Improving Test Case Generation for RESTful APIs through Seeded Sampling

Chiel de Vries , **Mitchell Olsthoorn** , **Annibale Panichella**
Delft University of Technology

Abstract

To validate the quality of software, test cases are used. These test cases are often manually-written, which is labor-intensive. To avoid this problem, automated software testing was invented. Search-based software testing is a useful tool for developers to automatically generate test cases. However, improvements are still needed to create test cases that compete with manually-written ones.

EvoMaster is a tool that generates system-level test cases for RESTful APIs using the MIO algorithm. An important aspect of this algorithm is sampling new test cases. Currently, EvoMaster employs random and smart sampling to achieve this goal. This paper aims to improve the coverage of the generated tests by expanding the sampling methods with seeded sampling. This method consists of extracting sequences of HTTP requests from manually-written tests and using these to sample new test cases.

Seeded sampling is evaluated on two RESTful APIs with 7 different parameter sets. We show that the addition of seeded sampling can improve the coverage achieved by EvoMaster compared to the current combination of sampling techniques. Nonetheless, this paper has some limitations. It only takes two RESTful APIs into account and has a small amount of benchmark runs to back its findings.

Keywords

Search-based Software Testing, Seeded Sampling, Test Case Generation, RESTful APIs

1 Introduction

Software developers generally see testing as labor-intensive, tedious work. It is, however, of great importance that it is done since it measures and ensures software quality [8]. To support developers and reduce testing time and effort, there exist techniques to automate this. Search-based software testing (SBST) is such a technique. It uses search algorithms to automatically generate test suites. EvoMaster [1] is an SBST tool that generates test suites for RESTful APIs. RESTful APIs are services that provide interoperability between computer systems online. They use the Hypertext Transfer Protocol (HTTP) for communication, enabling a requesting system to access or manipulate resources. Testing a RESTful API is challenging since their inputs and outputs are HTTP requests and responses. EvoMaster automates this process by trying to improve code coverage and fault detection by using the Many Independent Object (MIO) algorithm [2] as the core search algorithm. Arcuri [1] has shown that this method can successfully generate system-level test cases for RESTful APIs. However, there are cases where it cannot generate a higher coverage than manual testing [1].

In SBST, test cases are evolved to achieve certain objectives, for instance, high coverage. Before these test cases can be evolved however, they must be initialized (sampling). Typically this is done randomly (random sampling), but EvoMaster has an option to do it differently: smart sampling [1]. This method ensures that HTTP requests are called in a meaningful order, rather than a completely random one. This paper focusses on improving the current sampling method by adding a new option: Seeded sampling.

Seeding loosely refers to any technique that exploits previous, related knowledge to help solve a search problem [13]. EvoMaster uses a form of seeding by using the schema of the system under test (SUT) to create meaningful HTTP requests [1]. However, EvoMaster does not use existing test cases as seeds. Previous studies [4, 13] describe using existing, manually-written tests to improve the sampling of new test cases. While these studies showed the effectiveness of sampling from existing tests, they focused on unit testing. Therefore, it is still unclear to what extend seeding with manually-written tests can improve the performance of system-level test generation tools like EvoMaster.

In this paper, we design a system that uses manually-written test cases to improve test case sampling for EvoMaster. The following questions are taken into account when designing this new system:

RQ 1: What model can be used to extract useful information from manually-written test cases effectively?

RQ 2: How can the extracted model be used to generate new test cases?

RQ 3: To what extent can the extracted model improve coverage compared to the current combination of sampling

techniques used by EvoMaster?

The first and second questions are about designing a model that uses manually-written tests to seed the generation of new test cases. The third question strives to verify whether the new sampling technique improves upon the existing one implemented in EvoMaster.

More background information concerning RESTful APIs and EvoMaster are discussed in section 2. Section 3.1 contains a description of the problem. Section 3 covers the design of the new seeded sampling technique. In section 4 we describe our experimental setup, the gathered results, and the threats to validity. Section 5 contains the ethical aspects and reproducibility of our methods. The conclusion can be found in section 6.

2 Background

This section will cover relevant background information and other related work. First, basic knowledge of RESTful APIs will be discussed. Then EvoMaster and its current sampling methods will be introduced. Lastly, seeding methods using existing test cases will get explained.

2.1 RESTful APIs

RESTful APIs are a common sight in enterprise applications [1], especially when using a microservice architecture. A microservice architecture is a type of architecture where a large and complex application is split into individual components which are easier to develop and maintain [9]. Amongst big companies like Netflix, Airbnb, Amazon, and eBay it is common practice to use microservices [12].

REST stands for Representational State Transfer and was coined in 2000 by Roy Fielding [5]. It is an architectural style for distributed hypermedia systems that adheres to a set of architectural constraints. These constraints emphasize scalability, generality of interfaces, and independent deployment of components [5]. These characteristics make RESTful APIs ideal for use in systems with a microservice architecture.

REST often uses HTTP, the main protocol of communication on the internet [1]. An HTTP client sends a request to a server which will then send a response back to the client. An HTTP request consists of four main components:

Operation: The operation to perform, such as posting or deleting a resource.

Path: The location of the resource on which the operation should be performed.

Headers: Metadata about the request, such as the format in which the response is expected.

Body: The payload of the message.

This paper will mostly focus on the operation and path of a request since the headers and body are often application-specific and therefore not necessary to use in this explanation. Hence, the examples of HTTP requests in this paper will omit those components and only show the operation and path.

To get familiar with HTTP requests a couple of examples have been laid out here. Suppose we have a RESTful API of a supermarket.

```
GET     /products
```

The first element (GET) is the operation of the request and the second element(/products) is the path. This request tells the server that the client wants to receive the resource located at /products. This would be a list of the products available in the store. If the client wants to receive a specific product the request would look like this:

```
GET     /products/milk
```

This request would result in the client receiving the available information on milk. If a client is only interested in the price of milk, he could send the following request:

```
GET     /products/milk/price
```

Now suppose that the store wants to add a new product. That should be done using the POST operation.

```
POST    /products/yoghurt
```

The body of the request should contain the resource holding the information on yoghurt. This resource will be returned to any client that tries to use the GET operation on the same path.

There is much more to HTTP than these brief examples, but they should suffice to form a basic understanding of the HTTP protocol. A good summary of HTTP can be found in section 2 of "RESTful API Automated Test Case Generation with EvoMaster" [1].

A client does not know what the internal structure of a RESTful API is. How does a client know it should request

```
GET     /products/milk/price
```

instead of

```
GET     /products/milk/cost
```

when it wants to know how expensive milk is? This, among other things, can be found in the schema of the API. The schema contains a list of endpoints. An endpoint is an pair consisting of an operation and a path. The list of endpoints for the example API would contain the following three endpoints:

```
GET     /products/{productName}
POST    /products/{productName}
GET     /products/{productName}/price
```

Note the path elements in between the curly brackets . These are called path parameters, which represent variables in the path. This means that both

```
GET     /products/milk
```

and

```
GET     /products/yoghurt
```

match the same endpoint, but with different parameters. The response sent by the server will differ based on the given parameter. The schema also shows that

```
GET     /products/milk/cost
```

is not a valid request, since it does not match any of the listed endpoints.

2.2 EvoMaster and MIO

Testing RESTful APIs is challenging, time-consuming work. EvoMaster is a tool that tries to ease this process. It automatically generates system-level test cases for these services. Various studies discuss black-box testing of RESTful APIs. Black-box testing means that the tool creating the tests has no knowledge of the internals of the system under test (SUT) [10]. EvoMaster sets itself apart by focussing primarily on white-box testing, which means that the internals of the SUT are known [10]. The reason for choosing white-box testing over black-box testing is that white-box testing has potentially better results in terms of coverage [7], because of the extra knowledge it has access to. This, in turn, means that EvoMaster is meant for developers or operators since access to the source code of the application is necessary.

EvoMaster makes use of an evolutionary algorithm called MIO as its main search algorithm [1]. Other algorithms such as Whole Test Suite (WTS) [6] and Many-Objective Sorting Algorithm (MOSA) [11] are implemented as well, but will not be used for this work. MIO is a multi objective search algorithm. This means that it aims to achieve multiple objectives at the same time, e.g. different branches to cover. To do this, MIO maintains an archive of tests. In this archive it keeps a population of tests of size n for each objective. Thus, given m objectives, the archive can contain up to $m * n$ test cases.

When starting the search, MIO will sample a test case. From then on, it will iterative choose to either sample a new test case (with probabiity P_{sample}, or pick an existing test from the archive and mutate it (with probability $1 - P_{sample}$). Each sampled or mutated test is then evaluated with a fitness function. The fitness reflects how good the test is in achieving an objective. A test may be saved to 0 or more of the m populations in the archive if its fitness for that target is higher than some other test in that population [2]. Thus, after running the algorithm for a certain time t, an archive filled with tests that cover as many targets as possible remains.

This is a simplified description of the MIO algorithm developed by Andrea Arcuri [2]. MIO has more interesting quirks, like controlling the balance between exploration and exploitation of the search space, but that is outside of the scope of this paper. This paper focusses on the way tests are initially sampled in MIO.

2.3 Sampling in EvoMaster

Sampling new test cases happens in two distinct ways in EvoMaster [1]. These are random sampling and smart sampling. Both techniques have their advantages and disadvantages, which is why they are both used. When a new test case needs to be generated, the system chooses either random sampling (with probability P_{random} or smart sampling (with probability $1 - P_{random}$.

Random Sampling

Random sampling works in a straight forward manner. EvoMaster does not just generate a random stream of bits to send to the SUT, since the chance that that would result in a meaningful HTTP request is extremely small. Instead it picks a random endpoint based on information available in the schema of the SUT. This ensures that the request is, at the very least, something the application can handle. Random sampling is quick and easy, but there is a problem: testing some endpoints requires setting the state of the application beforehand. Consider the following example:

```
GET     /products/milk
```

This endpoint needs a product with called milk to exist, otherwise the response will be a 404 "not found", A randomly generated test is unlikely to generate the correct sequence of HTTP requests to create a matching resource. This is where smart sampling comes in.

Smart Sampling

Smart sampling uses predefined test templates when certain types of requests are encountered. It starts similar to random sampling by picking a random endpoint of the API. Consider the same example as before:

```
GET     /products/milk
```

Then, EvoMaster checks whether that endpoint needs a pre-existing resource associated with it. It does this by checking the operation (GET in this case), and checking whether that operation needs a pre-existing resource. The operations that need such a resource are GET, PUT, PATCH and DELETE. In this example a product called milk is necessary. It will then add a request to the test which creates this resource. The sequence of requests will then be as follows:

```
POST    /products/milk
GET     /products/milk
```

The resulting sequence will depend on the endpoints available in the SUT's schema. This technique cannot be used if the SUT has no endpoints with a POST operation.

This is a basic example showing the main idea behind smart sampling. However, it does more than defining templates for each HTTP operation such that it becomes meaningful. When the API has elements such as collections, smart sampling ensures that the collection is created and its ID remembered before an item is created for that collection. Furthermore, during the search, tests are mutated. This could potentially break the carefully crafted properties of a smart sampled test. Therefore, EvoMaster blocks some mutations that can break these properties. For instance, mutations that change the structure of the test case (i.e. the order of HTTP requests). A more in depth explanation can be found in section 5 of "RESTful API Automated Test Case Generation with EvoMaster" [1].

2.4 Seeding with existing test cases

Seeding refers to the use of information about the SUT during the search process [4]. This paper will use information in the form of existing, manually-written test cases to seed with. Previous studies have used this form of seeding [4, 13]. Two methods have demonstrated good results for unit testing and will be discussed here: cloning and carving.

Cloning

Suppose there exists a set of parsed test cases T. Cloning refers to the reuse of one of those test cases to sample a new one. Generally, for SBST tools, this means that given a probability P_{clone}, a test is not sampled randomly but copied from T. This in itself does not improve anything since it produces an exact copy of an already existing test. Therefore, a random amount of mutations between 0 and N are applied to the cloned test. This promotes diversity in the test suite, which could expand the reachable search space.

Carving

Suppose there exists a set of parsed tests T. Carving refers to the reuse of initialized objects that can be found in existing tests. Some objects in a SUT require a specific sequence of method calls to reach the state that should be tested. Randomly sampled tests are unlikely to generate this sequence of method calls. Therefore, it can be useful to reuse these objects for other tests. A carved object is then defined as the initialization of the object together with this sequence of specific method calls on it. Carved objects can be used for random sampling, by randomly picking it instead of some other method call, or as a mutation.

3 Seeded sampling

This section states the problem and describes the method used for solving this problem. Then the three main elements of the seeded sampling technique are covered. These elements are parsing, cloning and carving.

3.1 Problem description

The goal of this paper is to further improve sampling by using existing, manually-written test data of a SUT. The idea behind this approach is that a developer knows the specifics of what needs to be tested. Therefore, harvesting this knowledge and reusing it can potentially improve the coverage performance of EvoMaster. To do this the following two research questions are answered:

RQ 1: What model can be used to extract useful information from manually-written test cases effectively?

RQ 2: How can the extracted model be used to generate new test cases?

These questions are answered by applying the cloning and carving techniques discussed in section 2.4 to test case generation for RESTful APIs.

3.2 General approach

The main idea of this paper is to use manually-written test cases to improve the achieved coverage of EvoMaster. The first thing that is required for achieving this goal is finding a way to interpret these test.

The test cases are parsed to the internal representation used in EvoMaster and put into an archive. Each test in the archive is then further analysed to retreive any Resource Generating Sequences (RGS). An RGS is a sequence of POST and/or PUT requests. These can set the state of a RESTful API and are therefore useful parts of the test case. This process reflects carving, which is mentioned in section 2.4. Carved RGSs are also put into an archive.

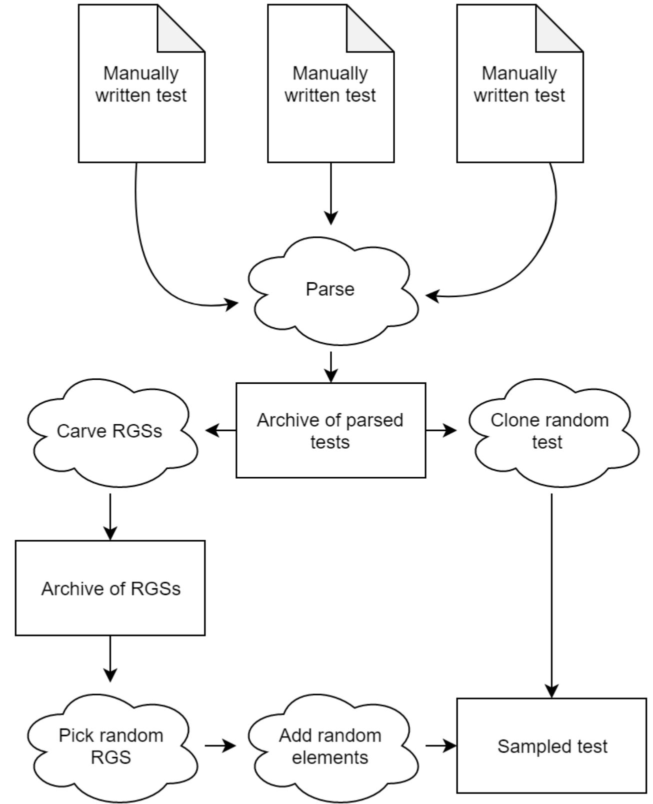


Figure 1: A diagram representing the general approach to seeded sampling. It shows the main flow from the manually-written test cases to the two archives from which new test cases can be sampled.

The two archives are used when performing seeded sampling. Each time the system randomly picks whether to copy a test from the test archive, or to take an RGS from the RGS archive. When an RGS is picked some random elements are added to it. The former reflects cloning, which also mentioned in section 2.4.

Seeded sampling is not meant to replace either random or smart sampling. The idea is to make them work alongside each other. That way the search space is widened which can lead to greater coverage of the generated test suites.

The final control flow is visualized in figure 1. In natural language the sampler looks as follows:

- The system randomly picks either random, smart or seeded sampling given probabilities P_{random}, P_{smart} and P_{seeded}, where $P_{random} + P_{smart} + P_{seeded} = 1$ and $P_{random}, P_{smart}, P_{seeded} \in [0, 1]$.
- When random or smart sampling are chosen they are executed as described by Arcuri [1].
- When seeded sampling is chosen, then the system randomly chooses to either clone a test or carve use a carved RGS given probabilities P_{clone} and P_{carve}, where $P_{clone} + P_{carve} = 1$ and $P_{clone}, P_{carve} \in [0, 1]$.

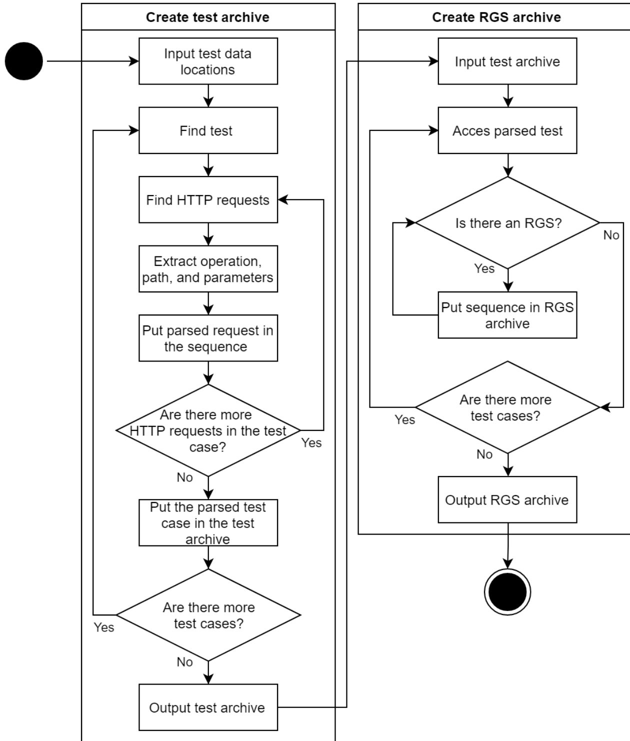


Figure 2: An activity diagram for the parser. The two parts of the parser are shown as well as the control flow of the system.

- Then cloning or carving is performed as described in sections 3.4 and 3.5 respectively.

3.3 Parsing

Parsing test cases might be the most important part of seeded sampling. Manually-written test cases for RESTful APIs come in many different forms since each developer has his or her own style for writing test cases. Therefore, the parser made for this paper is not designed to handle every single way one can write tests cases, but it should be sufficient to serve as a proof of concept.

The parser, visualized in figure 2, takes as input the paths of the test suites and it outputs the two archives: the test archive and the RGS archive. Creating these archives is done in two steps. First, the test archive is generated by extracting the sequences of HTTP requests from each test case. Only this sequence is necessary information, since EvoMaster can generate the assertions on each response by itself. This sequence is then put into the test archive. When all tests have been archived, the system loops over each test in the archive to extract RGSs from each test case. Each RGS is then put into the RGS archive.

After parsing the test cases we are left with two archives:

The test archive: This contains the complete sequences of HTTP request of the manually-written test cases.

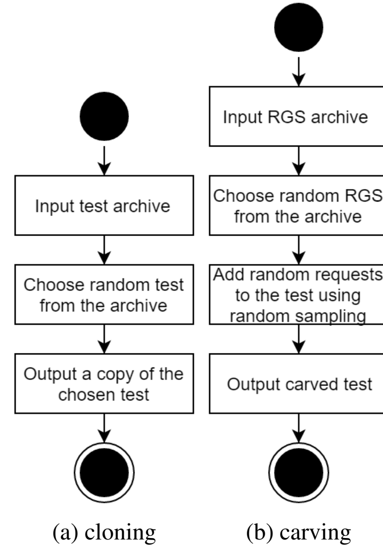


Figure 3: Activity diagrams for cloning and carving

The RGS archive: This contains sequences of POST and/or PUT requests found in the manually-written test cases.

3.4 Cloning

Cloning, together with carving, are part of the actual sampler of the system. It is a simple method, which can been seen in figure 3a. When a test case is sampled using seeded sampling, it randomly chooses (given a probability P_{clone}) to copy a random test case from the test archive. This test is then directly fed to the evolutionary algorithm that EvoMaster uses, wherein it will be evaluated and mutated. The cloned test case will not be mutated before being sampled, as is done in other literature [4, 13].

3.5 Carving

Carving is the more difficult of the two seeded sampling techniques used in this paper. Its workings are shown in figure 3b. The first part is fairly similar to cloning. When sampling, a random RGS is chosen from the RGS archive. To make this test case valuable some extra random request are added to the sequence which can act on the resources created by the RGS. Then the test is ready to be used in the evolutionary algorithm.

An RGS is similar to the initialisation of a particular object in unit testing. Just like the state of an object is set by a series of method calls, the state of a RESTful API is set by a series of POST and/or PUT requests. Therefore an RGS can be seen as an object initialisation, which is why they are carved from the existing test cases.

4 Empirical evaluation

This section describes the empirical evaluation method used to evaluate whether seeded sampling can improve coverage achieved by the generated test suite. First, the setup used for this evaluation is covered. Second, the results are presented with statistical analysis. Third, threats to the validity of the evaluation are discussed.

4.1 Experimental setup

The impact of seeded sampling will be evaluated by testing it on two RESTful APIs: features-service[1] and rest-news[2]. These APIs are chosen primarily because they have been used for benchmarks for earlier versions of EvoMaster. This means that the necessary drivers to run EvoMaster in white-box mode have already been written and are available in the EvoMaster Benchmark repository[3]. The second reason for using these two APIs is that they have test cases that are written using the REST-assured library[4]. This is the same library that EvoMaster used to generate its tests with, which makes parsing the tests easier. The parser developed for this paper is unable to parse all possible test cases. Therefore, some test cases had to be rewritten to be usable. All endpoints are of each API are considered in the evaluation.

The empirical evaluation is meant to answer **RQ 3:** To what extent can the extracted model improve coverage compared to the current combination of sampling techniques used by EvoMaster? This question is answered by running EvoMaster with a search budget of five minutes for ten times for different sets of parameters. Ten repetitions are chosen since that filters out the worst outliers, while keeping the overall computing time low. The parameters which are changed between runs are the probabilities that each form of sampling will occur: P_{random}, P_{smart}, P_{seeded}, P_{clone} and P_{carve}. Henceforth, these parameters will be referred to as a 5-tuple $(P_{random}, P_{smart}, P_{seeded}, P_{clone}, P_{carve})$. All other parameters are set to the defaults set by EvoMaster.

A baseline will be set using $(0.5, 0.5, 0.0, -, -)$. This is chosen as a baseline because Arcuri recommends using a middle value for P_{random} and P_{smart} [1]. The other parameter sets are determined by increasing the seeded sampling rate at the cost of the random and smart sampling rate with each consecutive run. This results in the following sets of parameters:

$\mathcal{P}_0 = (0.5, 0.5, 0.0, -, -)$

$\mathcal{P}_1 = (0.4, 0.4, 0.2, 0.5, 0.5)$

$\mathcal{P}_2 = (0.3, 0.3, 0.4, 0.5, 0.5)$

$\mathcal{P}_3 = (0.2, 0.2, 0.6, 0.5, 0.5)$

$\mathcal{P}_4 = (0.1, 0.1, 0.8, 0.5, 0.5)$

$\mathcal{P}_5 = (0.0, 0.0, 1.0, 0.5, 0.5)$

These parameter sets provide insight into the influence of seeded sampling as a whole, but it is also interesting to see the impacts of cloning and carving individually. Therefore, two more parameter sets are added:

$\mathcal{P}_6 = (0.4, 0.4, 0.2, 1.0, 0.0)$

$\mathcal{P}_7 = (0.4, 0.4, 0.2, 0.0, 1.0)$

In the first set, only cloning will be performed when the sampler chooses seeded sampling, and in the second only carving. These will be tested against a different baseline, namely $\mathcal{P}_1$. This shows whether cloning, carving or a mix of both performs best. $\mathcal{P}_1$ is chosen as a baseline because it is expected to perform the best of all the parameter sets.

Two measures are used to evaluate the performance of each parameter set: The amount of covered test targets and the percentage of bytecode line coverage. Test targets include bytecode statements, branches, call methods executions and HTTP return statuses [1]. Bytecode line coverage is the amount of lines of bytecode that is covered by the resulting test suite. Both these measures are included since only looking at coverage can be misleading. Therefore, the covered test targets are added to paint a more complete picture.

The results will be subjected to the Mann-Whitney-Wilcoxon U-test [3] to show whether they are significant. $\mathcal{P}_1 - \mathcal{P}_5$ will be compared to $\mathcal{P}_0$, and $\mathcal{P}_6$ and $\mathcal{P}_7$ will be compared to $\mathcal{P}_1$. This setup provides enough insight to validate or reject whether or not seeded sampling improves the coverage achieved by the generated test suite.

Table 1: Covered targets and coverage percentage results for $\mathcal{P}_0$ to $\mathcal{P}_5$ with corresponding p-values. The results for the news and the feature-service API are shown.

	covered targets	p-value	coverage	p-value
News				
$\mathcal{P}_0$	293.8	-	46.9%	-
$\mathcal{P}_1$	308.0	0.010	48.9%	0.038
$\mathcal{P}_2$	305.1	0.075	47.9%	0.336
$\mathcal{P}_3$	304.4	0.121	48.4%	0.196
$\mathcal{P}_4$	300.6	0.150	48.2%	0.218
$\mathcal{P}_5$	94.2	<0.001	16.8%	<0.001
Feature-service				
$\mathcal{P}_0$	399.4	-	35.8%	-
$\mathcal{P}_1$	404.9	0.003	36.8%	0.001
$\mathcal{P}_2$	400.3	0.003	36.3%	0.004
$\mathcal{P}_3$	403.6	0.006	36.7%	0.001
$\mathcal{P}_4$	408.0	0.343	36.9%	0.088
$\mathcal{P}_5$	86.0	<0.001	8.2%	<0.001

Covered targets and coverage are shown as the average over ten runs. Coverage is measured as bytecode line coverage.

4.2 Results

Table 1 shows the end results of running EvoMaster with parameter sets $\mathcal{P}_0$ to $\mathcal{P}_5$ on both SUTs. From this table it becomes evident that all parameter sets, except for $\mathcal{P}_5$, perform on average better on both SUTs. However, the average amount of covered increases by a no more than 15 and the coverage by a no more than 2 percent points. The significance of the results, especially for the news API, are debatable. For the news API, only $\mathcal{P}_1$ and $\mathcal{P}_5$ show significant results (p-value lower than 0.05). The feature-service API has no significant results for $\mathcal{P}_3$ (only on covered targets) and $\mathcal{P}_4$. $\mathcal{P}_5$ deviates the most from the baseline. This is likely because it only uses seeded sampling, and no other method. This results in very narrow exploration of the search space, since only the pre-existing tests can be used as a springboard for exploration.

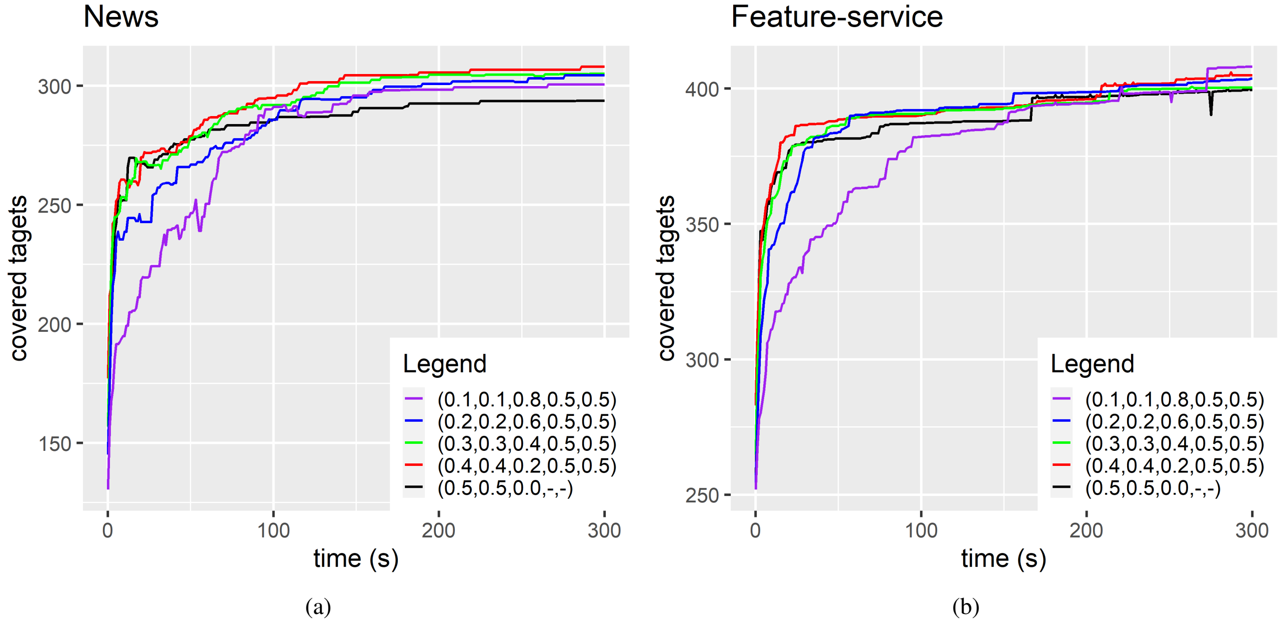
Figure 4 shows how many test targets are covered on average over time for different parameter sets. $\mathcal{P}_5$ has been omit-

[1] https://github.com/JavierMF/features-service

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

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

[4] http://rest-assured.io/



(a) (b)

Figure 4: Graphs showing the average amount of covered targets over time for $\mathcal{P}_0$ to $\mathcal{P}_4$. The parameter sets are written as a 5-tuple. (a) and (b) show the results for the news and the feature-service API respectively.

ted from the graphs because the amount of targets it covers are low compared to the other parameter sets. The graphs would become unreadable if $\mathcal{P}_5$ was included. These graph are shown because they give insight into the speed at which the test suites evolve. It is clear that even after just five minutes the search seem to converge. This reinforces the assumption that a runtime of five minutes is enough time to produce meaningful results. Furthermore, the graphs show that the parameter sets with a high value for P_{seeded}, namely $\mathcal{P}_3$ (blue) and $\mathcal{P}_4$ (purple), have a slower start compared to the other sets. The explored search space with seeded sampling is narrow, because the test are sampled from a limited source. Therefore, it can hinder the evolution speed of the test suite. $\mathcal{P}_0$, $\mathcal{P}_1$ and $\mathcal{P}_2$ have a similar evolution speed, but $\mathcal{P}_1$ and $\mathcal{P}_2$ end with more targets covered on average. Some dips or spikes in the amount of covered targets can be spotted in the graphs. This should not happen since the MIO algorithm never replaces tests in the archive with tests that achieve a lower coverage. These dips are artefacts created by missing values in the dataset used to create the graphs. EvoMaster pauses sometimes, not outputting any data for some time. These missing values are omitted when calculating the average of the runs. This can result in sudden dips or spikes in the graph.

Table 2 shows the end results of running EvoMaster with parameter sets $\mathcal{P}_1$, $\mathcal{P}_6$ and $\mathcal{P}_7$. $\mathcal{P}_1$ is added to make the comparison easier. From the high p-values it becomes clear that these parameter sets show no significant difference between each other. Therefore, nothing can be said about the influence of either cloning or carving on the outcome of seeded sampling. Although $\mathcal{P}_6$ shows higher amount of covered targets for the feature-service API, this is probably due to an anomaly in the data. A parameter set with a higher seeded sampling rate might have given more insight into the influences of cloning and carving on their own.

Table 2: Covered targets and coverage percentage results for $\mathcal{P}_1$, $\mathcal{P}_6$, and $\mathcal{P}_7$ with corresponding p-values. The results for the news and the feature-service API are shown.

	covered targets	p-value	coverage	p-value
News				
$\mathcal{P}_1$	308.0	-	48.9%	-
$\mathcal{P}_6$	308.2	0.405	48.7%	0.784
$\mathcal{P}_7$	303.0	0.307	48.4%	0.430
Feature-service				
$\mathcal{P}_1$	404.9	-	36.8%	-
$\mathcal{P}_6$	420.4	0.732	38.0%	0.816
$\mathcal{P}_7$	412.4	0.676	37.4%	0.625

Covered targets and coverage are shown as the average over ten runs. Coverage is measured as bytecode line coverage.

The covered targets over time graph in figure 5a shows that the different parameter sets evolve similarly, reinforcing the inconclusiveness of the experiment. In figure 5b the higher amount of covered targets for $\mathcal{P}_6$ (brown) can be seen clearly. Apart from that, the parameter sets seem to evolve at the same speed for the feature service API.

To summarize, **RQ 3** is answered. The seeded sampling technique can improve coverage compared to the current combination of sampling techniques used by EvoMaster. Low seeded sampling rates $(0.2 - 0.4)$ perform best on the evaluated SUTs, while higher seeded sampling rate can be detrimental to the performance. The individual influence of the to components of the model, cloning and carving are un-

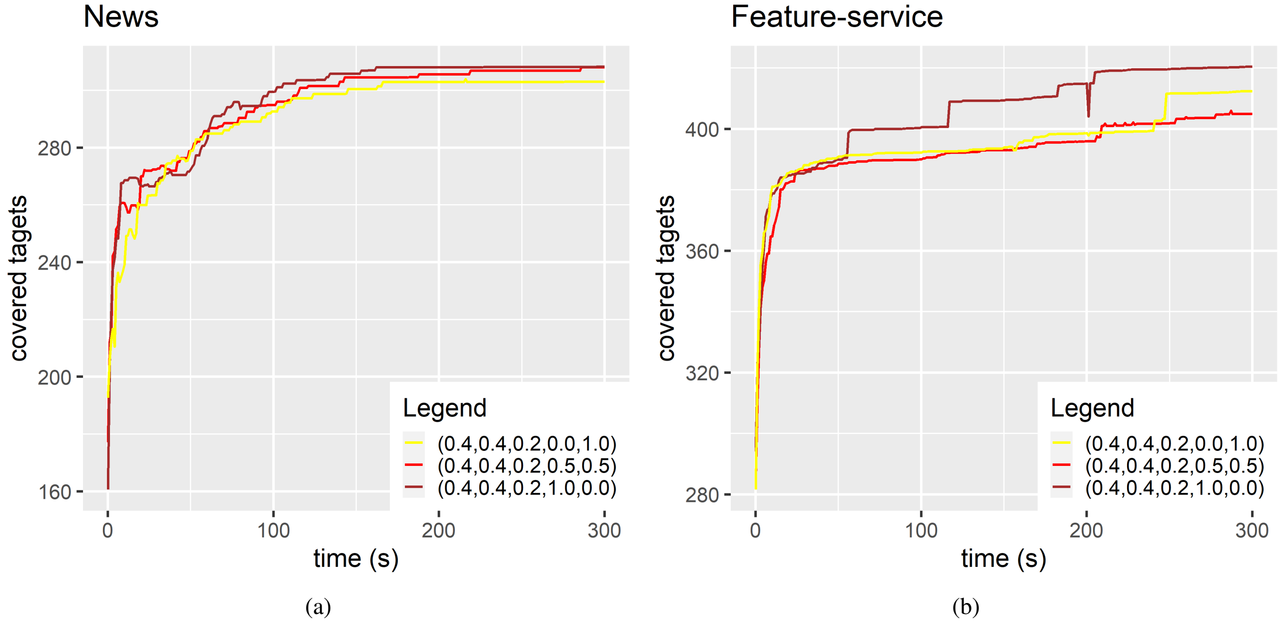


Figure 5: Graphs showing the average amount of covered targets over time for $\mathcal{P}_1$, $\mathcal{P}_6$ and $\mathcal{P}_7$. The parameter sets are written as a 5-tuple. (a) and (b) show the results for the news and the feature-service API respectively.

determined.

4.3 Threats to validity

Threats to internal validity stem mostly from the fact that the algorithm is based on randomness. To mitigate the impact of this randomness, each run is repeated 10 times. Doing more runs would give the results more confidence, but due to time constraints this was not possible. Furthermore, the parser implemented for seeded sampling is by no means a complete implementation. Some existing tests had to be omitted because they were not able to be parsed correctly. This limits overall performance of seeded sampling. Another threat to internal validity arises from the fact that EvoMaster is not a bug-free software package. Numerous benchmark runs had to be rerun because the system crashed in a way that seemed unrelated to the implementations made for this paper.

Threats to external validity stem for the fact that the results are gathered on only two RESTful APIs. This means that there is not much assurance that seeded sampling performs similarly on other APIs in the industry. Only two APIs were tested for similar time constraints as mentioned before, and because of the difficulty of finding open source RESTful APIs using the rest assured library. On top of that, to be able to run EvoMaster in white-box mode, a driver needs to be written, which is a time consuming effort as well.

5 Responsible research

This section briefly covers the integrity of the performed research as well as the reproducibility of the empirical evaluation.

5.1 Research integrity

There is assumed to be no breach of scientific integrity. All data gathered in the empirical evaluation is used for computing the results and no artificial data has been added. There is no conflict of interest since the authors of this paper have no financial stake in the succes or failure of EvoMaster. Furthermore, much effort has been put into preventing pagiarism as much as possible by carefully keeping track of the information presented and giving credit to other authors where credit is due.

5.2 Reproducibility

This research is based on a randomized algorithm. This randomness inherently threatens the reproducibility of the empirical evaluation. To combat this, multiple runs were performed for each set of parameters such that an average could be taken. When reproducing this research, it is improbable that the results will be exactly the same. However, the averages of the result should be similar if the results are reproduced correctly, assuming that enough runs were executed.

Aside from the inherit randomness of the system, the experiments should be fully reproducible. The EvoMaster tool is open source, as well as the additions this paper presents. Therefore, the code written for this paper can be easily accessed and used to reproduce our results.

6 Conclusion and future work

Microservice architectures are increasingly popular with big companies [12], and RESTful APIs are a great way to implement them. To ensure the quality of these APIs, they need to be tested properly. This, however, is challenging.

SBST solves this problem by automatically generating test cases for certain types of software. EvoMaster is a tool that can generate system-level test cases for RESTful APIs [1]. It uses the MIO algorithm as its main search algorithm.

This paper focusses on improving a specific aspect of this algorithm, namely the method used for sampling new test cases. EvoMaster currently employs two ways to sample test cases: Random sampling and smart sampling. We add a third technique called seeded sampling. It uses manually-written test cases written by developers. The tests are first parsed to an internal representation and put in an archive. Then, when seeded sampling is invoked, it either clones or carves a test case from this archive. Cloning means that a random parsed test is directly copied from the archive and put into the search algorithm. When a test is carved from the archive it means that a resource generating sequence (a sequence of POST and/or PUT requests) is randomly taken from a test, to which a random number of other requests is then added.

Seeded sampling has shown that it can improve the coverage achieved by EvoMaster. An empirical evaluation on two RESTful APIs was performed. This study provides significant evidence in support of using seeded sampling when proper parameters are set.

However, further improvements can be made. Future work could focus on developing a parser that can be used for a wider range of test cases. The first step is to investigate dynamic instead of static analysis when parsing the test cases. Furthermore, this paper covers only a small number of parameter sets. More sets should be tested to optimize the generation of test cases. Additionally, parameters that were not considered in this research should be examined. Lastly, a bigger variety of APIs should be evaluated with EvoMaster to increase the validity and reliability of seeded sampling. With these improvements, seeded sampling could be better applied by companies and these results would be more relevant for a wider range of purposes.

References

[1] ARCURI, A. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology 28*, 1 (2019).

[2] ARCURI, A. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technoloy (IST)* (2019).

[3] ARCURI, A., AND BRIAND, L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability 24*, 3 (2014), 219–250.

[4] DERAKHSHANFAR, P., DEVROEY, X., PERROUIN, G., ZAIDMAN, A., AND VAN DEURSEN, A. Search-based crash reproduction using behavioral model seeding. *arXiv preprint arXiv:1912.04606* (2019).

[5] FIELDING, R. T., AND TAYLOR, R. N. *Architectural styles and the design of network-based software architectures*, vol. 7. University of California, Irvine Irvine, 2000.

[6] FRASER, G., AND ARCURI, A. Whole test suite generation. *IEEE Transactions on Software Engineering 39*, 2 (2013), 276–291.

[7] KHAN, M. E., KHAN, F., ET AL. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl 3*, 6 (2012).

[8] KOOPMAN, P., ALIMARINE, A., TRETMANS, J., AND PLASMEIJER, R. Gast: Generic automated software testing. In *Symposium on Implementation and Application of Functional Languages* (2002), Springer, pp. 84–100.

[9] NEWMAN, S. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[10] NIDHRA, S., AND DONDETI, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA) 2*, 2 (2012), 29–50.

[11] PANICHELLA, A., KIFETEW, F. M., AND TONELLA, P. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)* (2015), IEEE, pp. 1–10.

[12] RAJESH, R. *Spring microservices*. Packt Publishing Ltd, 2016.

[13] ROJAS, J. M., FRASER, G., AND ARCURI, A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability 26*, 5 (2016), 366–401.