOSA (with a max improvement of $37.5\,\%$). Furthermore, LT-MOSA could detect, on average, 27 and 18 unique real-faults that were not detected by MIO and MOSA, respectively.

In summary, we make the following contributions:

1) A novel approach that uses Agglomerative Hierarchical Clustering to learn and preserve *linkage structures* embedded in REST API.
2) An empirical evaluation of the proposed approach with the two state-of-the-art algorithms (MIO and MOSA) on a benchmark of 7 web/enterprise applications.
3) A full replication package including code and results [15].

The remainder of this paper is organized as follows. Section II summarizes the related work and background concepts. Section III introduces our approach called, LT-MOSA, and gives a detailed breakdown of how it works. Section IV describes the setup of our empirical study. Section V discusses the obtained results and presents our findings. Section VI discusses the threats to validity and Section VII draws conclusions and identifies possible directions for future work.

II. Background and Related Work

This section provides an overview of basic concepts and related work in search-based software testing, REST API testing, test case generation, and linkage learning.

A. Search-based software testing

Search-based software testing has become a widely used and effective method of automating the generation of test cases and test data [16], [17]. Automatic test case generation significantly reduces the time needed for testing and debugging applications (*e.g.,* [18]) and it has been successfully used in industry (*e.g.,* [19], [20]). Popular tools for automatically generating test cases include EvoSuite [21], for unit testing, and Sapienz [20], [22], for Android testing.

Evolutionary Algorithms (EAs) are one of the most commonly used class of meta-heuristics in search-based testing. EAs have been used to generate both test data [16] and test cases [10]. The latter includes test data, method sequences and assertions. EAs are inspired by the biological process of evolution. It initializes and evolves a population of randomly generated individuals (test cases). These individuals are then evaluated based on a predefined fitness function. The individuals with the best fitness value are *selected* for reproduction through *crossovers* (swapping elements between two individuals) and random *mutations* (small changes to individuals). The offspring test cases resulting from the reproduction are then *evaluated*. Finally, the population for the next generation is created by selecting the best individuals across the previous population and the newly generated tests (*elitism*). This loop (reproduction, evaluation, and selection) continues until a stopping condition has been reached. The final test suite is created based on the population's best individuals.

B. REST API Testing

A REpresentational State Transfer (REST) API is oriented around resources. This differs from a command-oriented API like for example the Remote Procedure Call (RPC) standard. A REST API request performs an action on a specific resource. These actions are encoded by the different methods defined in the HTTP protocol. Common HTTP methods include `GET`, `HEAD`, `POST`, `PUT`, `PATCH`, and `DELETE`. These actions are performed on an endpoint, which is the location of the resource. An example of this would be performing a `GET` action on the `/user/3` endpoint to retrieve the information of a user with a user id of 3. Another example would be performing a `POST` action on the `/user/` endpoint to create a new user.

With the recent rise in popularity of REST APIs in the last decade, it is becoming more important to test this critical communication layer. There are two different ways system-level API testing can be approached: black-box and white-box testing. Black-box testing frameworks (*e.g.,* RESTTESTGEN [23], EvoMaster black-box [11]) examine the functionality of the system without looking at the internals of the system.

In contrast, white-box testing approaches rely on the internal structure of the system and measure the adequacy of the tests based on coverage criteria (*e.g.,* branch coverage). This allows the algorithm to easily identify which paths have been covered and which have not. Prior studies show white-box techniques achieve better results than their black-box counterparts [11]. Additionally, white-box techniques allow to integrate SQL databases in the test case generation process [9].

C. Test Case Generation for REST APIs

EvoMaster is a tool that aims to generate system-level test cases for REST APIs. It internally uses evolutionary

algorithms to evolve the test cases iteratively. At the time of writing, EvoMaster provides two EAs. The first algorithm is the Many Independent Objective (MIO) algorithm proposed by Arcuri *et al.* [13]. The second algorithm is a variant of the Many-Objective Sorting Algorithm (MOSA) proposed by Panichella *et al.* [24]. Both of these algorithms are specifically designed for test case generation and consider the peculiarities of these systems. The pseudo-code of these algorithms can be found in the respective papers.

1) MIO: The Many Independent Objective (MIO) algorithm is an evolutionary algorithm that aims to improve the scalability of many-objective search algorithms for programs with a very large number of testing targets (in the order of millions) [13]. It works under the assumption that: (i) each target can be independently optimized; (ii) targets can be strongly related, for example when nested, or completely independent; (iii) not all targets can be covered. Based on these assumptions, MIO maintains a separate population for each target of the System Under Test (SUT). The fitness function for each population consists solely of the objective of that population. So, each population compares and ranks the individuals based on a single testing target. At the start of the algorithm, all populations are empty. Each iteration, the algorithm either samples a random new test case with a certain probability or it samples a test case from one of the populations with an uncovered target. This test case is then added to all populations with an uncovered target and evaluated independently. The population that is chosen when a test case is sampled from one of the populations, is based on the number of times a test case has been sampled from that population before. Each time a population is sampled, a counter is incremented. If a test case with a better fitness value is added to the population, the counter is reset to zero. The sampling mechanism chooses the population with the lowest counter. This makes sure that the algorithm won't get stuck on an unreachable target. When a population reaches a certain predefined size, it removes the test case with the worst fitness value. At the end of the algorithm, a test suite is built with the best test case from each population.

2) MOSA: The Many Objective Sorting Algorithm (MOSA) is an evolutionary algorithm that focuses on optimizing multiple objectives (*e.g.,* branches) at the same time [24]. It adapts the NSGA-II algorithm, which is one of the most popular multi-objective search algorithms [25]. In MOSA, a test case is represented as a chromosome. Each testing target (*e.g.,* branch, line) in the code corresponds to a separate objective measuring the distance (of a given test) toward reaching that target. The fitness of the test cases is measured according to a vector of scalar values that represent these different objectives. Since MOSA has many different objectives, it uses two preference criteria to determine which test cases should be selected and evolved first: (i) minimal distance to the uncovered target; (ii) test case length. More precisely, it first looks for the subset of the Pareto front that contains test cases with a minimum distance for each uncovered objective. When multiple test cases are equally close to cover the same target, the smallest test case will be selected. In each generation, an archive collects the test cases that cover previously uncovered targets. The archive is updated every time a newly generated test case covers new targets or covers already covered targets but with fewer statements.

3) Comparison: Both MIO and MOSA produce good results in both unit and system-level tests. In the context of system-level testing, Arcuri [13] showed that MIO achieves the best average results, but there are web/enterprise applications in which MOSA achieves higher coverage. In unit testing, Campos *et al.* [5] showed that MOSA (and its variants) achieves overall better coverage than MIO. Therefore, in this paper, we consider both MIO and MOSA as they excel in different scenarios and are the state-of-the-art in test case generation for REST APIs.

Notice that an extension of MOSA, called DynaMOSA [26], has been proposed in the related literature for unit testing. Compared to MOSA, DynaMOSA organizes the coverage targets (*e.g.,* branches) of a given code unit into a global hierarchy based on their structural dependencies. Then, the list of search objectives is updated dynamically based on their structural dependencies and the previously covered targets. While previous studies in unit-testing showed that DynaMOSA outperforms its predecessor MOSA [26], [5], it cannot be applied for REST APIs as no global hierarchy exists across the coverage targets of different microservices or functions/classes within the same microservice[1].

4) Chromosome Representation: Test cases in both search algorithms included in EvoMaster are represented by two genes: action gene and input gene. The action gene represents the structure and order of the HTTP requests in the test case. EvoMaster extracts these actions from the Swagger/OpenAPI documentation that has to be provided for each system under test. An action gene consists of the HTTP method and the REST endpoint. An example of an action gene would be `POST /authentication`.

The input gene represents the input data for the HTTP request. An example of this input data would be the username and password that are required by the `/authentication` endpoint. This input data is sampled from the source code of the SUT.

D. Linkage Learning in EAs

Linkage-learning refers to a large body of work in the evolutionary computation community that aims to infer *linkage structures* present in promising individuals [27]. Linkage structures are groups of "good" genes that contribute to the fitness of a given population. Accurate inference of *linkage structures* has been used to design "competent" genetic operators [28] for numerical problems. These operators are designed to replicate rather than break groups of genes (patterns) into the offspring.

To learn linkage structures from numerical chromosomes, researchers have used different unsupervised machine learning algorithms. BOA [29] constructs a Bayesian Network and

[1]Micro-services are loosely coupled and deployable independently.

creates new numerical chromosomes using the joint distribution encoded by the network. DSMGA [30] uses Dependency Structure Matrix (DSM) clustering and applies the crossover by exchanging gene clusters between parent chromosomes. 3LO [31] employs local optimization as an alternative method for linkage learning.

Two state-of-the-art EAs for numerical problems are LT-GA [32] and GOMEA [33]. Both algorithms use clustering to infer linkage-trees, representing the linkage structures between genes (problem variables) using tree-like structures. GOMEA uses agglomerative hierarchical clustering as a faster and more efficient way to learn linkage-trees [33]. GOMEA uses the *gene-pool optimal mixing* to create new solutions by applying a local search within the recombination procedure. More precisely, it creates offspring solutions from one single parent by iteratively replicating (copying) gene clusters from different donors. In each iteration, the new solution is evaluated; if its fitness improves, the replicated genes are kept; otherwise, the change is reverted.

Linkage models have been successfully applied to evolutionary algorithms for numerical [34], permutation [35], and binary optimization problems [32], [36] with fixed length chromosomes. However, test cases for REST APIs are characterized by a more complex structure [11]: each test is a sequence of HTTP requests towards a RESTful service, each with input data, such as HTTP headers, URL parameters, and payloads for `POST`/`PUT`/`PATCH` methods. Besides, a test case might include SQL data and commands for microservices that use databases [9]. Finally, test cases have a variable size, and their lengths can also vary throughout the generations. Therefore, we need to tailor existing *linkage learning* methods according to the test case characteristics discussed above.

III. Approach

This section presents our approach, called LT-MOSA, for system-level test cases generation and that incorporates and tailors linkage learning into MOSA [24]. We selected MOSA as the base algorithm to apply linkage learning because it evolves a single population of test cases, which is a requirement for the learning process. Additionally, MOSA has been proved to be very competitive in the context of RESTful API testing [13], unit testing [37], [5], and DNN testing [38].

Algorithm 1 outlines the pseudo-code of LT-MOSA. The parts where LT-MOSA deviates from MOSA are highlighted with a blue color. LT-MOSA starts with initializing the population P and computing the corresponding objective scores (line 2). Each test case is composed of HTTP calls (*actions*) and SQL commands (*database actions*) [9]. The RANDOM-POPULATION function also executes the generated tests and computes their objective scores using the *branch distance* [39]. The *branch distance* is a well-known heuristic in search-based testing to measure how far each test case is from reaching a given coverage target (*e.g.,* branch). Then, the test cases are sorted in sub-dominance fronts using the *preference sorting algorithm* [24], in line 4. The test cases within the first front (Front[0]) are the closest ones in P to reach the coverage

Algorithm 1: LT-MOSA

```
   Input:
   Coverage targets Ω = {ω_1, ..., ω_n}
   Population size M
   Frequency K for updating the linkage tree model
   Result: A test suite T
1  begin
2      P ⟵ RANDOM-POPULATION(M)
3      archive ⟵ UPDATE-ARCHIVE(∅, P)
4      Fronts ⟵ PREFERENCE-SORTING(R)
5      while not (stop_condition) do
6          L ⟵ LEARN-LINKAGE-MODEL(Fronts[0], K)
7          P' ⟵ ∅
8          for index = 1..M do
9              parent ⟵ TOURNAMENT-SELECTION(P)
10             if apply_recombination then
11                 donor ⟵ TOURNAMENT-SELECTION(P)
12                 offspring ⟵ LINKAGE-RECOMB(parent,
                     donor, L)
13                 offspring ⟵ MUTATION(offspring)
14             else
15                 offspring ⟵ MUTATION(parent)
16             P' ⟵ P' ⋃ {offspring}
17             archive ⟵ UPDATE-ARCHIVE(archive, offspring)
18         R ⟵ P ⋃ P'
19         Fronts ⟵ PREFERENCE-SORTING(R)
20         P ⟵ ENVIRONMENTAL-SELECTION(Fronts, M)
21     T ⟵ archive
```

targets and, therefore, the fittest individuals to consider for model learning.

Afterwards, the population P is evolved through subsequent generations within the loop in lines 5-20. Each generation starts by training a *linkage tree model* on the first non-dominated front (line 6) with the goal of learning patterns of HTTP and SQL actions that strongly contribute to the "optimality" of the population. We discuss the learning procedure in detail in Section III-A. Once the *linkage tree model* is obtained, LT-MOSA selects the *fittest* test cases using the *tournament selection* (line 9 and 11) and creates an offspring population P' by using a *linkage-based recombination* [33] (line 12) and *mutation* [9] (line 13 and 15). The *linkage-based recombination* is a specialized *crossover* that relies on the *linkage tree model* to decide which patterns of genes (HTTP requests) can be copied into the offspring test cases. We describe the *linkage-based recombination* operator in Section III-B.

LT-MOSA adds the newly generated tests into the offspring population (line 16), executes them, and updates the archive (line 17) in case new coverage targets have been reached. The generation ends by selecting the best M test cases across the existing population P and the offspring population P'. This selection is made by combining the two population into one single pool R of size $2 \times M$ (line 18), applying the *preference sorting* (line 19), and selecting M solutions from the non-dominated fronts starting from Front[0] until reaching the population size M (line 20).

The search stops when the termination criteria are met (condition in line 5), the final test suite will then be composed of all test cases that have been stored in the *archive* throughout

the search. Note that LT-MOSA updates the *archive* in each generation by storing the shortest test case covering each target ω_i. Finally, the list of objectives is updated such that the search focuses only on the targets (branches) that are left uncovered.

In the following sub-sections, we detail the key novel ingredients in LT-MOSA, namely the *linkage model learning* (LT) (Section III-A), the *linkage-based recombination operator* (Section III-B), and the *mutation* operator (Section III-C).

A. Linkage tree learning

In this section, we describe the main changes we applied to the traditional *linkage learning* and adapt it to our context, *i.e.,* test case generation for RESTful APIs.

1) Linkage Encoding: The first problem we had to solve is encoding the test cases into discrete vectors of equal length, which can be interpreted and analyzed via hierarchical clustering. To this aim, we opted for *encoding* test cases as binary vectors whose entries denote the presence (or not) of the possible HTTP requests. Given an SUT (software under tests), there are N possible HTTP requests to the available APIs. This information can be extracted from the Swagger/OpenAPI definition [11], which is a widely-used tool for REST API documentation. A Swagger definition contains the HTTP operations available for each API endpoint. Each operation contains both fixed and variable parts. The fixed part includes the type of operation (`POST`, `GET`, `PUT`, `PATCH`, and `DELETE`), the IP address or the URL of the target API, and the HTTP headers. For the variable part, the Swagger definition includes information about the input data (*e.g., string*, *double*, *date*, etc.) that can vary. Therefore, for each API endpoint, we identify the available HTTP operations, hereafter referred to as *actions*, by parsing the Swagger definition.

Let $\mathcal{S} = \{S_1, \ldots, S_N\}$ be the set of N *HTTP actions* available for the target SUT. We encode each test case T as a binary string of size N as follows:

$$E(T) = \langle e_1, \ldots, e_N \rangle \;\; with \;\; e_i = \begin{cases} 0 & if\ S_i \notin T \\ 1 & if\ S_i \in T \end{cases} \qquad (1)$$

In other words, each element e_i in the encoded vector $E(T)$ is set to 1 if the test case T contains the *action* S_i; 0 otherwise. The linkage model is trained on the binary-coded vectors rather than on the original test cases. This encoding is used to determine, via statistical analysis, which group of HTTP *actions* often appear together within the fittest test cases, and which ones never occur together. This information is used to create more efficient *recombination* operators.

2) Linkage Model Training: In this paper, we use *agglomerative hierarchical clustering* (AHC) over other techniques (e.g. Bayesian Networks) for linkage tree learning. This is because prior studies show AHC is more efficient [33]. In particular, we apply the UPGMA (unweighted pair group method with arithmetic mean) algorithm [40]. In each iteration, UPGMA merges two clusters that are most similar based on the average distance across the data points (genes in our case) in the two clusters. The similarity between two HTTP actions genes S_i and S_j is computed using the mutual information as suggested by Thierens and Bosman [33]:

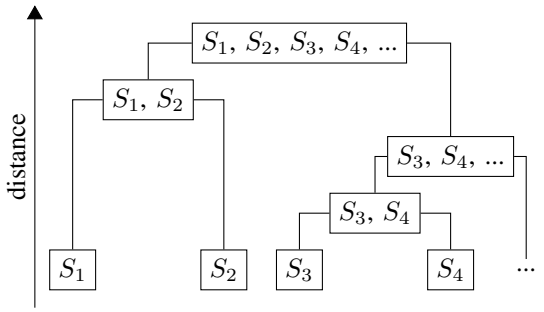

Fig. 1: Example of *linkage tree* model and the Family Of Subset (FOS)

$$MI(S_i, S_j) = H(S_i) + H(S_j) - H(S_i, S_j) \qquad (2)$$

where $H(.)$ denotes the information entropy [41].

Note that LT-MOSA infers the *linkage tree* for the most promising part of the population, *i.e.,* the first non-dominated front (line 6 in Algorithm 1). Furthermore, the training process is applied to the encoded test cases according to the schema described in Section III-A1 rather than on the actual test cases. Hence, the *linkage tree* obtained with UPGMA captures the hierarchical relationship between HTTP actions in our case.

For example, let us consider the *linkage tree* depicted in Fig. 1. In the example, the set of actions $\mathcal{S} = \{S_1, S_2, S_3, S_4, \ldots\}$ (the root of the tree) are partitioned into two clusters: S_1, S_2 and $S_3, S_4, \ldots$; each sub-cluster can be further divided in sub-cluster until reaching the leaf node. In general, the linkage tree has N leaves and $N-1$ internal nodes. The root node contains all HTTP actions of the SUT. Each internal node divides the set of HTTP actions into two mutually exclusive clusters (the child nodes). Finally, the leaves contain the individual HTTP actions, which are the starting point of the UPGMA algorithm.

The *linkage tree* nodes are often referred to as *Family of Subsets* ($\mathcal{F}$) in the related literature [32], [33]. Each node (or subset) $\mathcal{F}' \in \mathcal{F}$ with $|\mathcal{F} > 2|$ has two mutually exclusive subsets (or child nodes) $\mathcal{F}_x$ and $\mathcal{F}_y$ such that $\mathcal{F}_x \bigcap \mathcal{F}_y = \emptyset$ and $\mathcal{F}_x \bigcup \mathcal{F}_y = \mathcal{F}'$. Each subset $\mathcal{F}' \in \mathcal{F}$ represents a cluster of HTTP actions that often appear together and characterized the best test cases in the population. Therefore, the recombination operator should be applied by preserving these subsets (patterns) when creating new offspring tests. The next subsection describes the subsets-preserving recombination operator we implemented in LT-MOSA.

The computation complexity of UPGMA is $O(N \times M^2)$, where N is the number of genes and M is the population size. To reduce its overhead, the linkage tree learning procedure is not applied in each generation. Instead, the linkage tree model is re-trained every K generations (line 6 of Algorithm 1).

B. Linkage-based Recombination

MOSA creates offspring tests using the *single-point crossover* [24]. This crossover operator is the classic *recombination operator* used in genetic algorithms [42], and test case generation [10], [21]. This operator generates two offsprings by randomly swapping statements between two parent tests T_1 and T_2. As argued in Section I, exchanging statements between test cases in a randomized manner can lead to breaking gene patterns (HTTP actions) that characterized the fittest individuals. Randomized recombination is also disruptive toward building good partial solutions (building blocks), negatively affecting the overall convergence [33].

Therefore, LT-MOSA uses a *linkage-based recombination operator* rather than the classical *single-point crossover* to preserve the patterns of HTTP actions identified by the linkage tree model. The *recombination operator* generates only one offspring starting from two existing test cases, called *parent* and *donor*. Both test cases are selected from the current population P as indicated in lines 9 and 11. The offspring is created by copying all genes (HTTP actions with input data) from the *parent* and further injecting only some genes from the *donor*. These genes are selected by exploiting the *linkage tree model* trained according to Section III-A.

More precisely, we first identify the gene patterns (*i.e.,* the subsets $\mathcal{F}' \in \mathcal{F}$) that the *donor* contains. This is done by iterating across all subsets in the linkage tree model $\mathcal{F}$ and identifying the subsets $\mathcal{F}' \subset \mathcal{F}$ that appear in the encoded vector (see Section III-A1) of the *donor*. LT-MOSA randomly selects one of the identified subsets in $\mathcal{F}'$ and inserts it into the offspring. The *injection point* is randomly chosen, and the selected genes (HTTP actions with test data) are inserted into the offspring in the exact order as they appear in the *donor*.

If the *donor* does not contain any subset according to the linkage tree (*i.e.,* $\mathcal{F}' = \emptyset$), then the offspring is generated by applying the traditional *single-point crossover*. This operator can be applied to the latter case since the *linkage tree model* could not identify any useful gene pattern within the *donor*.

C. Mutation

In MOSA, each test case is mutated with a probability $p_m = 1/L$, where L is the test case length [24]. This also reflects the existing guidelines in evolutionary computation [43], [44], which suggest using a mutation probability p_m proportional to the size of the chromosome.

In recent years, Arcuri [13] improved the *mutation operator* in the context of system-level test case generation by using a variable mutation rate. Indeed, the mutation operator in MIO [13] increases the number of mutations applied to each test case from 1 (start of the search) up to 10 (end of the search) with linear incremental steps. The importance of having a large mutation rate for RESTful API testing has also been confirmed by a recent study results [9].

Based on these observations, LT-MOSA uses the same mutation rate of MIO (*i.e.,* increasing mutation rate from 1 up to 10 mutations) rather than the fixed mutation rate of MOSA.

IV. Empirical Study

This section details the empirical study that we carried out to evaluate the effectiveness of the proposed solution, called LT-MOSA, and compare it with the state-of-the-art algorithms (MIO, MOSA) *w.r.t.* to the following testing criteria: (i) code (line and branch) coverage and (ii) *fault detection* capability.

A. Benchmark

This study uses the EvoMaster Benchmark (EMB)[2] version 1.0.1. This benchmark was specifically created as a set of web/enterprise applications for evaluating the test case generation algorithms implemented in EvoMaster. We selected this benchmark since it has been widely used in the literature to assess test case generation approaches for REST APIs [11], [13].

In this study, we used five real-world open-source Java web applications and two artificial Java web applications. *CatWatch* is a metrics dashboard for GitHub organizations. *Features-Service* is a REST Microservice for managing feature models of products. *OCVN* (Open Contracting Vietnam) is a visual data analytics platform for the Vietnam public procurement data. *ProxyPrint* is a platform for comparing and making requests to print-shops. *Scout-API* is a RESTful web service for the hosted monitoring service "Scout". *NCS* (Numerical Case Study) is an artificial application containing numerical examples. *SCS* (String Case Study) is an artificial application containing string manipulation code examples. We use *NCS* and *SCS* since they have been designed for assessing test generation tools. These artificial web applications allow to cover many different scenarios (e.g., deceptive branches [13]). Compared to previous studies [11], [13], we added the *OCVN* application as it is the largest real-world system in the benchmark. We additionally removed the *rest-news* application as it contains artifical examples that are used for classroom teaching.

Table I summarizes the main characteristics of the applications in the benchmark, such as the number of classes, the number of test coverage targets, and the number of endpoints included in the service. This benchmark contains a total of 1655 classes with around 20 000 test coverage targets and 440 endpoints, not including tests or third-party libraries.

EvoMaster requires a test driver for the application under test. This test driver contains a controller that is responsible for starting, resetting, and stopping the SUT. We used the test drivers available in the EMB benchmark for the web applications used in this study.

B. Research Questions

Our empirical evaluation aims to answer the following research questions:

RQ1 *How does* LT-MOSA *perform compared to the state-of-the-art approaches with regard to code coverage?*

RQ2 *How effective is* LT-MOSA *compared to the state-of-the-art approaches in detecting real-faults?*

[2]https://github.com/EMResearch/EMB/releases/tag/v1.0.1

TABLE I: Web applications from the EvoMaster Benchmark (EMB) used in the empirical study. Reports the number of Java classes, test coverage targets (*i.e.,* lines and branches), and the number of endpoints.

Application	Classes	Coverage Targets	Endpoints
CatWatch	69	2182	23
Features-Service	23	513	18
NCS	10	652	7
OCVN	548	8010	258
ProxyPrint	68	3758	74
Scout-API	75	3449	49
SCS	13	865	11
Total	1655	19 429	440

RQ3 *How effective is* LT-MOSA *at covering test targets over time compared to the state-of-the-art approaches?*

The first two research questions aim to evaluate if preserving patterns in HTTP requests through linkage learning can improve the *effectiveness* of test case generation for REST APIs by reaching a higher coverage and detecting more faults.

The last research question aims to answer if our approach, LT-MOSA, is more *efficient* in covering these test targets by measuring how many test targets are covered at different times within the search budget.

C. Baseline

To answer our research questions, we compare LT-MOSA with the two state-of-the-art search-based test case generation algorithm for REST APIs as a baseline:

- *Many Independent Objective (MIO)* is the state-of-the-art for REST API testing, and it is the default search algorithm in EvoMaster. MIO aims to improve the scalability of many-objective search algorithms for programs with a very large number of testing targets (see Section II-C1).
- *Many-Objective Sorting Algorithm (MOSA)* is the base algorithm we use to build and design LT-MOSA. Therefore, we want to assess that our approach outperforms its predecessor. Furthermore, MOSA has been proven to be very competitive in the context of REST APIs testing (see Section II-C2).

D. Prototype Tool

We have implemented LT-MOSA in a prototype tool that extends EvoMaster, an automated system-level test case generation framework. In particular, we implemented the approach as described in Section III within EvoMaster.

The variant of MOSA implemented in EvoMaster differs from the original algorithm proposed by Panichella *et al.* [24]. The EvoMaster variant does not use the crossover operator but merely relies on the mutation operator to create new test cases. Therefore, we implemented the *single-point* crossover as described in [24] and adapted it to the encoding schema used for representing REST API requests in EvoMaster. See Section II-C2 for more details.

We chose EvoMaster because it already implements the state-of-the-art test case generation algorithms, and it is publicly available on GitHub. Besides, EvoMaster implements *testability transformations* to improve the guidance for search-based algorithms [45] and can handle SQL databases [9].

E. Parameter Setting

For this study, we have chosen to adopt the default search algorithm parameter values set by EvoMaster. It has been empirically shown [46] that although parameter tuning has an impact on the effectiveness of a search algorithm, the default values, which are commonly used in literature, provide reasonable and acceptable results. Thus, this section only lists a few of the most important search parameters and their values:

1) Search Budget: We chose a search budget (stopping condition) based on time instead of the number of executed tests. This choice was made as search time provides the fairest comparison given that we consider different kinds of algorithms with diverse internal routines (also in terms of computational complexity). Additionally, practitioners will often only allocate a certain amount of time for the algorithm to run. The search budget for all algorithms was set to 30 minutes as this strikes a balance between giving the algorithms enough time to explore the search space and making the study infeasible to execute. If the algorithm has covered all its test objectives, it will stop prematurely. Note that running time is considered a less biased stopping condition than counting the number of executed tests since not all tests have the same running time [6], [9], [13], [21]. We further discuss this aspect in the threats to validity.

2) MIO *parameters:* For MIO, we used the default settings as provided in the original paper by Arcuri *et al.* [47], [13].

- *Population size:* We use the default population size of 10 individuals per testing target. Notice that MIO uses separate populations for the different targets.
- *Mutation:* We use the default number of applied mutations on sampled individuals, which linearly increases from 1 to 10 by the end of the search.
- *F:* We use the default percentage of time after which a focused search should start of 0.5.
- P_r*:* We use the default probability of sampling at random, instead of sampling from one of the populations, of 0.5. This value will linearly increase/decrease based on the consumed search budget and the value of F.

3) MOSA *parameters:* For MOSA, we used the default settings described in the original paper *et al.* [26].

- *Population size:* 50 individuals (test cases).
- *Mutation:* We use the *uniform mutation*, which either changes the test case structures (adding, deleting, or replacing API requests) or the input data. Test structure and test data mutation are equally probable, i.e, each has 50% probability of being applied. The mutation probability for each statement/data gene is equal to $1/n$, where n is the number of statements in the test case.
- *Recombination Operator:* We use the single-point crossover with a crossover probability of 0.75.

- *Selection:* We use the tournament selection with the default tournament size of 10.

4) LT-MOSA *parameters:* For LT-MOSA, we used the same parameters as for the MOSA algorithm except for the mutation operator, for which we use the mutation described in Section III-C. Additionally, we use the following parameter values for the *linkage learning* model:

- *Frequency:* We use a frequency of 10 generations for generating a new Linkage-Tree model. From a preliminary experiment that we have performed, this provides a balance between having too much overhead (< 10) and having an outdated model (> 10).
- *Recombination Operator:* We use the linkage-based recombination with a probability of 0.75.

F. Real-fault Detection

To find out the number of unique faults that the search algorithms can detect, EvoMaster checks the returned status codes from the HTTP requests for `5xx` server errors, as an indicator for a fault. Since web applications handle many different clients, when an error occurs it is not desirable for the application to crash or exit as this would also impact the other clients. Thus, web applications return a status code in the `5xx` range, indicating an error has occurred on the server's side. EvoMaster keeps track of the last executed statement in the SUT (excluding third-party libraries) when a `5xx` status code is returned, to distinguish between different errors that happen on the same endpoint.

G. Experimental Protocol

For each web application, all three search algorithms (MOSA, MIO, LT-MOSA) are separately executed, and the resulting number of test targets that are covered is recorded.

Since all three search algorithm used in the study are randomized, we can expect a fair amount of variation in the results. To mitigate this, we repeated every experiment 20 times, with a different random seed, and computed the average (median) results. In total, we performed 420 executions, three search algorithms for seven web applications with 20 repetitions each. With each execution taking 30 minutes, the total execution time is 8.75 days of consecutive running time.

To determine if the results (*i.e.,* code coverage and fault detection capability) of the three different algorithms are statistically significant, we use the unpaired Wilcoxon rank-sum test [48] with a threshold of 0.05. This is a non-parametric statistical test that determines if two data distributions are significantly different. Since we have three different data distributions, one for each search algorithm, we perform the Wilcoxon test pairwise between each configuration pair: (i) LT-MOSA and MOSA; (ii) LT-MOSA and MIO. We combine this with the Vargha-Delaney statistic [49] to measure the effect size of the result, which determines how large the difference between the two configuration pairs is.

To determine how the two configuration pairs compare in terms of efficiency, we analyze the code coverage at different points in time. While the effectiveness measures the code coverage only at the end of the allocated time, we also want to analyze how algorithms perform during the search. One way to quantify the efficiency of an algorithm is by plotting the number of test targets at predefined intervals during the search process. This is called a convergence graph. We collected the number of targets that have been covered for every generation of each independent run. To express the efficiency of the experimented algorithms using a single scalar value, we computed the overall convergence rate as the Area Under the Curve (AUC) delimited by the convergence graph. This metric is normalized by dividing the AUC in each run by the maximum possible AUC per application[3].

TABLE II: Median number of covered test targets.

Application	MIO		MOSA		LT-MOSA	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	1173.00	12.75	1177.00	132.00	1215.50	161.75
Features-Service	488.00	72.25	455.50	33.25	478.00	5.00
NCS	622.50	1.25	623.00	4.00	622.00	3.25
OCVN	2421.50	374.75	2931.50	271.00	4031.50	338.75
ProxyPrint	1485.50	16.25	1501.00	78.25	1602.50	59.00
Scout-API	1727.50	54.75	1707.00	69.00	1826.50	33.25
SCS	853.00	5.50	852.00	8.00	853.00	3.00

V. Results

This section details the results of the empirical study with the aim of answering our research questions.

A. RQ1: Code Coverage

Table II reports the median and inter-quartile range (IQR) of the number of test targets covered by MIO, MOSA, and LT-MOSA for each of the seven applications.

From Table II, we observe that LT-MOSA achieved the highest median value (avg. +334.75 targets) for four out of the seven applications, and MOSA and MIO both achieved the highest median value (+10.00 and +0.5 targets, respectively) for 1 out of the 7 applications. The largest increase in code coverage is observable for *OCVN*, for which LT-MOSA covered +1100.00 more targets. For *SCS*, both LT-MOSA and MIO covered the same number of targets (853.00). For both artificial applications, namely *NCS* and *SCS*, the difference between the search algorithms is minimal (≤ 1).

In terms of variability (IQR), there is no clear trend with regard to the applications under test and/or the search approaches. For example, in some cases, the winning configuration (LT-MOSA on *CatWatch*) has the highest IQR with a significant margin (161.75 vs. 132.00 or 12.75). On *Scout-API*, LT-MOSA yields the lowest IQR by a significant margin (33.25 vs. 8.00 or 5.50). Within and across each search algorithm, the IQR varies.

Table III reports the statistical significance (p-value), calculated by the Wilcoxon test, of the difference between the number of targets covered by LT-MOSA and the two baselines, MIO and MOSA. It also reports the magnitude of the differences according to the Vargha-Delaney $\hat{A}_{12}$ statistic.

[3]Which corresponds to the area of the box with a height of the maximum code coverage and a width equal to the search budget.

TABLE III: Statistical results (p-value and $\hat{A}_{12}$) for the covered test targets (*RQ1*). Significant p-values (*i.e.,* p-value < 0.05) are marked gray.

Application	LT-MOSA vs MIO		LT-MOSA vs MOSA	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
CatWatch	<0.01	0.87 (Large)	0.04	0.66 (Small)
Features-Service	0.34	0.54	<0.01	0.83 (Large)
NCS	0.86	0.49	0.90	0.38 (Small)
OCVN	<0.01	1.00 (Large)	<0.01	1.00 (Large)
ProxyPrint	<0.01	1.00 (Large)	<0.01	0.86 (Large)
Scout-API	<0.01	0.96 (Large)	<0.01	0.96 (Large)
SCS	0.86	0.40 (Small)	0.50	0.50

TABLE IV: Median number of detected real-faults.

Application	MIO		MOSA		LT-MOSA	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	13.00	0.25	12.00	2.00	13.50	2.25
Features-Service	17.00	0.00	17.00	0.00	18.00	0.50
NCS	0	0.00	0	0.00	0	0.00
OCVN	34.00	5.25	37.50	5.25	48.00	3.50
ProxyPrint	32.50	1.00	33.00	1.00	34.00	0.25
Scout-API	54.50	3.75	60.00	1.50	64.00	3.00
SCS	0	0.00	0	0.00	0	0.00

From Table III, we can observe that for the non-artificial web applications, LT-MOSA achieves a significantly higher code coverage than MIO in four out of five applications with a *large* effect size ($\hat{A}_{12}$ statistics). LT-MOSA significantly outperforms MOSA in all five applications. The effect size is *large* in four applications and *small* for *CatWatch*. For the two artificial applications, *NCS* and *SCS*, there is no statistical difference between the results of LT-MOSA and the two baselines (MIO and MOSA). This confirms our preliminary results reported in Table II. Moreover, the difference between LT-MOSA and MIO is not significant for *Features-Service*. Finally, in none of the applications in our benchmark, neither of the baselines achieved a significantly larger coverage than LT-MOSA.

In summary, LT-MOSA achieves significantly higher (most of the cases) or equal code coverage when applied to REST APIs as compared to both MIO and MOSA.

B. RQ2: Fault Detection Capability

Table IV reports the median number of real-faults (and the corresponding IQR) detected by MIO, MOSA, and LT-MOSA for each of the seven applications.

We observe that for both the artificial applications, *NCS* and *SCS*, the number of faults that have been detected by any search algorithm is zero. This is because these artificial applications are not designed to fail softly by returning `5xx` faults. For the open-source applications, LT-MOSA detects the largest number of faults (avg. +3.40 faults) in all five cases. The largest increase in fault-detection rate is observable for the *OCVN* application, with +10.5 more faults detected by LT-MOSA than the baselines. It is noteworthy that the largest difference between LT-MOSA and the baselines is on the *OCVN* application, which is the application with by far the most classes (*i.e.,* 548) and endpoints (*i.e.,* 258) in our benchmark. This could be explained by the fact that LT-MOSA also achieved a much higher code coverage for this application. However, the difference in detected faults for *OCVN* is larger than for the other applications in the benchmark, which could indicate that LT-MOSA is especially effective for testing large REST APIs. The faults detected by LT-MOSA are a superset of the faults detected by MIO and MOSA. These newly discovered faults originate from the additional coverage that LT-MOSA achieves.

TABLE V: Statistical results (p-value and $\hat{A}_{12}$) for the detected real-faults (*RQ2*). Significant p-values (*i.e.,* p-value < 0.05) are marked gray.

Application	LT-MOSA vs MIO		LT-MOSA vs MOSA	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
CatWatch	<0.01	0.7 (Large)	<0.01	0.84 (Large)
Features-Service	<0.01	0.92 (Large)	<0.01	0.98 (Large)
NCS	-	-	-	-
OCVN	<0.01	0.99 (Large)	<0.01	0.92 (Large)
ProxyPrint	<0.01	0.89 (Large)	0.03	0.66 (Small)
Scout-API	<0.01	1.00 (Large)	<0.01	0.91 (Large)
SCS	-	-	-	-

TABLE VI: Median normalized AUC for the number of covered test targets. The highest values are marked in gray.

Application	MIO	MOSA	LT-MOSA
CatWatch	0.77	0.78	0.78
Features-Service	0.78	0.75	0.82
NCS	0.99	0.99	0.99
OCVN	0.50	0.50	0.66
ProxyPrint	0.87	0.82	0.88
Scout-API	0.84	0.81	0.86
SCS	0.95	0.96	0.96

Table V reports the results of the statistical test, namely the Wilcoxon test, applied to the number of faults detected by LT-MOSA and the two baselines, MIO and MOSA. It also reports the magnitude of the differences (if any) obtained with the Vargha-Delaney $\hat{A}_{12}$ statistic. Significant p-values (*i.e.,* p-value < 0.05) are highlighted with gray color. From Table V, we can observe that LT-MOSA detects a significantly higher number of faults than MIO and MOSA in all non-artificial applications. The effect size ($\hat{A}_{12}$) is *large* in all comparisons, except for *ProxyPrint*, where the effect size is *small* when comparing LT-MOSA and MOSA. Since none of the algorithms detected any faults in the artificial applications, Table V does not report any p-value or $\hat{A}_{12}$ statistics for these applications.

In summary, we can conclude that LT-MOSA detects more faults than the state-of-the-art approaches, namely MIO and MOSA, for all applications in our benchmark.

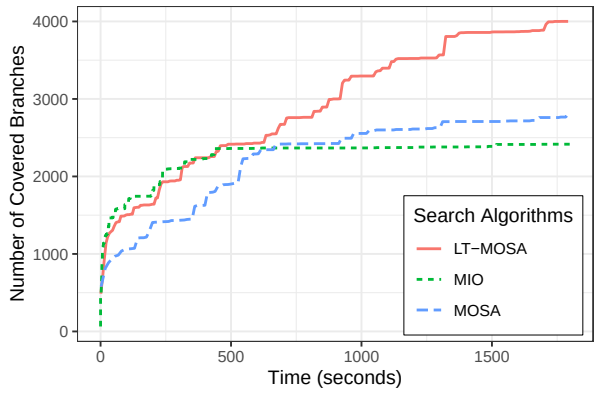

Fig. 2: Average number of targets covered by our approach (LT-MOSA) and the baselines (MOSA, MIO) for *OCVN*.

C. RQ3: Code Coverage over Time

Table VI reports the median Area Under the Curve (AUC) related to the number of targets covered over time by MIO, MOSA, and LT-MOSA for each of the seven applications. The AUC indicates how efficient the search algorithms are at reaching a certain code coverage. For more information on how the AUC is calculated and normalized see Section IV-G. Table VI highlights the search algorithm (in gray color) that achieved the highest AUC value.

We observe that for the open-source applications, LT-MOSA has the highest AUC (avg. +0.06) in four out of five applications, with the largest difference (+0.16) in the *OCVN* application. For *CatWatch*, both MOSA and LT-MOSA have the same AUC (*i.e.,* 0.78). From Tables II and III however, we can see that LT-MOSA covers significantly more targets (+38.5) after 30 minutes of search budget. This means that MOSA reaches a higher coverage in the beginning but loses to LT-MOSA over time.

Fig. 2 shows the (median) number of targets covered over time by the different search algorithms for *OCVN*, which is the largest application in our benchmark. In the beginning of the experiment (0-500 seconds), MIO and LT-MOSA perform roughly equal. After the first 500 seconds, LT-MOSA outperforms MIO. This results in a much larger AUC value (+ 0.16) for LT-MOSA compared to MIO as indicated in Table VI. We conclude that LT-MOSA significantly outperforms both MOSA and MIO in term of effectiveness and efficiency on this application. To reaffirm this, we can observe that in Fig. 2, MIO never reaches 2700 covered targets, MOSA takes 1311 seconds to reach that many targets, and LT-MOSA performs this in just 713 seconds, almost half the time of MOSA.

For the two artificial applications, *NCS* and *SCS*, the difference in AUC between the three search algorithms is very minimal ($\leq$ 0.01). From Table II, we can also see that LT-MOSA covers one target more than MOSA on *SCS* and one target less on *NCS*. However, they both yield the same AUC, *i.e.,* 0.96 (*SCS*) and 0.99 (*NCS*). These results are in line with the results from *RQ1*.

> In summary, we can conclude that LT-MOSA achieves higher AUC values than the baselines, i.e., it covers more targets and in less time.

VI. Threats to Validity

This section discusses the potential threats to the validity of the study performed in this paper.

Threats to *construct validity*. We rely on well-established metrics in software testing to compare the different test case generation approaches, namely code coverage, fault detection capability, and running time. As a stopping condition for the search, we measured the search budget in terms of running time (*i.e.,* 30 minutes) rather than considering the number of executed tests, or HTTP requests. Given that the different algorithms in the comparison use different genetic operators, with different overhead, execution time provides a fairer measure of time allocation.

Threats to *external validity*. An important threat regards the number of web services in our benchmark. We selected seven web/enterprise applications from the EMB benchmark. The benchmark has been widely used in the related literature on testing for REST APIs. The applications are diverse in terms of size, application domain, and purpose. Further experiments on a larger set of web/enterprise applications would increase the confidence in the generalizability of our study. A larger empirical evaluation is part of our future agenda.

Threats to *conclusion validity* are related to the randomized nature of EAs. To minimize this risk, we have performed each experiment 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [50] and analyzed the possible impact of different random seeds on our results. We used the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

VII. Conclusions and Future Work

In this paper, we have used agglomerative hierarchical clustering to learn a *linkage tree model* that captures promising patterns of HTTP requests in automatically generated system-level test cases. We proposed a novel algorithm, called LT-MOSA, that extends state-of-the-art approaches by tailoring and incorporating *linkage learning* within its genetic operators. Linkage learning helps to preserve and replicate patterns of API requests that depend on each other.

We implemented LT-MOSA, in EvoMaster and evaluated it on seven web applications from the EMB benchmark. Our results show that LT-MOSA significantly improves code coverage and can detect more faults than two state-of-the-art approaches in REST API testing, namely MIO [13] and MOSA [24]. This suggests that using unsupervised machine learning (and agglomerative hierarchical clustering in our case) is a very promising research direction.

Based on our promising results, there are multiple potential directions for future works. In this paper, we used the UPGMA algorithm for hierarchical clustering. Therefore, we intend to investigate more learning algorithms within the hierarchical clustering category. We also plan to investigate other categories of machine learning methods alternative to hierarchical clustering, such as Bayesian Network [29]. Finally, LT-MOSA uses a fixed parameter K for the linkage learning frequency. We plan to investigate alternative, more adaptive mechanisms to decide whether the linkage tree model needs to be retrained or not. Finally, we intend to implement and apply linkage learning to unit-test case generation as well.

REFERENCES

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to soap, wsdl, and uddi," *IEEE Internet computing*, vol. 6, no. 2, pp. 86–93, 2002.

[2] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine 2000, vol. 7.

[3] V. Lenarduzzi and A. Panichella, "Serverless testing: Tool vendors' and experts' points of view," *IEEE Software*, vol. 38, no. 1, pp. 54–60, 2020.

[4] V. Lenarduzzi, J. Daly, A. Martini, S. Panichella, and D. A. Tamburri, "Toward a technical debt conceptualization for serverless computing," *IEEE Software*, vol. 38, no. 1, pp. 40–47, 2020.

[5] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[6] A. Panichella and U. R. Molina, "Java unit testing tool competition-fifth round," in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2017, pp. 32–38.

[7] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, no. 3, pp. 611–639, 2015.

[8] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[9] A. Arcuri and J. P. Galeotti, "Handling sql databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.

[10] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[11] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[12] M. Zhang, B. Marculescu, and A. Arcuri, "Resource-based test case generation for restful web services," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1426–1434.

[13] A. Arcuri, "Test suite generation with the many independent objective (mio) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.

[14] R. A. Watson and T. Jansen, "A building-block royal road where crossover is provably essential," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1452–1459.

[15] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Replication package of "Improving Test Case Generation for REST APIs Through Hierarchical Clustering"," 2021.

[16] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[17] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[18] M. Soltani, A. Panichella, and A. Van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1294–1317, 2018.

[19] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from ocl constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[20] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 3–45.

[21] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[22] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[23] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 142–152.

[24] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proceedings of the International Conference on Software Testing, Verification and Validation, (ICST'15)*, Graz, Austria, 2015, pp. 1–10.

[25] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[26] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb 2018.

[27] R. A. Watson, G. S. Hornby, and J. B. Pollack, "Modeling building-block interdependency," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1998, pp. 97–106.

[28] M. Pelikan and D. E. Goldberg, "Escaping hierarchical traps with competent genetic algorithms," in *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, 2001, pp. 511–518.

[29] M. Pelikan, D. E. Goldberg, E. Cantú-Paz *et al.*, "Boa: The bayesian optimization algorithm," in *Proceedings of the genetic and evolutionary computation conference GECCO-99*, vol. 1. Citeseer, 1999, pp. 525–532.

[30] T.-L. Yu, D. E. Goldberg, K. Sastry, C. F. Lima, and M. Pelikan, "Dependency structure matrix, genetic algorithms, and effective recombination," *Evolutionary computation*, vol. 17, no. 4, pp. 595–626, 2009.

[31] M. W. Przewozniczek and M. M. Komarnicki, "Empirical linkage learning," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 6, pp. 1097–1111, 2020.

[32] D. Thierens, "The linkage tree genetic algorithm," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2010, pp. 264–273.

[33] D. Thierens and P. A. Bosman, "Optimal mixing evolutionary algorithms," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 2011, pp. 617–624.

[34] A. Bouter, T. Alderliesten, C. Witteveen, and P. A. Bosman, "Exploiting linkage information in real-valued optimization with the real-valued gene-pool optimal mixing evolutionary algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 705–712.

[35] P. A. Bosman, N. H. Luong, and D. Thierens, "Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 637–644.

[36] M. Olsthoorn and A. Panichella, "Multi-objective test case selection through linkage learning-based crossover," in *International Symposium on Search Based Software Engineering*. Springer, 2021.

[37] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018.

[38] F. U. Haq, D. Shin, L. C. Briand, T. Stifter, and J. Wang, "Automatic test suite generation for key-points detection dnns using many-objective search," *arXiv preprint arXiv:2012.06511*, 2020.

[39] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[40] R. Sokal and C. D. Michener, "A statistical method for evaluating systematic relationships," *University of Kansas science bulletin*, vol. 38, pp. 1409–1438, 1958.
[41] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.
[42] S. Sivanandam and S. Deepa, "Genetic algorithms," in *Introduction to genetic algorithms*. Springer, 2008, pp. 15–37.
[43] J. D. Schaffer, R. Caruana, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proceedings of the 3rd international conference on genetic algorithms*, 1989, pp. 51–60.
[44] J. E. Smith and T. C. Fogarty, "Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1996, pp. 441–450.
[45] A. Arcuri and J. P. Galeotti, "Testability transformations for existing apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 153–163.
[46] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.
[47] A. Arcuri, "Many independent objective (mio) algorithm for test suite generation," in *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'17)*. Paderborn, Germany: Springer International Publishing, 2017, pp. 3–17.
[48] W. J. Conover, *Practical nonparametric statistics*. John Wiley & Sons, 1998, vol. 350.
[49] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[50] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.