Testability Transformations For Existing APIs

Andrea Arcuri
Department of Technology
Kristiana University College
Oslo, Norway

Juan P. Galeotti
Depto. de Computación, FCEyN-UBA
ICC, CONICET
Buenos Aires, Argentina

***Abstract*---Search-based software testing (SBST) has been shown to be an effective technique to generate test cases automatically. Its effectiveness strongly depends on the guidance of the fitness function. Unfortunately, a common issue in SBST is the so called *flag problem*, where the fitness landscape presents a plateau that provides no guidance. In this paper, we provide a series of novel *testability transformations* aimed at providing guidance in the context of commonly used API calls. An example is when strings need to be converted into valid date/time objects. We implemented our novel techniques as an extension to EVOMASTER, a SBST tool that generates system level test cases. Experiments on six open-source REST web services , and an industrial one, show that our novel techniques improve performance significantly.**

***Index Terms*---SBST, test generation, testability transformation, system testing, REST**

I. INTRODUCTION

Search-based software testing (SBST) [1], [2] has been shown to be an effective technique to automatically generate test cases. Examples are unit testing of Java software with open-source tools like EvoSuite [3], and testing of mobile applications with the Sapienz tool at Facebook [4].

When doing white-box testing, different techniques are used to define heuristics to smooth the search landscape. The most common in the literature of SBST is the so called *Branch Distance* [5], which is used for example in tools like EvoSuite and EVOMASTER [6] (this latter targeting system test generation for web services). Given a boolean predicate in the code of the system under test (SUT), the branch distance provides a heuristic value to guide the search toward solving such constraints.

Unfortunately, a common issue is the so called *flag problem* [7], where the branch distance is not able to provide any gradient. An approach to address this issue is to transform the code of the SUT to improve the fitness function, with the so called *Testability Transformations* [8]. An example is when dealing with string operations returning booleans [9], like comparisons of two strings for equality. By default, a predicate like `x.equals("foo")` would just be a flag true/false. But testability transformations can be used to replace such functions (or insert probes) to provide better heuristic values, with different kinds of string distances. For example, EvoSuite [3] uses bytecode manipulation to replace all boolean methods in the class `java.lang.String` with its own custom versions. In Java bytecode, there is no specific type for booleans, which are then represented with just an integer: 0 for `false`, and 1 for `true`. So, the functions replaced by EvoSuite return 0 for `true`, and then a heuristic distance for `false`, which is as high as the predicate is far from being evaluated to `true`. After this, EvoSuite needs to modify all the bytecode jumping instructions in which such modified boolean predicate is involved, to handle the fact that now the value 0 represents `true` instead of `false`. Analyzing all possible usages of a boolean predicate could be very complex (e.g., when those are stored in a variable, and such variable is used in a subclass). Therefore, EvoSuite applies such transformations only when those string methods are used directly before a jump instruction (e.g., `if(x.equals("foo"))`), and not when they are stored in a variable (e.g., `boolean b = x.equals("foo")`) or returned from a function (e.g., `return x.equals("foo")`).

In this paper we provide the following contributions:

- A novel approach for testability transformations of commonly used APIs. The presented approach is not limited to transforming methods only before jump instructions and it can also supports methods that throw exceptions (such as `parseInt`).
- A novel technique to track test inputs that are used directly in our transformed methods without modifications. In such cases, feedback is given to the search algorithm to generate the needed data directly.
- A novel testability transformation which is specific for REST web services, and that is able to detect when the SUT is using HTTP query parameters and headers not specified in the schema of the web service. As before, this feedback is then used to include such parameters and headers as part of the search.

We implemented our novel techniques as an extension of the EVOMASTER tool, which is open-source. We evaluated our techniques on six open-source REST web services, and an industrial one. Our experiments show that our novel techniques improve performance significantly, both in terms of fault finding and code coverage.

II. MOTIVATING EXAMPLE

A web service is any application or data source that is available through HTTP/HTTPS. In recent years, there has been a clear shift in industry toward REST (Representational

This work is funded by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385), and partially by UBACYT-2018 20020170200249BA, PICT-2015-2741.

```java
@RestController
@RequestMapping(path = "/api/testability")
public class TestabilityRest {
 @GetMapping(path="/{date:\\d{4}-\\d{1,2}-\\d{1,2}}/{number}/{setting}",produces=APPLICATION_JSON_VALUE)
 public String get(@PathVariable("date")String date,
  @PathVariable("number")String number,
  @PathVariable("setting")String setting) {
  LocalDate d = LocalDate.parse(date);
  int n = Integer.parseInt(number);
  List<String> list = Arrays.asList("Foo", "Bar");
  if(d.getYear() == 2019 && n == 42 && list.contains(setting))
   return "OK";
  else
   return "ERROR";
 }
```

Fig. 1. Example of a REST controller in Java Spring framework, with one HTTP GET endpoint.

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
 TestabilityRest testRest0 = new TestabilityRest();
 try {
  testRest0.get("", "", "");
  fail("Expecting:DateTimeParseException");
  } catch(DateTimeParseException e) {
   verifyException("java.time.format.
    DateTimeFormatter", e);
  }
}
@Test(timeout = 4000)
public void test1() throws Throwable {
 TestabilityRest testRest0 = new TestabilityRest();
 try {
  testRest0.get(null,"TestabilityRest","PATCH");
  fail("Expecting: NullPointerException");
 } catch(NullPointerException e) {
  verifyException("java.util.Objects", e);
 }
}
```

Fig. 2. Tests generated by EvoSuite on the `TestabilityRest` class from Figure 1.

```java
@Test
public void test_0() {
 given().accept("application/json")
 .get(sutUrl+"/api/testability/2019-03-04/42/Foo")
 .then()
 .statusCode(200)
 .assertThat()
 .contentType("application/json")
 .body(containsString("OK"));
}
```

Fig. 3. A test generated by EvoMaster, with our novel testability transformations, on the Spring application where the `TestabilityRest` class from Figure 1 is used.

State Transfer) when developing web services. Figure 1 shows the code of a Java class, in which a REST endpoint is defined using the popular Spring Framework [10]. An endpoint is a location where a REST web service can be accessed. When a HTTP call is made toward such endpoint, three `String` variables are extracted from the URL of the HTTP request. One of the strings is parsed to a date, whereas another to a number. The string `"OK"` is returned only if the date is a valid date with year 2019, the number is 42, and the last string is present in an existing list of strings.

Generating a test case that returns `"OK"` is not trivial, because `LocalDate.parse` and `Integer.parseInt` will throw exceptions when called with non-valid inputs. Furthermore, although there exist testability transformations for `String.equals`, those cannot be directly used in the case of *List.contains*, because API classes loaded with the Java bootstrap classloader cannot be instrumented.

Figure 2 shows the result of running EvoSuite on such class for one hour. Only two low-coverage unit tests were generated, throwing exceptions. This is because EvoSuite has no gradient to generate inputs to reach the `return "OK";` statement.

When dealing with system testing, there is a further complication. When Spring creates a REST handler based on the `@RestController` and `@GetMapping` annotations, it executes a validation check on the `date` variable. In particular, the `date` parameter is checked against the regular expression $\backslash d\{4\} - \backslash d\{1,2\} - \backslash d\{1,2\}$ even before the SUT method `TestabilityRest.get` is called. The check is done deep inside the Spring framework's library, using `Matcher.matches`. Furthermore, a string matching that regular expression is not necessarily a valid date. For example, a value like 42 for the month or day would be wrong (i.e., `LocalDate.parse` will throw an exception), although it would still match $\backslash d\{1,2\}$.

Figure 3 shows one system-level test case generated by EvoMaster when extended with our novel techniques. It was possible to generate the right HTTP call (using the library `RestAssured`) that returns `"OK"` in less than a minute. By applying our novel testability transformations, it was possible to achieve full coverage on such SUT. Nine different test cases were generated covering different scenarios (but not displayed here in this paper due to space limitations).

III. Related Work

"A testability transformation is a source-to-source transformation that aims to improve the ability of a given test generation method to generate test data for the original program" [8]. In the literature, different transformations have been proposed [11], [12], mainly to deal with *flag conditions* [7], [13] (i.e., branches in the code that depend

on the value of a boolean constant). Flags in the code could depend on string operations [9], loop assignments [14], [15], nested predicates [16], and calls to boolean functions [17], [18].

Testability transformations can also be used to generate pseudo-oracles [19], which can be helpful to detect numerical inaccuracies and race conditions. Furthermore, besides SBST, testability transformations can also be useful for dynamic symbolic execution [20].

A testability transformation does not need to preserve the original semantics of the transformed code, as it only needs to *''preserve test sets that are adequate with respect to some chosen test adequacy criterion''* [12]. Most of the aforementioned work apply non-semantics preserving transformations, as targeting test generation for only single target (e.g., a specific branch in the SUT). In contrast, in our work we aim at system testing where whole test suites are generated (e.g., as done in [21] for unit testing) aimed at maximizing coverage over the whole SUT. In such context, many of those non-semantics preserving transformations would not be applicable. For this reason, a main difference in our work is that we must preserve the semantics of the SUT. Furthermore, we provide several novel transformations for API calls that have not be investigated before in the literature.

Besides affecting the fitness score, in our work we applied runtime feedback from the transformed methods to directly modify the genotype of the evolved test cases. This is based on the analysis of how the input to those methods related to the data in the test cases. In some ways, this is related to *taint analysis* in security testing [22].

IV. Testability Transformations using Method Replacements

To collect coverage metrics, tools like EvoMaster need to instrument the code of the SUT. This can be done automatically by manipulating the bytecode of the Java classes when they are first loaded into memory.

In our approach, we implemented a series of custom classes with method replacements for some existing APIs in the JDK. Every time a SUT class is loaded, we check if it is uses any method M for which we have a replacement R. If so, we remove M from the bytecode, and replace it with R. For example, a call to `String.equals` would be replaced by our custom `StringClassReplacement.equals`.

All our replacement methods are static, and have the same return types as their original versions. However, the inputs are different. If the original method is non-static, then the first parameter in R is the caller of the original method. The last parameter is a string object with a unique id, based on the SUT class name and instruction line in which M was called. For example, given in `java.lang.String` the signature:

```
public boolean equals(Object anObject)
```

then our method replacement R for it would have signature:

```
public static boolean equals(String caller,
  Object anObject,
  String idTemplate)
```

A call to `x.equals("foo")` would hence be transformed into a call to `StringClassReplacement.equals` with arguments `x`, `"foo"` and `"name-line"`, where `"name-line"` would be an actual class name and line number where in the SUT the replacement is done. Such `idTemplate` needs to be pushed on the JVM stack before doing the `INVOKESTATIC` call on `StringClassReplacement.equals`.

All of our method replacement Rs have the same semantics as the original version Ms. For the same input, they either give the same output, or throw exactly the same kind of exception.

Every time a replaced R is called at runtime during the search, before returning its value (or before throwing an exception), we compute a heuristic distance. Such distance is similar in concept to the *branch distance*, and it aims at determining how far was the input from making R returning either `true` or `false`. Once such distance is computed, we create two new testing targets based on `idTemplate`, e.g., named `"name-line-true"` and `"name-line-false"`. These new testing targets will be added to the fitness function, and will be part of the search, like any other coverage target (e.g., for lines and branches). In other words, the search will reward the generation of at least one test case in which R evaluates to `true`, and at least one test case in which R evaluates to `false`. These tests can then be useful during the search to cover other targets, like branches.

Besides replacing methods that return a boolean, we also replace methods that can throw exceptions. The bytecode manipulation is the same, it is just that the computed distance has a different meaning, i.e., how far the input was from throwing an exception and not throwing an exception. For example, given the method `parse` in class `java.time.LocalDate` with the following signature :

```
public static LocalDate parse(CharSequence
    text)
```

we provide a replacement class named `LocalDate ClassReplacement` with a method:

```
public static LocalDate parse(CharSequence
    input, String idTemplate)
```

Here, a heuristic distance is computed to check how close a string is from being a valid date in the format `YYYY-MM-DD`. Two new targets are then added to the search.

Technically speaking, such method replacements could be provided for all the methods in the API of the JDK that either return a boolean, or throw an exception if the input is invalid. The challenge then would be to define the right heuristic distances for the different kind of methods. In this paper, we provide method replacements for some of the most common APIs, e.g., all of the ones that were present in the SUTs of our case study.

In EvoMaster, each testing target has a heuristic score in $[0, 1]$, where 1 means a target is covered. All other values represent the target not being covered, where 0 is the worst heuristic score. Values closer to 1 still represent non-covered targets, but they are heuristically closer to cover the target. For each method replacement we create two targets, e.g., one to represent the evaluation of the function to `true`, whereas the other target to represent the evaluation to `false`. If such a call is never reached during the test execution, both targets get score 0. If on the other hand that method call is reached, one of the two targets will necessarily get the score 1 (representing the target being covered), whereas the other target will get a score lower than 1 (representing the target not being covered).

Here, we discuss some of the details of the heuristic computations for the `true` targets. The values for the `false` targets will simply be 0 if the method returns true (and 1 otherwise). Similarly, we discuss the cases of the heuristic computations for the targets representing a method non-throwing an exception. Our method replacements are as follows:

Collection.isEmpty: returns $\frac{1}{1+l}$ where l is the length of the caller collection. If $l = 0$, then the heuristic value is 1. The greater the collection's length, the closer to 0 the heuristic value will be.

String.isEmpty: Returns the same heuristic value that `Collection`'s `isEmpty`.

String.equals: Given two strings, the distance d is calculated as the sum of the character distance between each position in the strings. If string lengths differ, the maximum character distance (i.e., 2^{16}) is used for the missing positions. In turn the heuristic value returned is $\frac{1}{1+d}$.

String.equalsIgnoreCase: The heuristic value is the value returned by `String.equals` using the lowercased versions of both strings.

String.contentEquals: The method compares the caller string to the `toString` representation of the input object parameter. The computed heuristic value is the one returned by `String.equals`.

String.startsWith and endsWith: This method returns true if the string argument is a prefix (respectively suffix) of the caller string; false otherwise. The heuristic value is the one returned by `String.equals` using a prefix (respectively suffix) of the caller string with the argument's length.

String.contains: Computes all the heuristic values of `String.equals` for all substrings of the caller string with argument's length. In turn, the heuristic value for `contains` will return the highest of such heuristic values.

Date.equals: If the compared reference is `null` or is it a non-null value that is not an instance of `Date`, the heuristic returns 0. Otherwise, both `Date` instances are compared as long values through the `getTime()` method. In this case, it returns $1 - \nu(\|v_1 - v_2\|)$ where ν is the normalization function $\nu(x) = \frac{x}{x+1}$ in $[0, 1]$ presented in [23], and v_1, v_2 are the return values of method `getTime()` on each `Date` instance.

Objects.equals: We leverage on the `equals` method replacements previously defined. In case two strings are compared, the `String.equals` heuristic value is returned. Same approach is used for `Integer`, `Date`, `Long`, etc.

Boolean.parseBoolean: This method returns `true` if the input string is equal to the string `true` without considering casing, otherwise it always returns `false`. Therefore, the heuristic for `parseBoolean` returns the heuristic value of `String.equals` to the lowercased input to the constant `"true"`.

Float.parseFloat, Double.parseDouble: Each method parses a string into a `float` or a `double` value respectively. If the string cannot be parsed, a `NumberFormatException` is thrown. If the parsed value it is too large or it is too small (in absolute terms) it will be represented as $\pm\infty$ or ± 0 respectively. For simplicity, we do not currently support the exponential notation (i.e., inputs such as `"9.18E+09"`), our heuristic optimises for the a string containing an optional minus symbol (i.e., `"-"`), a non-empty list of digits (i.e., `"0..9"`) with an optional dot symbol (i.e., `"."`). For each symbol in the current string, the distance of each character (as a char) is compared to the expected symbol for that position. The resulting heuristic value is $b + \frac{1-b}{d+1}$ where d is the distance computed as described above, and b is the base value 0.1 that represents a base value when the input string is not a null value.

Integer.parseInt, Long.parseLong: Each method parses a string into a `int` or a `long` value respectively. Unlike `parseFloat` or `parseDouble`, if the input string cannot be represented within the range of values, a `NumberFormatException` is thrown. The distance d for these methods is similar to the `parseFloat` and `parseDouble` but without expecting any dot symbol. As with `parseFloat` and `parseDouble`, the resulting heuristic value is $b + \frac{1-b}{d+1}$ where d is the distance computed as described above, and b represents a base value for the case when the input string is not a null value (i.e., $b = 0.1$).

DateFormat.parse: This method returns a `Date` instance if the input string is parseable on the date format defined in this object. When the input string is not parseable, the method will signal a `ParseException`. If the pattern of the format matches *"year-month-day"* (e.g., `"2019-10-14"`) or *"year-month-day hour:minute"* (e.g., `"2019-10-14 13:45"`), a distance d is computed that represents how far is the input string to satisfy of the corresponding pattern. In turn, the final heuristic value is computed as $b + \frac{1-b}{d+1}$ where b is the 0.1 base representing the heuristic value if the input string is not null. If the pattern used by the `DateFormat` cannot be obtained, or the format does not match any of those supported (i.e., *"year-month-day"* or *"year-month-day hour:minute"*), we currently provide no gradient.

LocalDate.parse: Similarly to the `parse` method of `DateFormat`, it parses an input string expected in the *"year-month-day"* into a `LocalDate` instance (e.g., `"2019-10-14"`). If the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as with `DateFormat.parse`.

LocalTime.parse: This method parses an input string of the format *"hour:minute"* or *"hour:minute:second"* into a `LocalTime` instance (e.g., `"13:45"`, `"13:45:30"`). Similar to the `LocalDate` parse method, if the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as with `DateFormat`'s `parse` method.

LocalDateTime.parse: This method parses an input string of the format *"year-month-day*T*hour:minute"* into a `LocalDateTime` instance (e.g., `"2019-10-14T13:45"`). Observe that the "T" character in the input string is case insensitive. If the string does not satisfy the expected format a `DateTimeParseException` is thrown. The heuristic value is computed as the aggregation of `LocalDate` and `LocalTime` parse methods.

Date.after and before: This method tests if the current date is after (respectively before) the specified date. The heuristic value for this method is based on the branch distance to $v_1 > v_2$ (respectively $v_1 < v_2$), where v_1, v_2 are the return values of method `getTime()` on each `Date` instance.

Collection.contains: If the argument is an instance of a class with gradient for `equals` (i.e., `String`, `Integer`, etc.), the heuristic value is the highest heuristic between the argument and any instance of that data type stored in the caller collection. If no such values matching the data type are stored, the returned heuristic value is the lowest (i.e., 0).

Map.containsKey: For handling this method, we compute the heuristic value of `contains` over the set of keys obtained by invoking `keySet` method.

Pattern, Matcher and String.matches: For matching regular expressions, we apply the same heuristics as in [9].

Matcher.find: Gradient for the `find` method is provided by wrapping the input regular expression with the leading and trailing regular expression *"(.*)"* and reusing the gradient for the `matches`. With this transformation, the heuristic value will return the value to matching the input regular expression in any substring of the caller input.

When we have a replacement method R for M in a class C, we apply the replacement even for all implementations of M in the subclasses of C. For example, the method replacement for `Collection.contains` is applied to all the collections in Java that inherits from `java.util.Collection`, like for example `ArrayList.contains`.

One case we do not handle is when a SUT class extends a JVM class for which we have replacement methods, and the keyword `super` is used. For example, the SUT might have have a class `MyList` that extends `java.util.Collection`. In its code, it could have a call like `super.contains(x)` inside its `contains` method. In such case, we apply no transformation. The problem is that, in bytecode, that call is an `INVOKESPECIAL`, which can be used only in the subclass. If we made a naive replacement with `CollectionClassReplacement` using `contains(caller, value, idTemplate)` (where `caller` is an instance of `MyList`), then we would not be able to call `caller.contains(value)` after computing the heuristic distance, as we would otherwise end up in an infinite recursion. In the JVM, it is not possible to call the actual implementation of `Collection.contains` on an instance of a subclass `MyList` outside of `MyList` itself. Problem is, that in our implementations of R we still need to call M to make sure that, given the same input, R returns exactly the same value as M.

An approach to solve this issue would be to replicate (e.g., copy&paste) the code of M inside R, and apply Java reflection to manipulate the internal variables of the `caller` instance. However, it would significantly complicate our techniques, making any tool deciding to using them harder to implement and maintain (e.g., if any new JDK release does change such code).

V. Improving Search through Input Tracking

Assume that in the code of the SUT there is a string comparison like `x.equals("foo")`. Now, that variable `x` might depend on some of the inputs of the test case. For example, in the testing of a REST web service, `x` could be a query parameter in the request URL, or a JSON field in the HTTP body payload. Or maybe `x` is not related at all with the input of the current HTTP request. For example, a previous `POST` request could have written such value into a SQL database, and then a HTTP `GET` request would just read it afterwards.

Even if `x` depends directly on the input `z`, the value itself could go throughout several changes before reaching the statement `x.equals("foo")`. For example, different strings could be concatenated. Therefore, using `"foo"` directly as a value for `z` does not necessarily mean that `x.equals("foo")` will be evaluated as true. Here, the heuristic distance plays a major role to guide the search algorithm to find the right value of `z` to get `x.equals("foo")` true.

But what about the cases in which the input value is not changed during execution, and it is used directly as it is in some of our replacement methods (recall Section IV)? Our heuristic distances can provide gradient to the search, but it would be more efficient to just use `"foo"` directly as input.

Our approach is to analyze if any such case does indeed happen, by tracking all the inputs to our replacement methods. If indeed we detect that any such input is used directly in a replacement method, then we can mutate directly the test cases to use the needed values. Recall the example in Figure 1, where based on the REST schema the search evolves three different string variables (*genotype*), which then are used to create a valid URL when a HTTP call is made in the fitness evaluation (*phenotype*). Consider the variable `date`, that based on the REST schema would be of type *string*. Once EvoMaster executes an HTTP call as part of a test case fitness evaluation, the SUT running Spring will read such incoming HTTP request (done in the HTTP server Tomcat), and extract all the needed information (e.g., the *path* component of the HTTP request). Based on the path component, the Spring framework would determine that `TestabilityRest.get` should be the handler for such HTTP request. The string object *date* would then be checked against a regular expression (see Figure 1). And such check would be made in a `Matcher.matches` call. Such method is one for which we do provide a replacement (recall Section IV). As the string variable `date` in the genotype of the test case is used directly in the replacement method for `Matcher.matches`, we could inform the search to directly evolve strings for `date` based on the regular expression $\backslash d\{4\} - \backslash d\{1,2\} - \backslash d\{1,2\}$. As EvoMaster has support for sampling and mutating strings based on regular expressions, this would be more efficient than mutating a standard string and hoping it would match such regular expression by chance.

To achieve this, there are some challenges that we need to solve first. For example, a check on inputs can be outside the main code of the SUT, i.e., in a third-party library. This is the case of `Matcher.matches` deep inside the Spring's library code. On the one hand, we need to make such method replacements even in third-party libraries. On the other hand, we do not want to create new search targets for those libraries, as that is not the software we want to maximize coverage for. The amount of code of the third-party libraries can be much, much more than the code of the SUT (we will show some statistics on this in Section VII). Even if specialized search algorithms such as MIO [24] can handle large numbers of optimization objectives, adding unnecessary search targets could hamper the search. Furthermore, these extra targets could lead to generate more test cases in the final test suite given as output at the end of search, covering different paths in such third-party libraries which might not be of interest to the user. In this case, the solution is relative simple: we apply the method replacements to all classes, both in the SUT and the third-party libraries; however, the new testing targets are only created if the method replacement is done in a SUT class; finally, our *input tracking* in the method replacements is done in all the classes.

A further issue is how to detect that a string value in the execution of the SUT is related to a variable evolved in the genotype of the test case inside the search algorithm. For example, for that endpoint in Figure 1, EvoMaster would have three genes of type *string*. However, what the SUT would see is just an HTTP message read by the HTTP Tomcat server. If the genotype of a test case is for example the three string values $(``a", ``b", ``c")$, what the SUT would read from the TCP socket would just be:

```
GET /api/testability/a/b/c HTTP/1.1\r\n
Host:localhost\r\n
\r\n
```

In other words, we need some way to inform our instrumentation runtime of the three distinct values $(``a", ``b", ``c")$ to track. Unfortunately, as in EvoMaster the search algorithm and the SUT are running on different processes [6], [25], sending such information over TCP before a test evaluation might incur in a non-negligible time overhead. Furthermore, even if such information was sent, then there would still be the problem to determine inside the replacement of `Matcher.matches` if a given string value $``a"$ is part of the test inputs, or if just an unrelated string constant used in the code of the SUT or a third-party library (and so changing the genotype by adding the regular expression info could have negative side effects).

A complete static analysis of the SUT and all third-party libraries might answer this kind of questions. However, for what we need for test generation, a simpler, light-weighted approach would suffice. Given a certain probability P, when in EvoMaster we mutate a string gene, in our approach we rather replace it with the value `"evomaster_x_input"`, where x is a unique number. The idea is to create string values that are very unlikely to be used in the SUT source code, and that then can be easily detected by our replacement methods. For example, we could check if the given input in a replacement method does match the regular expression $evomaster\_\backslash d+\_input$. This has the benefit of not needing to send any extra information to the instrumentation runtime before a test case is evaluated. Although it would be very unlikely that a modified/changed or unrelated input would match that regular expression, even if it happens it would not be a major issue. If the fitness value of the test would not improve, the test case would simply die out during the evolutionary search.

With such an approach, given the test case $(``a", ``b", ``c")$, a string gene like $``c"$ could be mutated into $``z"$ (e.g., a regular, standard mutation in the default version of EvoMaster), while $``a"$ could be mutated into `"evomaster_0_input"` with some probability P. The resulting test case would hence be $(``evomaster\_0\_input", ``b", ``z")$. When this test case is going to be evaluated, the value `"evomaster_0_input"` would be extracted from the HTTP request, and then would end up as input to our `MatcherClassReplacement.matches`, which can recognize it is a variable that should be tracked. Once the test case is completed, the instrumentation runtime would then inform back the search algorithm that the input `"evomaster_0_input"` was matched against the regular expression $\backslash d\{4\} - \backslash d\{1,2\} - \backslash d\{1,2\}$. This information is then used to change the gene type of that input from *string* to *regex*. Next time the test case is mutated, the value `"evomaster_0_input"` would be automatically mutated into a valid string matching that regular expression, like for example `"3156-42-77"`. Further mutations on this gene would still generate strings matching that regular expression, like the `"77"` could be mutated into a `"78"`, but not into a `"7a"`.

Mutating a string into a `"evomaster_x_input"` value can be useful at the beginning of the search, but it could be disruptive for the cases in which we need to evolve complex strings for constraints not handled by our *input tracking* technique. Therefore, given a starting probability P, we gradually decrease it throughout the search down to 0 when in the MIO [24] algorithm (which is the default one used in EvoMaster) its ''focused search'' starts after 50% of search budget is used.

One problem though is that a variable could be used as input in many replacement methods. As soon as a `"evomaster_x_input"` would be mutated into an appropriate string for the first replacement method, the following replacement methods would not be able to recognize it any more, as likely not matching $evomaster\_\backslash d+\_input$.

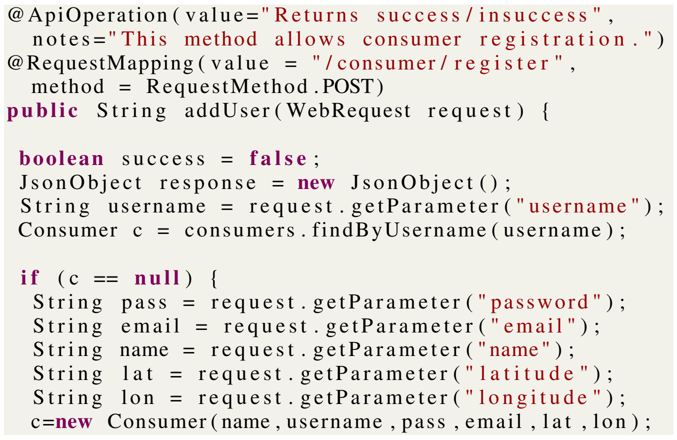
Let us keep considering the case of the variable `date` in Figure 1. After been mutated into something like `"3156-42-77"` after been through `Matcher.matches`, it is used as input to `LocalDate.parse`. However, `"3156-42-77"` is not a valid date. Our novel heuristics in Section IV would give gradient to evolve a string representing a valid date. However, as such input arrives to `LocalDate.parse` unmodified, it would be more efficient to rather inform the search to treat that gene not as a *regex*-type gene, but rather as a *date*-type gene (EVOMASTER has support for mutating and sampling genes representing valid date strings). Such gene would then still be needed to evolve, due to the constraint `d.getYear() == 2019` in the SUT. However, that constraint is numerical (i.e., comparison of two integers) inside an `if` statement, and so would be handled directly by the *branch distance* (i.e., there would be direct gradient to modify the year component of the *date*-type gene into the value 2019, whereas mutations on the month and day components would have no influence on the fitness).

As regular expression constraints are not uncommon in REST web services, and because they can be quite complex, we applied the following approach. If based on our *input tracking* technique we transform a *string*-type gene into a *regex*-type gene, then we will track the values of those *regex*-type genes (and only those) as well besides $evomaster\_\backslash d+\_input$. This means that, before we evaluate the fitness of a test case, we need to send over TCP to the instrumentation runtime the information of which extra values to track, like `"3156-42-77"` in our example.

Not all the replacement methods in Section IV need to handle our *input tracking* technique, like for example methods such as `Collection.isEmpty`. As already stated, parsing of regular expressions would be mapped to *regex*-type genes, whereas parsing of dates would me mapped to *date*-type genes. Strings parsed to numbers are mapped to the respective number gene types, e.g., the input to a `Double.parseDouble` is mapped to a *double*-type gene, but still treated as string, e.g., quoted as a string `"42"` instead of just `42`. String comparisons are mapped to *enum*-type genes, composed of all the constants the input strings are compared to. `String.equalsIgnoreCase` is treated specially, as it gets mapped into *regex*-type gene matching the string constant, but ignoring its case. For example, given the call `x.equalsIgnoreCase("a+b")` where x is an input represented by a *string*-type gene, then our *input tracking* technique would transform it into a *regex*-type gene for the regular expression $(a|A)\backslash Q+\backslash E(b|B)$. In other words, each character with different lower and upper case is in an or `|` between such two cases, while regex control characters are quoted inside a $\backslash Q\backslash E$.

VI. GENOTYPE EXPANSION

To generate test cases for a REST web service, tools like EVOMASTER need to know which endpoints are available, which query parameters they expect, what is the type and structure of the body payloads, etc. All this information can be provided with a schema, where OpenAPI/Swagger is likely the most used [26].



```
@ApiOperation(value="Returns success/insuccess",
  notes="This method allows consumer registration.")
@RequestMapping(value = "/consumer/register",
  method = RequestMethod.POST)
public String addUser(WebRequest request) {

 boolean success = false;
 JsonObject response = new JsonObject();
 String username = request.getParameter("username");
 Consumer c = consumers.findByUsername(username);

 if (c == null) {
  String pass = request.getParameter("password");
  String email = request.getParameter("email");
  String name = request.getParameter("name");
  String lat = request.getParameter("latitude");
  String lon = request.getParameter("longitude");
  c=new Consumer(name,username,pass,email,lat,lon);
```

Fig. 4. Snippet of a REST controller from the class `ConsumerController` in the case study *proxyprint*.

There are two approaches to generate a schema: either by writing it manually, or by generating it automatically from the SUT code using a tool. In both approaches there can be a mismatch between what specified in the schema and what the web service actually does. Even when an automated tool is used to infer the schema from the SUT code, not only such tool might have bugs, but also there might be cases in which it cannot infer the whole correct schema.

In the case of a Spring application, libraries like Springfox can generate the schema automatically, by analyzing static information like annotations such as `@PathVariable` and `@GetMapping` (e.g., recall Figure 1). However, they would not be able to determine information that depends on the code execution of the SUT. Consider for example the code snippet in Figure 4, from the case study *proxyprint*. Here, the needed data from the HTTP request is not injected directly in the controller (which is the common practice, like for example done in Figure 1). On the other hand, a whole `WebRequest` is injected, from which query parameters like `"username"` are then extracted. A schema generation tool such as Springfox might not be able to infer that parameters such as `"username"` should be part of the schema. This is because it would not be enough to just use Java reflection to analyze the annotations on the SUT methods and their parameters. It would require code analysis of the all the method instructions. Furthermore, whether some query parameters are read might depend on non-trivial computation, like for example the `if(c == null)` branch depending on a SQL query into the database, i.e., the `consumers.findByUsername` in Figure 4. In that specific case study *proxyprint*, no query parameter information was inferred by Springfox and added to the schema.

To avoid this kind of issue, we track at runtime all usages of the `WebRequest` objects. This is done by a simple testability transformation in which we store the input values of all method calls like `getParameter` and `getHeader`, by replacing them with our own custom static methods. Once

TABLE I
REST WEB SERVICES USED IN THE EMPIRICAL STUDY. WE REPORT THE NUMBER OF JAVA/KOTLIN CLASSES, LINES OF CODE (LOC), AND NUMBER OF HTTP ENDPOINTS.

Name	Classes	LOC	Endpoints
catwatch	69	5442	23
features-service	23	1247	18
proxyprint	68	7534	74
rest-news	10	718	7
rest-scs	13	862	11
scout-api	75	7479	49
industrial	75	5687	20
Total	333	29032	202

a test case execution is completed, and any such method was called, we *''expand''* the genotype of the test cases by adding genes representing those query parameters. For example, once such `/consumer/register` endpoint is called for the first time with no query parameters (as such info is not in the schema), the test case will get a new gene for the query parameter `username`, initialized with a random string. To avoid modifying the phenotype of an evaluated test, such new genes are marked as a *optional* genes (see [25] for details), off by default. When this test case is going to be mutated during the search, its optional genes could be mutated from off to on (and so added to the URL of the HTTP request), and their string content will be mutated as well as part of the search. This *genotype expansion* is applied every time new headers and query parameters are accessed, like for example when `if(c == null)` will be evaluated as true, and so all the other parameters such as `password` and `email` will be accessed in that code branch.

VII. EMPIRICAL STUDY

In this paper, we aim at answering the following research questions:
RQ1: How often can our testability transformations be applied?
RQ2: How effective are our novel testability transformations on open-source software?
RQ3: How effective are our novel testability transformations on industrial software with complex input validation?

A. Artefact Selection

Our case study is divided in two parts: a set of six open-source REST web services (the same used in our previous work on EVOMASTER [24], [25]), and an industrial web service provided by one of our industrial partners.

Table I shows some statistics like the number of classes (not including tests, nor third-party libraries) and HTTP endpoints in these SUTs. The industrial web service is simply referred with the label *industrial*.

B. Experiment Settings

We carried out two distinct sets of experiments. In the first one, we run EVOMASTER on the six open-source services, considering five different configurations:

TABLE II
NUMBER OF TIMES, PER SUT, IN WHICH A METHOD WAS REPLACED WITH OUR TESTABILITY TRANSFORMATIONS (SECTION IV). WE DISTINGUISH WHETHER THIS HAPPENED IN ONE OF THE CORE CLASSES OF THE SUT, OR IN A THIRD-PARTY LIBRARY. WE ALSO REPORT HOW OFTEN THE TESTABILITY TRANSFORMATIONS WERE USED TO COLLECT INFO ON WHAT ACCESSED FROM THE HTTP REQUESTS (SECTION VI).

SUT	Replacement		HTTP
	SUT	Third-party	
catwatch	34	3973	0
features-service	6	3369	0
proxyprint	113	3507	15
rest-news	4	3618	0
rest-scs	86	2254	0
scout-api	52	3732	0
Total	295	20453	15

- **Base**, the default version of EVOMASTER, with none of our novel techniques presented in this article.
- **M**: using *Method Replacement* (Section IV) and *Genotype Expansion* (Section VI).
- **M+T**: as in M, but also using *Input Tracking* (Section V). For the probability P of applying the mutation, we considered three different values: 0.1, 0.5 and 0.9.

To take into account the randomness of the search algorithm, each experiment was repeated 50 times with different random seeds, for a total of $6 \times 5 \times 50 = 1500$ runs. As stopping criterion, we chose the same amount of fitness evaluations as done in [24], [25], which consisted of 100,000 HTTP calls per run. Depending on the SUT and the hardware employed, each run would take roughly between 20 and 60 minutes.

The second set of experiments on the industrial case study was with only three configurations, i.e., *Base*, M and $M + T$ with the best P from the experiments on the open-source projects. As such industrial case study does complex, heavy input validation on most of its endpoints, we experimented with longer search budget. In particular, we considered a stopping criterion of 500,000 HTTP calls, which roughly required 4 hours per run. Due to the longer execution time, we repeated each experiment only 10 times, for a total of $1 \times 3 \times 10 = 30$ runs.

For both sets of experiments we relied on EVOMASTER with its exact default settings except for the HTTP call budget. The results of the experiments were analyzed following the guidelines in [27]. In particular, we used the Wilcoxon-Mann-Whitney U-test, and the Vargha-Delaney $\hat{A}_{12}$ effect size.

C. Results on Open-Source

Table II shows how often our transformations were applied on the open-source SUTs. These only count the number of times a method was replaced in classes loaded by the SUT during the search. For example, if a class is never used (and so never loaded in the JVM of the SUT), then it is not instrumented by the EVOMASTER runtime, and would not be counted in that table. The numbers reported are the total number considering all the 1500 runs.

The number of applied testability transformations varies significantly from SUT to SUT, like from 4 for *rest-news*

TABLE III
AVERAGE (OVER 50 RUNS) NUMBER OF COVERED TARGETS PER CONFIGURATION, ON EACH OF THE OPEN-SOURCE SUTS. RESULTS ARE RANKED PER SUT, FROM 1 (BEST CONFIGURATION) TO 5 (WORST CONFIGURATION). VALUES IN BOLD ARE THE BEST (I.E., RANK 1).

SUT	Base	M	$M+T$		
			0.1	0.5	0.9
catwatch	1140.7 (5)	1149.7 (3)	**1158.6 (1)**	1151.0 (2)	1148.6 (4)
features-service	586.9 (5)	**602.3 (1)**	593.8 (2)	592.2 (3)	588.7 (4)
proxyprint	1287.2 (5)	1315.3 (4)	**1319.0 (1)**	1316.9 (3)	1317.8 (2)
rest-news	292.3 (5)	296.6 (4)	304.2 (3)	307.8 (2)	**309.6 (1)**
rest-scs	564.8 (5)	683.5 (4)	718.9 (3)	738.2 (2)	**756.8 (1)**
scout-api	1563.6 (5)	**1593.5 (1)**	1580.8 (4)	1583.9 (3)	1588.7 (2)
Average Rank	5.0	2.8	2.3	2.5	2.3

TABLE IV
STATISTICAL COMPARISONS OF THE TWO CONFIGURATIONS *Base* AND $M+T$ WITH $P=0.9$. WE REPORT THE RESULTING $\hat{A}_{12}$ EFFECT SIZES, AND THE p-VALUES OF THE STATISTICAL TESTS.

SUT	Base	$M+T$	$\hat{A}_{12}$	p-value
catwatch	1140.7	1148.6	0.74	≤ 0.001
features-service	586.9	588.7	0.41	0.142
proxyprint	1287.2	1317.8	0.85	≤ 0.001
rest-news	292.3	309.6	0.89	≤ 0.001
rest-scs	564.8	756.8	1.00	≤ 0.001
scout-api	1563.6	1588.7	0.61	0.045

to 113 for *proxyprint*. However, it is clear that there is a very large number third-party libraries involved in the execution of the SUT. On each case study, there are between 2 and 4 thousands method calls that were replaced with our testability transformations. This is not surprising, considering the complexity of handling HTTP requests (e.g., using Tomcat) and accessing databases (e.g., using Hibernate), with beans autowired by Spring.

Regarding our *Genotype Expansion* technique (Section VI), it was applicable 15 times, but only on the *proxyprint* case study.

RQ1: *our testability transformations could be applied 295 times on the SUT classes, and more than 20,000 times on the classes of the third-party libraries.*

Table III shows the results of comparing the five different configurations of our experiments on each of the six open-source SUTs. Performance is based on number of covered targets, which in EVOMASTER includes several different criteria, e.g., code coverage of statements, branches, HTTP status codes and triggered faults.

The *Base* configuration of EVOMASTER was the worst (average rank 5) on every single SUT, whereas $M+T$ (average rank $2.3-2.5$) was better compared to it and M (average rank 2.8). However, there are two SUTs in which M is the best, i.e., *features-service* and *scout-api*. This could be explained if indeed our novel heuristics give better gradient to the search, but, at the same time, they involve inputs that are transformed before used in the replacement methods. In those cases, using string values like `evomaster_x_input` would be just a waste of resources.

The configuration $M+T$ with $P=0.1$ and $P=0.9$ have the same rank 2.3. Between the two, we prefer $P=0.9$, as it is the one with the largest improvements in raw terms (e.g., for *rest-scs*). Table IV shows a more in depth comparison between *Base* and $M+T$ with $P=0.9$. Our novel techniques improve performance significantly in 5 out of the 6 SUTs, with strong effect size, e.g., the maximum possible 1.0 for *rest-scs*, and strong $0.85-0.89$ for *proxyprint* and *rest-news*. For *features-service* there was no strong enough evidence (i.e., the p-value was greater than 0.05) to claim any performance improvement (or loss of it) even with 50 runs.

RQ2: *our novel techniques with configuration $M+T$ and $P=0.9$ provides significant performance improvements on 5 out of the 6 SUTs in the case study.*

TABLE V
AVERAGE RESULTS (OVER 10 RUNS) ON THE INDUSTRIAL WEB SERVICE, WHERE THE *Base* CONFIGURATION IS COMPARED AGAINST $M+T$ WITH $P=0.9$. BESIDES REPORTING THE AVERAGE NUMBER OF COVERED TARGETS (WHICH IS A TOTAL OF SEVERAL DIFFERENT METRICS), WE REPORT AS WELL SOME OF THOSE METRICS, LIKE BYTECODE LINE COVERAGE, PERCENTAGE OF ENDPOINTS FOR WHICH A SUCCESSFUL 2XX STATUS CODE WAS RETURNED, A FAULTY 5XX WAS RETURNED, AND THE NUMBER OF DETECTED FAULTS IN SUCH A WEB SERVICE. WE ALSO REPORT THE RESULTING $\hat{A}_{12}$ EFFECT SIZES, AND THE p-VALUES OF THE STATISTICAL TESTS.

	Base	$M+T$	$\hat{A}_{12}$	p-value
Targets	117.4	380.4	1.00	≤ 0.001
Lines	0.6%	8.2%	1.00	≤ 0.001
HTTP 2xx	5.0%	17.1%	1.00	≤ 0.001
HTTP 5xx	95.0%	95.0%	0.50	1.000
Faults	19.0	24.9	1.00	≤ 0.001

D. Results on Industrial Software

Table V shows the results of the experiment on the industrial case study. On such SUT, the improvements are very large and substantial. Target coverage more than tripled, with line coverage going from not even 1% to 8.2%. It was possible to automatically detect 5 new faults in such web service. On each single metric but HTTP 5xx, we have very low p-values, with the strongest possible $\hat{A}_{12}=1$ effect size.

This web service is composed of 20 endpoints, most of them using input validation based on several regular expressions and date parsing. Without our novel techniques, a tool like EVOMASTER (or any black-box technique) would have extremely low probability of sampling strings that would pass those input validations.

To detect faults, EVOMASTER checks the returned status codes from the HTTP calls, using a 5xx code as a representative for a fault. For example, when there is an exception in the SUT due to a bug (e.g., a null-pointer exception), a framework like Spring would not crash the whole server, and rather would return a HTTP response with 500 as status code. To distinguish between different bugs in the same endpoint, EVOMASTER uses the heuristic of keeping track of the last executed statement in the core classes of the SUT (not of the third-party libraries, like Spring and Tomcat) when a 5xx code is returned.

Based on the results of Table V, we can see that there is 1 out of 20 endpoints for which it is easy to obtain a 2xx code

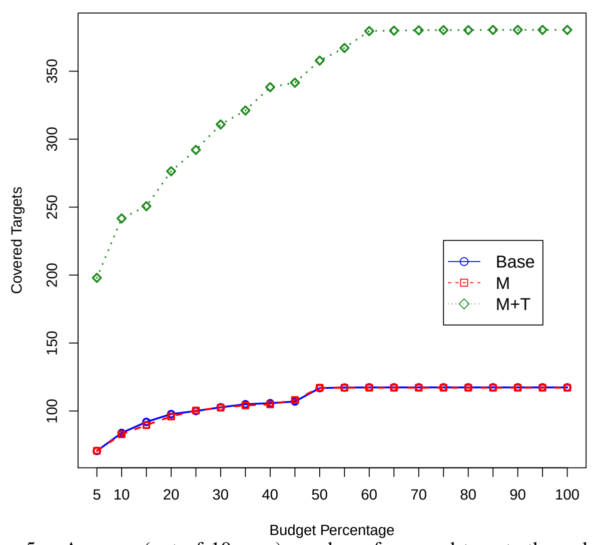


Fig. 5. Average (out of 10 runs) number of covered targets throughout the search, for the three configurations *Base*, M and $M+T$.

(so likely no input validation on it). However, on the other 19 endpoints there are several regular expression checks made by Spring before the SUT main code is even executed. This would explain the very low statement coverage 0.6% for *Base*. However, even with random inputs, it was possible to generate invalid inputs for which the SUT wrongly returns a 5xx status (server error) code instead of a 4xx one (which is used in HTTP to represent user errors). On the other hand, with our novel testability transformations, it was possible to create the right data to at least pass this first layer of input validation.

The performance of a search algorithm strongly depends on for how long it is left running. However, how engineers will use such test generation tools in practice is not yet clear. Would 4 hours be a too long time? Or would engineers be happy to keep the tool running overnight, and collect the results the day after (e.g., after 16 hours)? For fair comparisons, algorithms and their configurations should be compared with different search budgets, possibly representing realistic budgets that engineers would use in practice.

Figure 5 shows the performance of the three compared configurations at each 5% intervals of the search budget (i.e., every 25,000 HTTP calls). There is no performance difference between *Base* and *M*. This is expected, as M has no effect on third-party libraries (e.g., the regular expression checks in Spring). On the other hand, already at 5% of the budget, $M+T$ has performance nearly twice as high as the other configurations at the end of the search (i.e., at 100%).

Even considering the $M+T$ configuration, a line coverage of 8.2% could be considered low. However, this is system testing of a complex industrial web service, where fitness evaluations are expensive, because they can involve several HTTP calls, with access to external databases like Postgres. It would hence be important to repeat these experiments for larger budgets, like the typical 24-hour budget used in the literature of fuzzers (e.g., [28]), to see what kind of code coverage results could be achieved. At any rate, even with the current settings, our results are already of practical importance for our industrial partners, as 5 new faults were automatically found.

RQ3: *with our novel testability transformations, target coverage more than tripled, and 5 new faults were automatically found in the industrial SUT.*

VIII. Threats to Validity

Threats to *internal* validity comes from the fact that that our experiments are based on an extension of the EvoMaster tool. Bugs in such extension could undermine the validity of our results. Although our extension was rigorously tested, we cannot guarantee it is fault-free. To address this problem, and to also enable the replicability of this study and third-party independent validation, our EvoMaster tool is published as open-source on *GitHub*.

Evolutionary algorithms are based on randomized algorithms. To analyze the effects of randomness on the results, we followed the guidelines in [27]. In particular, we used the Wilcoxon-Mann-Whitney U-test, and the Vargha-Delaney $\hat{A}_{12}$ effect size. All experiments were repeated either 10 or 50 times, using different random seeds.

Threats to *external* validity comes to the fact that only seven web services were used in the case study, due to the high cost of running experiments on system testing. We used both open-source services to enable replicability of our study, and an industrial service to study relevance and effectiveness in practice. However, we cannot be sure that our results would generalize to other web services as well.

Our novel testability transformations were evaluated in the context of system testing of web services. Some of those transformations could be used also in other white-box testing contexts, like for example in unit testing. But, without empirical validation, we cannot be sure that our novel testability transformations would be as effective in those other contexts.

IX. Conclusion

In this paper, we have presented a series of novel *testability transformations* to enhance search-based software testing. Experiments on six open-source and one industrial REST web services show that our novel techniques improve performance significantly. For example, it was possible to automatically detect five new faults in the industrial web service.

Future work will aim at providing more transformations for other common API methods. Furthermore, techniques and ideas from taint analysis, seeding (e.g., [29]) and dynamic symbolic execution could be integrated in the search algorithms to improve performance even further.

To enable the replicability of our study, we implemented our techniques as an extension to our EvoMaster open-source tool. To learn more about EvoMaster, visit its website at:

http://www.evomaster.org

References

[1] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, ''A systematic review of the application and empirical investigation of search-based test-case generation,'' *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 6, pp. 742--762, 2010.

[2] M. Harman, S. A. Mansouri, and Y. Zhang, ''Search-based software engineering: Trends, techniques and applications,'' *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[3] G. Fraser and A. Arcuri, ''EvoSuite: automatic test suite generation for object-oriented software,'' in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416--419.

[4] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, ''Deploying search based software engineering with Sapienz at Facebook,'' in *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2018, pp. 3--45.

[5] B. Korel, ''Automated software test data generation,'' *IEEE Transactions on Software Engineering*, pp. 870--879, 1990.

[6] A. Arcuri, ''EvoMaster: Evolutionary Multi-context Automated System Test Generation,'' in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 394--397.

[7] A. Baresel and H. Sthamer, ''Evolutionary testing of flag conditions,'' in *Genetic and Evolutionary Computation Conference (GECCO)*, 2003, pp. 2442--2454.

[8] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, ''Testability transformation,'' *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 1, pp. 3--16, 2004.

[9] M. Alshraideh and L. Bottaci, ''Search-based software test data generation for string data using program-specific search operators,'' *Software Testing, Verification and Reliability (STVR)*, vol. 16, no. 3, pp. 175--203, 2006.

[10] ''Spring Framework,'' https://spring.io.

[11] M. Harman, A. Baresel, D. W. Binkley, R. M. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper, ''Testability transformation - program transformation to improve testability,'' in *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer, 2008, pp. 320--344. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_11

[12] M. Harman, ''We need a testability transformation semantics,'' in *International Conference on Software Engineering and Formal Methods*. Springer, 2018, pp. 3--17.

[13] D. Gong and X. Yao, ''Testability transformation based on equivalence of target statements,'' *Neural Computing and Applications*, vol. 21, no. 8, pp. 1871--1882, 2012. [Online]. Available: https://doi.org/10.1007/s00521-011-0568-8

[14] A. Baresel, D. Binkley, M. Harman, , and B. Korel, ''Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach,'' in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 108--118.

[15] D. W. Binkley, M. Harman, and K. Lakhotia, ''Flagremover: A testability transformation for transforming loop-assigned flags,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 12:1--12:33, 2011. [Online]. Available: https://doi.org/10.1145/2000791.2000796

[16] P. McMinn, D. Binkley, and M. Harman, ''Empirical evaluation of a nesting testability transformation for evolutionary testing,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, pp. 11:1--11:27, 2009. [Online]. Available: https://doi.org/10.1145/1525880.1525884

[17] S. Wappler, J. Wegener, and A. Baresel, ''Evolutionary testing of software with function-assigned flags,'' *Journal of Systems and Software*, vol. 82, no. 11, pp. 1767--1779, 2009. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.037

[18] Y. Li and G. Fraser, ''Bytecode testability transformation,'' in *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. B. Cohen and M. Ó. Cinnéide, Eds., vol. 6956. Springer, 2011, pp. 237--251. [Online]. Available: https://doi.org/10.1007/978-3-642-23716-4_21

[19] P. McMinn, ''Search-based failure discovery using testability transformations to generate pseudo-oracles,'' in *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, F. Rothlauf, Ed. ACM, 2009, pp. 1689--1696. [Online]. Available: https://doi.org/10.1145/1569901.1570127

[20] H. Converse, O. Olivo, and S. Khurshid, ''Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution,'' in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 241--252. [Online]. Available: https://doi.org/10.1109/ICST.2017.29

[21] G. Fraser and A. Arcuri, ''Whole test suite generation,'' *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 2, pp. 276--291, 2013.

[22] E. J. Schwartz, T. Avgerinos, and D. Brumley, ''All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),'' in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317--331.

[23] A. Arcuri, ''It really does matter how you normalize the branch distance in search-based software testing,'' *Softw. Test., Verif. Reliab.*, vol. 23, no. 2, pp. 119--147, 2013. [Online]. Available: https://doi.org/10.1002/stvr.457

[24] ------, ''Test suite generation with the Many Independent Objective (MIO) algorithm,'' *Information and Software Technology (IST)*, 2018.

[25] ------, ''Restful api automated test case generation with evomaster,'' *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[26] ''OpenAPI/Swagger,'' https://swagger.io/.

[27] A. Arcuri and L. Briand, ''A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering,'' *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219--250, 2014.

[28] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, ''Evaluating fuzz testing,'' in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123--2138.

[29] J. M. Rojas, G. Fraser, and A. Arcuri, ''Seeding strategies in search-based unit test generation,'' *Software Testing, Verification and Reliability (STVR)*, vol. 26, no. 5, pp. 366--401, 2016.