Guiding automated system test generation for RESTful APIs using log statements

Michael Kemna[1,2] , **Mitchell Olsthoorn**[1] , **Annibale Panichella**[1]
[1]Delft University of Technology, The Netherlands
[2]Cardiology Informatics, Philips Healthcare, The Netherlands

Abstract

Automated generation of system tests for RESTful APIs has been extensively investigated. Previous investigations use either a white box or a black box approach, wherein the quality of the test cases can be assessed on the HTTP response in the prior and also on the results of byte-code analysis in the latter. Both approaches are limited however, as the black box is often under performing, while the white box can only be applied to a limited set of RESTful APIs.

In this paper, we introduce a novel approach where the system under test (SUT) is defined as a container. In this approach, the log output can be used to assess the quality of the test cases. We present a prototype that retrieves all semantically relevant information from the logs using regex patterns, which was subsequently used to maximize the resulting test suite by an evolutionary algorithm.

The container, white box and black box mode were performed on three SUTs to evaluate the effectiveness and performance of these modes. An increase in code coverage was observed in the container versus black box mode in all SUTs ($p < 0.001$, $p < 0.001$ and $p = 0.054$). In all SUTs, significantly less actions could be evaluated by the container as compared to the black box and white box mode.

Our results show promising results for the novel approach outlined in this paper. Importantly, this approach can be applied to any RESTful API that is deployed as a container.

I Introduction

Representational state transfer (REST) is a well-known, tried-and-tested architectural style for any application programming interface (API) [6]. While REST merely consists of a set of constraints, the *OpenAPI* specification provides a standard for RESTful APIs [44]. Many companies offer their services with RESTful APIs and therefore require system tests to test their implementation. Creating system tests is time consuming however, given that the RESTful APIs often have dependencies on other (internal) services (e.g. databases), and often require the overhead of an internet protocol to access (e.g. HTTP) [17].

Numerous strategies are available to test RESTful APIs, as well as verifying that it adheres to the *OpenAPI* model. Importantly, a differentiation is made between black box testing and white box testing [15]. In general, black box testing is defined as testing a system without having insights into the internals of that system under test (SUT). In other words, the tester has access to the input and output of the SUT, without knowing how it came to that output. In contrast, the tester has complete insights into the internals of the SUT in white box testing. In general, white box testing in the field of Computer Science indicates that the tester has access to the source code and/or the byte-code at run-time of the SUT.

Several attempts have previously been published to automatically create system tests for RESTful APIs [9]–[11], [18], [19], [21], [26], [30]. *EvoMaster* is the only known tool that automatically generates system-level test cases using a white box approach [30], [35]. The tool exploits domain knowledge from the *OpenAPI* specification of the RESTful API to create test cases, similar to most black box testing approaches. Moreover, the tool evolves the test cases with an evolutionary algorithm to maximize the effectiveness of the final test suite. The effectiveness of test cases is based on the results of byte-code analysis (e.g. branch and/or code coverage) and (erroneous) return values of the SUT.

While the previous attempts have shown their effectiveness, they remain limited for various reasons. black box testing can be performed on any RESTful API, but the resulting test suites do not come close to the effectiveness of a manually written test suite. In contrast, *EvoMaster* has shown to reach up to 80% code coverage [30]. However, *EvoMaster* in white box mode only accepts java based systems and is therefore limited in its applicability.

In this paper, we introduce a novel container approach that regards the SUT as a Docker container. This approach gives access to the log output of the SUT that can be used to define the effectiveness of test cases, alongside the HTTP response that is also available in a white box approach. Results of our empirical evaluation shows that our approach significantly improved the effectiveness of the resulting test suite, as compared to the black box approach. To the best of our knowledge, this is the first work in test automation that utilizes the log output of the SUT to improve the resulting test suite. This pioneering study shows the potential of this approach and several optimization options are described that may be further investigated in future research.

The remainder of this paper is organized as follows. Section II describes in more detail the global objective of testing and verification, as well as the previously described black box and white box modes available in *EvoMaster*. Section III describes our container approach. In section IV, we report the findings of our empirical evaluation that investigates the effectiveness and performance of a prototype of our approach as compared to the white box and black box mode of *EvoMaster*. Next, the results and limitations are discussed in section V. Finally, our conclusions and recommendations for future work are described in section VI.

II Problem formulation

While this paper is specifically focused on system testing of a RESTful API, we will briefly describes the global objective of software testing and verification. Thereafter, the previously described black and white box approaches are explained in more detail in the light of *EvoMaster*.

a Testing and verification of the system under test

Software testing is arguably as old as computer science itself, starting with the hand proofs of correctness by Turing [1]. Throughout history, software testing has been implemented with various levels of thoroughness, ranging from none to stringent requirements in both aviation [5] and the biomedical field [23]. While software testing and software verification are not the same, their differences are less pronounced on a system level. Indeed, system tests are generally regarded as the primary means for verification [12]. The definition of verification is given at definition 1 according to the Food and Drug Administration (FDA) [4].

Definition 1 (Verification - FDA)
Verification means confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.

The given definition is abstract on purpose, as formal verification of a complex system such as a RESTful API is nothing short of solving the halting problem. A more concrete goal of verification is to ensure that no software failure will occur while in use. Software failures of medical devices in particular can lead to life-threatening scenarios. However, they are also highly relevant in any other product. Indeed, poor quality software has been reported to cost US \$2.84 trillion, of which a large majority of those costs are directly related to software failure [29]. While proving that no software failure will occur is as difficult as formal verification, thorough testing can catch software failures before the software is used in production. In practical terms, (branch) code coverage is often used as an (imperfect) measure that is inversely related to the chance of software failure.

To achieve this goal, several testing approaches are available. *EvoMaster* makes use of an evolutionary algorithm that evolves generations in an iterative manner and selects the individuals that are perceived to be the fittest. In this paper, we will specifically concentrate on the fitness function and the information about the SUT that it has access to. Importantly, the fitness function results in a fitness value that indicates the quality of the test case. The fitness value can be regarded as a vector of targets with corresponding heuristics. The heuristic value ranges between 1 (high quality) and 0 (low quality). In all approaches, the heuristic values are used to maximize the quality of the resulting test suite. The approaches described below differ in the targets that are available to assess, as well as the data that are available to base the heuristics assignment on. In general, the closer the heuristic assignment is related to the effectiveness of the test case, the better the result of the optimization process.

Algorithm 1: Black box fitness function

```
Input:
sut = system under test (black box)
T = test case with $\{a_1, ..., a_m\}$ HTTP request actions
Output: Fitness value (fv) of T
1 begin
2     fv ← HANDLE-SIZE(T)
3     for $a \in T$ do
4         httpResult ← execute a against sut
5         fv ← HANDLE-HTTP(fv, httpResult)
6     end
7     return fv
8 end
```

b Black box approach

A description of a black box that can be subjected to *EvoMaster* using the black box approach is given in definition 2.

Definition 2 (*EvoMaster* black box)
A RESTful API with an HTTP endpoint that can takes an HTTP request as input and will return an HTTP response as output according to a predefined OpenAPI specification.

Similar to most commercial applications that offer black box testing of the *OpenAPI* specification, *EvoMaster* creates targets that should be covered by the test suite based on the expected input and output from the *OpenAPI* specification. Subsequently, *EvoMaster* initializes a starting population of test cases through dynamic programming and estimates the fitness of each test case (see algorithm 1). In the case of the black box mode, the HANDLE-HTTP function adds heuristics to the action based on the HTTP request.

As an example, consider an action that attempts to cover a GET request on endpoint $news/id$. If the status code of the HTTP response is 200, the target is considered covered and the heuristic is set to 1. In contrast, a status code in the form of $4xx$ indicates a bad request and is assigned a low heuristic of 0.1. Unfortunately, the response of the RESTful API often offers little information on the effectiveness of the HTTP request for security purposes. As such, the heuristic assignment remains crude in the black box approach.

Algorithm 2: White box fitness function

```
Input:
sut = system under test (white box)
sc = sut controller with instrumentation
T = test case with $\{a_1, ..., a_m\}$ HTTP request actions
Output: Fitness value (fv) of T
begin
    reset sut with sc
    fv ← HANDLE-SIZE(T)
    for a ∈ T do
        httpResult ← execute a against sut
        fv ← HANDLE-HTTP(fv, httpResult)
        codeResult ← retrieve byte-code from sc
        fv ← HANDLE-CODE(fv, codeResult)
    end
    return fv
end
```

c White box approach

The definition of a white box that can be subjected to *EvoMaster* using the white box approach is given in definition 3.

Definition 3 (*EvoMaster* white box)
A RESTful API compiled to a JVM environment (java 8 or 11) with an HTTP endpoint that can takes an HTTP request as input and will return an HTTP response as output according to a predefined OpenAPI specification, as well as the corresponding byte-code analysis.

With roots originating from *EvoSuite* [13], *EvoMaster* has been created specifically for the purpose of white box testing in JVM environments [24], [25], [27], [28], [30]. The fitness function of the white box mode uses a similar algorithm (see algorithm 2 as the fitness function of the black box mode, but has more data to base the fitness value of test cases on. Importantly, an *EvoMaster* driver is required to control the SUT. The driver injects instrumentation into the SUT during run-time for byte-code analysis. As a result, the fitness function is not only able to base the fitness value on the HTTP response, but also on the byte-code analysis.

Similar to the HTTP response, the fitness function transforms the byte-code analysis to targets with a heuristic between 1 and 0. A heuristic of 1 indicates that the byte-code line is covered and 0 indicates it is not covered. Most importantly, the byte-code analysis makes use of the branch distance to allow for a more accurate heuristic if the line is not yet covered. As an example, consider an example where the target is the TRUE assignment of the conditional statement $x == y$. If $x = 2, y = 2$, the heuristic is 1. Now consider the two instances $x = 2, y = 2000$, and $x = 2, y = 3$. The line is not covered in both instances, With a branch distance of 1998 and 1, respectively. However, the latter instance is closer to cover the target and is therefore given a heuristic closer to 1, while the prior instance is given a heuristic closer to 0.

All targets (lines of code) are known before the start of the evolutionary algorithm. The algorithm can therefore optimize the available search budget to maximize the amount of covered targets. The search space is complex however, and several algorithms have been implemented to search that space with varying strategies. Relevant algorithms include the whole test suite (WTS), many independent object (MIO) [28] and many-objective sorting algorithm (MOSA) [22].

Algorithm 3: Container fitness function

```
Input:
sut = system under test (container)
dc = docker sut controller
T = test case with $\{a_1, ..., a_m\}$ HTTP request actions
Output: Fitness value (fv) of T
begin
    reset sut with dc
    fv ← HANDLE-SIZE(T)
    for a ∈ T do
        httpResult ← execute a against sut
        fv ← HANDLE-HTTP(fv, httpResult)
        logResult ← retrieve logs from dc
        fv ← HANDLE-LOGS(fv, logResult)
    end
    return fv
end
```

III Our contribution

In this section, we describe our novel container approach. We will first describe the overall strategy. Thereafter, we describe in more detail how the log output is parsed and transformed into heuristics such that they can be used by the fitness function of *EvoMaster*.

a Container approach

The definition for a container that can be subjected to *EvoMaster* in container mode is given in definition 4.

Definition 4 (*EvoMaster* container)
A Docker container that allows control of a RESTful API with an HTTP endpoint that can takes an HTTP request as input and will return an HTTP response as output according to a predefined OpenAPI specification, as well as the corresponding log output.

A Docker container can be regarded as a black box that packages up code and all its dependencies [49]. With this definition, the system under test (SUT) could be controlled by an *EvoMaster* driver, similar to the white box approach. In addition, the log output can be gathered from such a Docker deployment, which can subsequently be used by the fitness function. The SUT remains to be a black box though, as *EvoMaster* has no insights in the logic inside the container.

In contrast to the white box approach, the SUT is not required to be compiled to JVM 8 or 11 byte-code. Instead, it can be implemented in any language as long as it is packaged in a Docker container. As such, this is a generic approach that could be applied to more SUTs as compared to the white box approach. For java applications, the tool *Jib* is available to easily transform the application to a Docker container [36].

The algorithm for the Docker fitness function is described in algorithm 3. It is similar to the black box mode, but stands apart by the introduction of a new function call named HANDLE-LOGS. This function will be described in greater detail in the next subsection(s). However, we will first explore log statements in more detail to understand how they can be transformed to an appropriate heuristic.

b Parsing logs

One of the driving forces behind the data mining movement is the large stream of log statements that are available, Several companies such as Elastic [43], Splunk [46] and Loggly [39] offer log analytic tools to gain valuable information from this data stream. In a similar vein, these logs may be valuable to determine the fitness value of a test case in *EvoMaster*. Transforming the data stream to useful knowledge is not a trivial task however, as the log statements often contain unstructured messages. An example is given in log statement 1.

Log statement 1 (unstructured java log)

```
2020-06-03 07:42:25.57 INFO 1 --- [main]
  o.t.s.e.news.app: Started app in 6.33 seconds
```

This log statement cannot be used as an individual target in the current form, given that the same log message a moment later would be considered as different target due to the timestamp, despite being semantically equal. As such, the logs need to be parsed first. Ideally, the log statement would be delivered as a structured log. An example of the same log statement, but in structured form, is given in log statement 2.

Log statement 2 (structured java log)

```
{
  "timestamp": 2020-06-03 07:42:25.57,
  "loglevel": "INFO",
  "pid": 1,
  "thread": "main",
  "class": "o.t.s.e.news.app",
  "message": {
    "template": "Started app in {time}",
    "time": "6.33 seconds"
 }
}
```

Structured logging has become increasingly popular with the advent of log analytics, yet many services in production still log unstructured statements. As such, Logstash uses a plugin name Grok [37] that attempts to destructure log statements through regex expressions. A Grok consists of a set of regex patterns in combination with their semantical names. For instance, the Grok `%{.*:data}` would match the pattern `.*` and tag the result with the name `data`. While this example is simple, these groks can be combined and reused to create more complex regex patterns [40].

c Transform log statements to heuristics

As shown in algorithm 3, the fitness function for the container mode requests the function HANDLE-LOGS after every action. The HANDLE-LOGS function is described in algorithm 4.

Algorithm 4: HANDLE-LOGS function

```
Input:
fv = fitness value
L = list of log statements
G = $\{g_1, ...g_n\}$ of named groks, where
$g_i$ = regex pattern with set $C_i$ named captures
Output: Updated fitness value (fv)
1  begin
2      for log $l \in L$ do
3          for grok $g_i \in G$ do
4              ng $\leftarrow$ name of $g_i$
5              M $\leftarrow$ match g on l
6              for name nc, pattern pc $\in M$ do
7                  if nc $\in C_i$ then
                       /* a match was found in the
                          log statement l using
                          the named grok ng, with
                          the the regex pattern pc
                          with semantic name nc
                          */
8                      id $\leftarrow$ hash from ng, nc, pc
9                      fv $\leftarrow$ ADD-HEURISTIC(fv, id)
10                 end
11             end
12         end
13     end
14     return fv
15 end
```

Algorithm 5: ADD-HEURISTIC function

```
Input:
fv = fitness value with $< target, heuristic >$ map
id = the target id that was found from the log
A = archive with $< target, counter >$ map
Output: Updated fitness value
1  begin
2      A[id]++
3      if id $\notin$ fv then
           /* a target was covered by the test
              case, so a heuristic is created
              that is inversely related with
              the amount of times the log
              statement has been seen before
              */
4          c $\leftarrow$ A[id]
5          heuristic $\leftarrow 1/c$
6          fv[id] $\leftarrow$ heuristic
7      end
8      return fv
9  end
```

The Grok and related semantics that are relevant for the given SUT are expected to be defined by the user. While these patterns could potentially be derived from sufficient test data, this is out of scope for the given article. From line 7 of algorithm 4, all log statements have been transformed to meaningful targets. The next challenge is to transform those targets to heuristics that are related to the effectiveness of the test case. While code coverage cannot be retrieved from these targets, log statement coverage may be related to code coverage. As such, covering all log statements may result in a higher code coverage. To maximize the log diversity, the heuristics assigned to the log statements should encourage *EvoMaster* to explore newly discovered logs, while ignoring log statements that occur frequently.

The assignment of a heuristic to a log target is described in algorithm 5. Importantly, an archive is maintained where the previously discovered logs are stored, together with a count of the amount of times log has been seen before. If the log target is only seen for the first time, the counter and heuristic are both 1. In contrast, the heuristic assignment of a log target that has been seen 100 times before is only 0.01. As such, log statements that have seldom been seen will be selected by *EvoMaster* with a higher chance as compared to the log statements that have frequently been seen.

IV Empirical evaluation

In the following section, the methodology and results of an empirical evaluation of our proposed container mode is described.

The goal of the empirical evaluation is to assess the performance and effectiveness of the container approach in comparison to the conventional black box and white box approach of *EvoMaster*. Our hypothesis is that the log output of the container can be used to improve the effectiveness of *EvoMaster* as compared to the black box mode. To make this possible, an *EvoMaster* driver is required to gain access to the log output. We hypothesize that this will negatively impact the performance of the tool. We expect that the performance will be similar in container mode as compared to white box mode, given that they will both use an *EvoMaster* driver to control the SUT. Finally, we hypothesize that white box mode will still be superior to black box testing.

a Research questions

- *RQ1* - Is the effectiveness of *EvoMaster* significantly improved by including the log statements in the fitness function as compared to the black box mode? The effectiveness of the resulting test suite is defined as code coverage in white box mode.
- *RQ2* - Is the performance of *EvoMaster* non-inferior using the container mode as compared to the white box mode? The performance is defined as the amount of actions that could be evaluated given a fixed search budget.

b Prototype

The container mode was implemented in *EvoMaster*, using the black box mode as starting point. In particular, a SUT controller to start, stop and reset the SUT was added that utilizes the open-source library *Testcontainers* (with a manual lifecycle) [47]. Importantly, Testcontainers allows the retrieval of the logs of the Docker container of interest. The logs, that are given as unstructured strings, are subsequently parsed using *Grok* [48], an open-source library that is used by *Logstash* [37]. The parsed logs are subsequently transformed to a target that can be understood by the *EvoMaster* evolutionary algorithm, as previously described.

c Subjects

For the purpose of this evaluation, SUTs are required that can be used by *EvoMaster* in white box, black box, and container mode. For this purpose, artificial RESTful APIs in the *EvoMaster* benchmark were used for which an *EvoMaster* driver is readily available (the *news*, *scs* and *ncs* SUT). These applications were Dockerized using *Jib* [36] and can be found on Docker hub [41]. For stateful applications, an endpoint was implemented to reset the state through a GET request. The benchmark repository was forked to implement the minor changes that were necessary to make the SUTs applicable for the container mode. The fork can be found on Github [42].

To investigate the impact of logging, a second version of the *news* SUT with more log statements will be subjected to the *EvoMaster* in container mode. In this version (*news:logged*), 16 log statements were added to log validation errors of the user [33].

d Metrics

The following metrics will be measured to investigate the effectiveness and performance of the different modes:

- **effectiveness:** code coverage of the resulting test suite on the system under test.
- **performance:** amount of evaluated test cases in a given search budget.

e Experimental protocol

The behavior and expected results of the container mode are unknown and other experiments using a similar approach to base our expectations on. As such, the *news* SUT was used to explore the behavior of the prototype first. During this exploration phase, the main experiment will be performed on this SUT with several different *EvoMaster* configurations. In particular, the SUT was subjected to *EvoMaster* using the WTS, MOSA and MIO search algorithms. After exploration, the main experiment was performed on all included SUTs.

In the main experiment, the SUT is subjected to *EvoMaster* in white box, black box and container mode. The effectiveness of the resulting test suite of each mode was assessed by the fitness function of the white box mode. The fitness function of the white box mode assesses the line code coverage of the compiled byte-code during run-time. The performance and effectiveness metrics are stored after each test run for further statistical evaluation.

To explore the impact of the amount of log statements of the SUT on the effectiveness of the container mode, the *news* and *news:logged* were both subjected to the container mode. Since lines of code were added to the *news:logged* to facilitate the log statements, this may impact the code coverage in both versions. To avoid this confounding factor, the test suite resulting from both versions were subjected to the fitness function of the white box mode of the *news:logged* version.

The normality of the resulting variables was verified graphically using histograms and verified with the Kolmogorov-Smirnov test if the histogram was inconclusive [14]. Descriptive statistics are described as median with interquartile range (IQR, presented as 50^{th} [25^{th} - 75^{th}] percentile) for non-gaussian data.

Statistical significance in the metrics of interest between the black box mode vs container mode, as well as container mode vs white box mode were investigated using the non-parametric Wilcoxon Rank Sum test with an α-value of 0.05 [3], [20]. A significant p-value for the effectiveness metric between the container mode and black box mode indicates that the null hypothesis can be rejected and the container mode is superior to the black box mode in terms of effectiveness. A non-significant p-value for the performance metric between the container mode and the white box mode indicates that the null hypothesis cannot be rejected and the container mode is non-inferior to the white box mode in terms of performance. A one-way ANOVA or Kruskal-Wallis ANOVA was specifically not used given that we are only interested in possible differences with the container mode and not between the black box and white box mode. Moreover, the effect size of the difference was measured using the the Vargha-Delaney statistic. The Vargha-Delaney statistic classifies the magnitude of the observed difference in either *negligible*, *small*, *medium* or *large* [7].

EvoMaster has several configuration options that could impact the effectiveness and performance of the test results. As described above, the effect of the hyper-parameters that were deemed most relevant will be explored first. It has previously been shown that extensively fine-tuning these configuration can lead to significantly improved results, but the process is time consuming [16]. The objective of this empirical evaluation is to investigate whether our proposed approach is of interest, not to quantify the best performance.

f Results

The container mode of *EvoMaster* was first explored on the *news* SUT in white box, black box, and container mode using the MOSA search algorithm. Each mode was run twenty times with a search budget of five minutes. The code coverage was significantly higher in container mode (35% IQR [34%-35%]) as compared to the results from the black box mode (34% IQR [34%-34%], p<0.001). The effect size was considered *large* by the Vargha-Delaney statistic. As hypothesized, the white box mode remains superior overcontainer mode (39% IQR [39%-39%], p<0.001), see also figure 1. The coverage of the targets that *EvoMaster* uses internally to represent the code coverage follows a similar pattern (see appendix A).

With regards to the performance, the amount of actions evaluated by the container mode (31744 IQR [31365-32807]) is inferior to both black box (113635 IQR [112094-115074], p<0.001, Vargha-Delany *large*) as well as white box (69603 IQR [66470-71121], p<0.001, Vargha-Delany *large*). Surprisingly, *EvoMaster* favored test cases with less actions in the white box mode as compared to the black box and container mode (see appendix A).

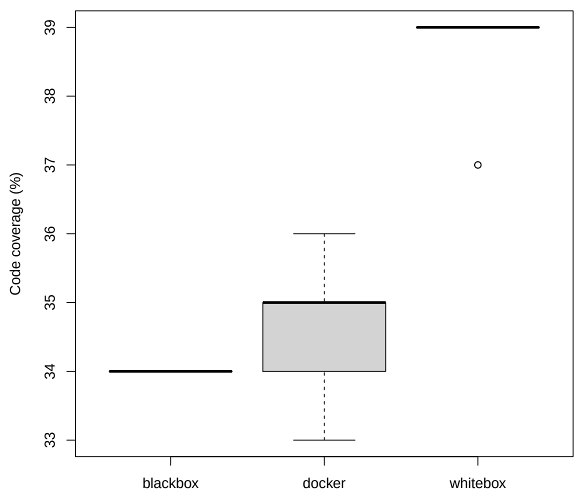

Figure 1: Boxplot of the code coverage (%) on the *news* SUT using the MOSA algorithm.

mode	lines	%	p-value	magnitude
news	*187*	*100*		
container	65	35	*reference*	
black box	64	34	<0.001	*large*
white box	72	39	<0.001	*large*
ncs	*286*	*100*		
container	142	50	*reference*	
black box	137	48	<0.001	*large*
white box	250	87	<0.001	*large*
scs	*301*	*100*		
container	157	52	*reference*	
black box	155	51	0.054	*small*
white box	217	72	<0.001	*large*

Table 1: Median line coverage of the different *EvoMaster* modes on the included SUTs using the MOSA algorithm.

The statistical dispersion observed in the measured outcome metrics was considered adequate to reliably observe results with the given amount of runs and search budget. The experiment was repeated with the WTS and MOSA algorithm, for which similar results were observed (data not shown). As such, the main experiment was performed on all SUTs with the MOSA algorithm and a search budget of five minutes, each mode was run twenty times per SUT. The experimental protocol was automated by a *shell* script, which can be found on Gitlab in the *scripts* directory [34]. Moreover, this directory contains the raw data and test suites resulting from running the *shell* scripts, as well as an *R* script that was used for statistical evaluation.

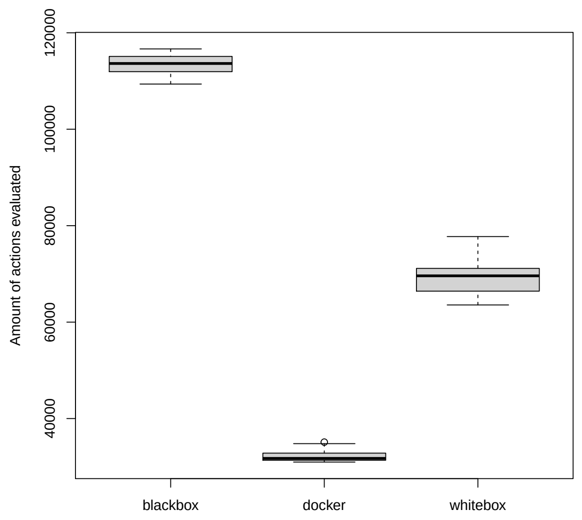

Figure 2: Boxplot of the amount of actions evaluated on the *news* SUT using the MOSA algorithm.

The results of the effectiveness metric of the different modes that were tested on all included SUTs are described in table 1. In the *scs* SUT, a trend towards statistical significance was observed in the increase of effectiveness in container versus black box mode, but this did not reach the α-value threshold. The difference between container and white box mode was more pronounced in the *ncs* and *scs*, as compared to the *news* SUT. With regards to the performance metric, similar results were observed in the remaining SUTs as previously described (see table 2).

No log statements were added by the developer in all SUTs that were evaluated in the main experiment. The effectiveness of the container mode on the *news* SUT was compared with the *news:logged* version, in which 16 log statements were added. The code coverage was significantly higher in the *news:logged* (39 IQR [39-39]) versus the *news* version (35 IQR [34-35], p<0.001, Vargha-Delany *large*).

V Responsible Research

The validity of the problem is discussed in section I. No human subjects were included in the empirical evaluation. Approval of an ethical committee was deemed not to be necessary for the research presented in this paper. The authors do not believe that the scientific contribution of this paper will harm society, nor do we believe that it can be abused in such a manner. Threats to validity, including reproducibility are discussed in section VI. Importantly, the prototype was implemented in an open-source project. All scripts used to generate the data, as well as the actual data are available online [34], [42]. Readers are encouraged to reproduce, evaluate and improve the container mode mode and report their findings in public. Value utilization of the presented approach is discussed in section VII. The authors have no (financial) conflicts of interests to declare.

mode	actions	p-value	magnitude
news			
container	31744	*reference*	
black box	113635	<0.001	*large*
white box	69603	<0.001	*large*
ncs			
container	45053	*reference*	
black box	177045	<0.001	*large*
white box	50353	<0.001	*large*
scs			
container	38726	*reference*	
black box	183680	<0.001	*large*
white box	97207	<0.001	*large*

Table 2: Median amount of actions performed by the different *EvoMaster* modes on the included SUTs using the MOSA algorithm.

VI Discussion

In this paper, we have introduced a novel approach to assess the fitness of test cases for a RESTful API using the log statements of the system under test. Our results show that an evolutionary algorithm that has access to the log output alongside the HTTP response of the SUT can produce a test suite that is significantly better in terms of code coverage than an evolutionary algorithm that only has access to the HTTP response. While our container mode is less effective than a white box approach that has access to the byte-code, our approach can be applied to any container.

In the following sections, any potential threats to validity of our empirical evaluation will be discussed. Thereafter, we will compare the results to previously published experiments.

a Threats to validity

Potential threats to *construct*, *internal*, *external* and *statistical conclusion* validity are described [45]. Threats to *construct validity* are threats related to the quality of choices for the (in)dependent variables that are studied in the current experiment. The dependent variable that was chosen for the effectiveness metric is widely adopted in the literature. It should be noted that complete code coverage does not guarantee that no software failure will occur while in use. Code coverage was measured using the white box fitness function, which is not completely deterministic but observed differences in the code coverage of the same test suite using the white box fitness function multiple times were negligible (data not shown). A potential threat to *construct* validity is that the independent variable may not be adequately represented. Indeed, the container mode prototype may be inadequately implemented, thereby the results of the experiment may not be an accurate representation of the approach outlined in this paper. As such, we believe that the effectiveness of the container mode can be further improved.

Threats to *internal validity* for the presented experiment include possible misconfiguration of the hyper-parameters present for *EvoMaster*. *EvoMaster* was run with default values, which have shown to be valid options [16]. Most importantly, the search algorithm has previously shown to have a dramatic effect on the effectiveness of *EvoMaster* [22], [28].

These hyper-parameters were therefore first explored prior to the main experiment.

A glaring threat to the *external validity* is the fact that the approach was only tested against artificial SUTs. Most importantly, no log statements were added to these SUTs. As such, the only log statements that were captured originated from external packages. The *scs* and *ncs* include several algorithms (string and numerical, respectively) with many lines of code that are not reflected in the log output nor HTTP response. As such, the white box mode performs significantly better in these SUTs than any other mode. Adding log statements to the *news* SUT significantly improved the effectiveness of the container mode. The objective of the empirical evaluation described in this paper was merely to investigate whether our proposed approach is of interest for future research, not to quantify the best performance. The results of our empirical evaluation should warrant further investigation to optimize the approach, after which the performance should be evaluated on more representative applications.

Threats to *statistical conclusion validity* seem limited given the small statistical dispersion of the dependent variables. In addition, the statistical analysis applied in our empirical evaluation are according to current best practices [20].

b Related work

EvoMaster has been extensively studied, in particular with regards to the search algorithm [22], [30]. Importantly, the same SUTs were subjected to *EvoMaster* in a previous publication [30]. In the reported experiment, the code coverage for the *news* SUT was 65%, which is significantly higher than the coverage we have reported. Several factors explain this difference. First and foremost, Arcuri et al. measure the statement coverage through executing the resulting test suite with *IntelliJ IDEA* [38]. In contrast, we have measured code coverage of the compiled byte-code through the fitness function of the white-box mode. We have manually run samples of the resulting test suite in *IntelliJ IDEA*, resulting in a similar statement coverage (data not shown). In addition, *EvoMaster* was run with similar hyper-parameters in both experiments. In the paper of Arcuri et al., *EvoMaster* was run with a search budget of 10K and 100K HTTP requests, while our experiment was run for a search budget of 5 minutes. The search budget of 5 minutes translated to roughly 20k HTTP requests. The other paper executes 100 runs, while our empirical evaluation executes 20 runs. Given that the interquartile range is small in our evaluation, we do not believe that the difference in the amount of runs will be relevant.

Zhang et al. reported a resource-based test case generation approach for RESTful APIs [32]. Domain specific knowledge about the semantics of HTTP methods is exploited by this approach to generate test actions according to a set of effective templates. The results of their empirical evaluation suggests that the resource-based approach may result in an increase in coverage of up to 42%. This approach does not require byte-code analysis and could therefore be implemented in our container mode as well.

Galeotti et al. reported an approach to analyse possible SQL transactions performed by the SUT [31]. This approach can be useful to promote test cases that result in successful SQL transactions, or mutate the database to a correct state. Theoretically, the concept of this approach does not require access to byte-code analysis of the SUT, as it could be implemented as a wrapper around the database. In practice, the prototype as outlined by this publication requires byte-code analysis and is therefore not easily integrated with the container mode.

The presented container mode is more effective than black box mode (*RQ1*), but has lower performance than white box mode (*RQ2*).

VII Conclusions and Future Work

In conclusion, we have shown that the log output of the SUT can be utilized to increase the effectiveness of an evolutionary algorithm to generate system tests for RESTful APIs. The effectiveness of the container mode is increased when more log statements are available in the SUT. The overhead added by *Testcontainers* to deploy Docker containers does reduce the performance of the container mode as compared to white box mode, but also enables *EvoMaster* to be performed on SUTs that are not compiled to a JVM environment.

Do the test suites resulting from *EvoMaster* in container mode satisfy the global objective described in section II? The test suites most certainly do not ensure that no software failure will occur in the SUT. Yet, when is a test suite truly sufficient? To be certified as a medical device by the Conformité Européenne (CE), it needs to be tested in accordance with the *State of the Art* [8]. While the term is abstract, one could consider an evolutionary algorithm as such. Moreover, even a system that is covered by a test suite with 100% code coverage may fail during use. Any test suite should be part of an extensive testing strategy that aims to reach the global objective. In such a strategy, a test suite provided by *EvoMaster* can offer value as an integration test in a continuous development pipeline. In addition, the test suite may uncover server errors and other unintended behavior of the SUT.

While the difference in effectiveness between the container and black box mode is significant, there is room for improvements. Ideally, all semantics that are available in the log statements from the SUT should be captured. We hypothesize that structured logging will capture more semantics of the logs, including parameterized log messages. In particular, the parameter values might give some insights into the logic of the SUT. For instance, the inputs given to the SUT may be related in some way with these parameters. Understanding this relation could guide the mutation of the input parameters in the next generation. In addition, the current implementation only investigates the semantics in each individual log statements. We hypothesize that there may be relevant information between log statements that could offer additional information about the effectiveness of a test case.

The container mode may also be further improved to make *EvoMaster* more accessible. Currently, only Docker containers can be subject to *EvoMaster*. However, the approach outlined in this paper could be applied to other type of containers, as well as complete deployments (e.g. docker-compose or Kubernetes cluster). *EvoMaster* could be hooked onto already existing logging infrastructure to retrieve all logs that are stored centrally, such as an elastic stack [43].

While these suggestions for future work might improve the effectiveness of the mode, will it become non-inferior to the white box mode? The latter has two distinct advantages over the container mode. First, it has access to all lines of code, including those that are not yet covered. Second, it can calculate the branch distance to attribute a higher heuristic that is closer to an undiscovered branch. These mechanisms cannot be exploited by the container mode, as Edger Dijkstra suggested: *"a convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box"* [2]. The container mode may therefore not be able to cover all code lines, yet may be able to cover the code lines leading up to log statements. One could argue that these lines of code may be most important, given that the developer specifically added these log statements. This is particularly true when these log statements are used for audit logging. While white box mode may remain superior in terms of effectiveness, the container mode will remain superior in terms of accessibility.

References

[1] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE", *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. [Online]. Available: https://doi.org/10.1093/mind/LIX.236.433.

[2] T. H. E. O. der Wiskunde and E. W. Dijkstra, *Notes on structured programming*. 1969.

[3] W. J. Conover and W. J. Conover, "Practical nonparametric statistics", 1980.

[4] F. Administration and Drug, *DESIGN CONTROL GUIDANCE FOR MEDICAL DEVICE MANUFACTURERS FDA 21 CFR 820.30 and Sub-Clause 4.4 of ISO 9001*, 1997.

[5] L. A. Johnson *et al.*, "DO-178B, Software considerations in airborne systems and equipment certification", *Crosstalk, October*, vol. 199, 1998.

[6] Roy Thomas Fielding, "Roy Thomas Fielding", *Architectural Styles and the Design of Network-based Software Architectures*, vol. 42, no. 11, 2000, ISSN: 01962892. DOI: 10.1109/TGRS.2004.834800.

[7] 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, ISSN: 1076-9986.

[8] A. 6. 2006, *Medical device software—Software life cycle processes*, 2006.

[9] S. Hanna and M. Munro, "Fault-based web services testing", in *Fifth International Conference on Information Technology: New Generations (itng 2008)*, 2008, pp. 471–476.

[10] S. K. Chakrabarti and P. Kumar, "Test-the-rest: An approach to testing restful web-services", in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, 2009, pp. 302–308.

[11] S. K. Chakrabarti and R. Rodriquez, "Connectedness testing of restful web-services", in *Proceedings of the 3rd India software engineering conference*, 2010, pp. 143–152.

[12] W. L. Oberkampf and C. J. Roy, *Verification and validation in scientific computing*. Cambridge University Press, 2010, ISBN: 1139491768.

[13] 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.

[14] A. Ghasemi and S. Zahediasl, "Normality tests for statistical analysis: a guide for non-statisticians", *International journal of endocrinology and metabolism*, vol. 10, no. 2, p. 486, 2012.

[15] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review", *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[16] 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.

[17] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey", *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.

[18] P. Lamela Seijas, H. Li, and S. Thompson, "Towards property-based testing of RESTful web services", in *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, 2013, pp. 77–78.

[19] P. V. P. Pinheiro, A. T. Endo, and A. Simao, "Model-based testing of RESTful web services using UML protocol state machines", in *Brazilian Workshop on Systematic and Automated Software Testing*, 2013, pp. 1–10.

[20] 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, ISSN: 0960-0833.

[21] T. Fertig and P. Braun, "Model-driven testing of restful apis", in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1497–1502.

[22] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem", in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, IEEE, 2015, pp. 1–10, ISBN: 1479971251.

[23] R. C. Fries, *Reliable Design of Medical Devices*. CRC Press, 2016, ISBN: 9781439894941. [Online]. Available: https : / / books . google . be / books ? id = hhnSBQAAQBAJ.

[24] A. Arcuri, "RESTful API automated test case generation", in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2017, pp. 9–20, ISBN: 1538605929.

[25] ——, "RESTful API automated test case generation", in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2017, pp. 9–20, ISBN: 1538605929.

[26] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of RESTful web APIs", *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2017.

[27] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation", in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2018, pp. 394–397, ISBN: 1538650126.

[28] ——, "Test suite generation with the Many Independent Objective (MIO) algorithm", *Information and Software Technology*, vol. 104, pp. 195–206, 2018, ISSN: 0950-5849.

[29] H. Krasner, "The cost of poor quality software in the us: A 2018 report", *Consortium for IT Software Quality, Tech. Rep.*, 2018.

[30] A. Arcuri, "RESTful API automated test case generation with Evomaster", *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, Jan. 2019, ISSN: 15577392. DOI: 10.1145/3293455. [Online]. Available: https://dl.acm.org/doi/10.1145/3293455.

[31] A. Arcuri and J. P. Galeotti, "SQL data generation to enhance search-based system testing", in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1390–1398.

[32] 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.

[33] *Add logs to the news sut by mjkemna · Pull Request #2 · mjkemna/EMB*. [Online]. Available: https://github.com/mjkemna/EMB/pull/2/files.

[34] *BEP / 2019-2020-Q4 / TestPlus / black-box-testing · GitLab*. [Online]. Available: https://gitlab.ewi.tudelft.nl/bep/2019-2020-q4/testplus/black-box-testing.

[35] *EMResearch/EvoMaster: A tool for automatically generating system-level test cases*. [Online]. Available: https://github.com/EMResearch/EvoMaster.

[36] *GoogleContainerTools/jib: Build container images for your Java applications*. [Online]. Available: https://github.com/GoogleContainerTools/jib.

[37] *Grok filter plugin — Logstash Reference [7.7] — Elastic*. [Online]. Available: https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html.

[38] *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. [Online]. Available: https://www.jetbrains.com/idea/.

[39] *Log Analysis — Log Management by Loggly*. [Online]. Available: https://www.loggly.com/.

[40] *logstash-patterns-core/patterns at master · logstash-plugins/logstash-patterns-core*. [Online]. Available: https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns.

[41] *mjkemna/emb-cs-rest-artificial-news - Docker Hub*. [Online]. Available: https : / / hub . docker . com / r / mjkemna/emb-cs-rest-artificial-news.

[42] *mjkemna/EMB: EvoMaster Benchmark (EMB): a set of web/enterprise applications for experimentation in automated system testing*. [Online]. Available: https://github.com/mjkemna/EMB.

[43] *Open Source Search: The Creators of Elasticsearch, ELK Stack & Kibana — Elastic*. [Online]. Available: https://www.elastic.co/.

[44] *OpenAPI Specification - Version 3.0.3 — Swagger — Swagger*. [Online]. Available: https : / / swagger . io / specification/.

[45] *Overview of Threats to the Validity of Research Findings*. [Online]. Available: http://www.mathcs.duq.edu/~packer/Courses/Psy624/Validity.html.

[46] *SIEM, AIOps, Application Management, Log Management, Machine Learning, and Compliance — Splunk*. [Online]. Available: https://www.splunk.com/.

[47] *Testcontainers*. [Online]. Available: https : / / www . testcontainers.org/.

[48] *thekrakken/java-grok: Simple API that allows you to easily parse logs and other files*. [Online]. Available: https://github.com/thekrakken/java-grok.

[49] *What is a Container? — App Containerization — Docker*. [Online]. Available: https://www.docker.com/resources/what-container.

A Appendix

In the following section, additional data is presented that support the overall conclusions drawn in the paper. The figures and tables shown here are not included in the paper to keep the results and conclusions clear and concise. For transparency purposes, they are presented here.

a Targets

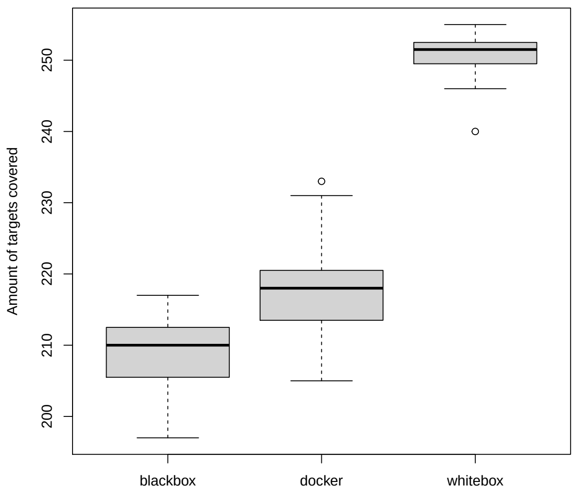

Figure 3: Boxplot of the amount of targets covered on the *news* SUT using the MOSA algorithm.

The targets are the heuristic targets that the evolutionary algorithm of *EvoMaster* is configured to maximize. They represent either HTTP responses, line code coverage and/or log statements, depending on the mode that *EvoMaster* is run with. In essence, the targets are a proxy of the final outcome that is attempted to be maximized. The higher the amount of targets that are covered by *EvoMaster*, the better the effectiveness is expected to be. The results of the exploratory experiment is shown in figure 3, the results of the main experiment on all the SUTs is shown in table 3.

b Tests

In *EvoMaster*, the final result consists of one test suite containing several test cases. A test case subsequently consists of a set of actions that are performed consecutively. The results of the exploratory experiment is shown in figure 4, the results of the main experiment on all the SUTs is shown in table 4.

mode	**targets**	**p-value**	**magnitude**
news			
container	218	*reference*	
black box	210	<0.001	*large*
white box	252	<0.001	*large*
ncs			
container	353	*reference*	
black box	337	<0.001	*large*
white box	539	<0.001	*large*
scs			
container	511	*reference*	
black box	496	<0.001	*large*
white box	681	<0.001	*large*

Table 3: Median amount of targets covered by the different *EvoMaster* modes on the included SUTs using the MOSA algorithm.

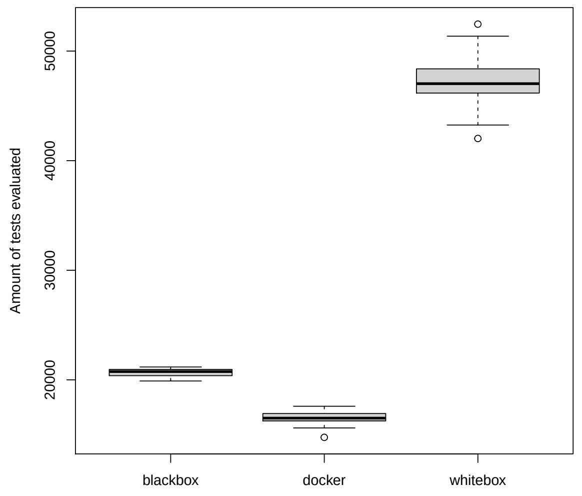

Figure 4: Boxplot of the amount of tests evaluated on the *news* SUT using the MOSA algorithm.

mode	**tests**	**p-value**	**magnitude**
news			
container	16506	*reference*	
black box	20758	<0.001	*large*
white box	47024	<0.001	*large*
ncs			
container	10892	*reference*	
black box	32137	<0.001	*large*
white box	40033	<0.001	*large*
scs			
container	14090	*reference*	
black box	33383	<0.001	*large*
white box	63292	<0.001	*large*

Table 4: Median amount of tests performed by the different *EvoMaster* modes on the included SUTs using the MOSA algorithm.