Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster

MAN ZHANG, Kristiania University College, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

REST web services are widely popular in industry, and search techniques have been successfully used to automatically generate system-level test cases for those systems. In this article, we propose a novel mutation operator which is designed specifically for test generation at system-level, with a particular focus on REST APIs. In REST API testing, and often in system testing in general, an individual can have a long and complex chromosome. Furthermore, there are two specific issues: (1) fitness evaluation in system testing is highly costly compared with the number of objectives (e.g., testing targets) to optimize for; and (2) a large part of the genotype might have no impact on the phenotype of the individuals (e.g., input data that has no impact on the execution flow in the tested program). Due to these issues, it might be not suitable to apply a typical low mutation rate like $1/n$ (where n is the number of genes in an individual), which would lead to mutating only one gene on average. Therefore, in this article, we propose an adaptive weight-based hypermutation, which is aware of the different characteristics of the mutated genes. We developed adaptive strategies that enable the selection and mutation of genes adaptively based on their fitness impact and mutation history throughout the search. To assess our novel proposed mutation operator, we implemented it in the EvoMaster tool, integrated in the MIO algorithm, and further conducted an empirical study with three artificial REST APIs and four real-world REST APIs. Results show that our novel mutation operator demonstrates noticeable improvements over the default MIO. It provides a significant improvement in performance for six out of the seven case studies, where the relative improvement is up to +12.09% for target coverage, +12.69% for line coverage, and +32.51% for branch coverage.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Software testing and debugging;**

Additional Key Words and Phrases: REST API testing, search-based software testing, test generation, hypermutation

ACM Reference format:
Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 2 (September 2021), 52 pages.
https://doi.org/10.1145/3464940

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

Authors' addresses: M. Zhang, Kristiania University College, Prinsens gate 7-9, 0152, Oslo, Norway; email: man.zhang@kristiania.no; A. Arcuri, Kristiania University College and Oslo Metropolitan University, Prinsens gate 7-9, 0152, Oslo, Norway; email: andrea.arcuri@kristiania.no.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.

1049-331X/2021/09-ART2 $15.00
https://doi.org/10.1145/3464940

1 INTRODUCTION

In industry, Representational State Transfer (REST) web services [33] are widely used for building enterprise applications (e.g., using a microservice architecture [59]) and as well for providing public APIs to access web resources over a network (e.g., internet). Due to their wide use in industry, it would be useful to have an approach to automate system test generation for such services [5]. However, it is quite challenging to automatically test these systems, due to their complexity (e.g., when dealing with communications over a network). One possible approach to tackle these challenges is the use of search-based software testing, which has demonstrated its applicability and effectiveness to automate software testing in several different contexts [1, 35, 41, 53].

To generate white-box tests for such web services with search techniques, the search needs to optimize for a large number of testing targets, e.g., lines, branches, and faults. Furthermore, the evolved individuals (i.e., the tests) need to handle various kinds of actions and data types in their genes (e.g., to represent numbers, strings, and objects). For instance, a REST API typically interacts with a database, and the behavior of the **system under test** (**SUT**) might be affected by the status of the database. To manipulate the status in the context of testing, a test needs to include not only HTTP calls (having query parameters, body payloads, etc.) but also **Structured Query Language (SQL)** commands (as *action*s in an individual shown in Figure 1). In such kind of tests, the number of the SQL commands can be tens or hundreds [12]. In addition, regarding the genes, corresponding to different parameters and data types in REST, they can be defined in various types with additional properties (as *gene*s in an individual shown in Figure 1). For example, a parameter could be in the URL path or an HTTP header, and its type can be primitive, e.g., an integer, or structured, e.g., a reference object (e.g., an HTTP body payload in JSON) which contains a set of fields of various types. To mutate such an individual, considering the quantity and type of genes, it might not be suitable to apply the typical standard mutation rate $1/n$ (where n is the number of genes) used in the literature, based on mutating one gene on average.

In this article, we propose a *weight-based hypermutation* (Figure 1) specialized for handling mutation for such individuals. Instead of employing a uniform mutation rate, the proposed mutation manages to determine a mutation rate customized for each gene based on its own weight (as w_i in Figure 1), and the weight can be calculated based on its own characteristics, e.g., the data type. In addition, considering the huge search space with limited budgets, we increase the average mutation rate (also referred as *hypermutation* in the literature [45, 52]) to improve the exploration of the search landscape with the mutation operator. Instead of mutating one gene on average, the mutation operator decides a number of mutated genes (as calculating ℓ in Figure 1) based on a set of rules, and the number is also linearly reduced (to 1) with the time spent by the search (in a similar fashion as temperature cooling in Simulated Annealing [48]).

Although there might exist a large collection of genes in an individual, not all of the genes might have an impact on the fitness function for the testing targets that are not covered yet. For example, some of the fields in a JSON body payload in an HTTP POST request might just be data that will be saved in a database, with no impact on the execution control flow in the SUT. Mutating the genes that are not related to the testing targets simply results in wasted search budget. Automatically detecting all these cases would be useful (e.g., using program slicing [56]), but its viability and scalability on system testing of web services is still an open research challenge. Therefore, in this article, we further develop an *adaptive* mutation operator that tracks gene evolution information such as mutation history with fitness values and impacts on the testing targets during the search (i.e., the mutated gene has either impact or no impact on the evaluated targets as *impact collection* in Figure 1), then use such tracked information to adaptively mutate these genes during the search (as calculating w with impacts and adaptive gene mutation in Figure 1). For instance, with

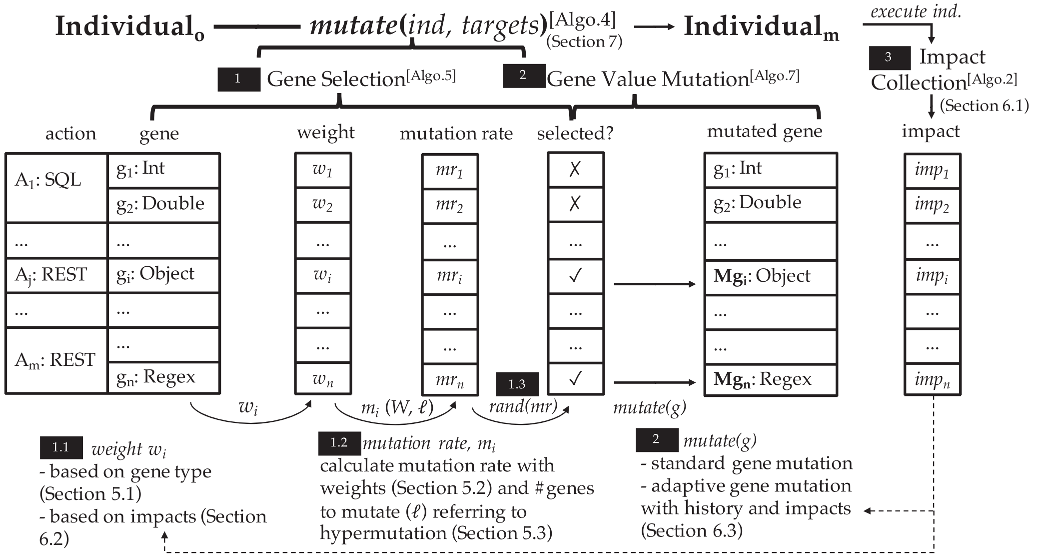


Fig. 1. An overview of adaptive hypermutation for RESTful API Testing.

impact information, we manipulate weights of the genes to guide gene selection for reducing the probability of involving irrelevant genes.

We implemented our approach with the **Many Independent Objective** (**MIO**) algorithm [6] in the testing tool EvoMaster [4, 7]. We named our technique MIO-WH*. EvoMaster is an automated testing tool for REST APIs, which implements several evolutionary algorithms for test generation. To be able to do white-box testing, the tool is capable of instrumenting the SUT with search-based heuristics and analyzing coverage at runtime. However, EvoMaster can also be use for black-box testing [8]. It also has some novel techniques integrated for REST API testing, e.g., handling heuristic on SQL queries [12], SQL data generation, and testability transformation support [13]. MIO is an evolutionary algorithm designed for the system test generation which we have been using in our work [7, 12, 13, 68] for producing more effective tests for REST APIs. Compared with other test generation algorithms (i.e., **Many-Objective Sorting Algorithm (MOSA)** [60] and **Whole Test Suite (WTS)** [36, 61]), its performance on system testing of web services has been studied in [6], showing the best overall results on artificial problems and also real software.

To evaluate our novel techniques, we conducted an empirical study for MIO-WH* by comparing it with a baseline technique (i.e., the default version of MIO) on seven open-source RESTful APIs using three test coverage metrics, i.e., target coverage, line coverage, and branch coverage. Results of our empirical study showed that our novel technique MIO-WH* achieved a significant improvement over the baseline technique on six out of the seven case studies in test coverage. Relative improvements with average coverage are up to +12.09% for target coverage, +12.69% for line coverage, and +32.51% for branch coverage.

The rest of the article is organized as follows. Section 2 introduces related background topics for providing a better understanding of the rest of the article. Related work is discussed in Section 3. In Section 4, we introduce an example to better illustrate the problem we are addressing in this article. The proposed approaches are presented in Section 5 for weight-based hypermutation and Section 6 for adaptive mutation, followed by an integrated representation and implementation in Section 7. The empirical study and experiment results are shown in Section 8. We discuss threats to validity in Section 9, and conclude the article in Section 10.

2 BACKGROUND

2.1 REST and OpenAPI Specification

The REST is a set of design guidelines for building web services on top of HTTP. It was first introduced by Fielding in his PhD thesis [33] in 2000. Now it is widely applied in industry, e.g., Google,[1] Amazon,[2] and Twitter.[3] A RESTful API is an API that follows the REST guidelines to manipulate resources with stateless operations over the network using HTTP. For instance, when following the REST guidelines, the operations should be implemented in accord with the protocol semantics of HTTP, e.g., you should not delete a resource when handling a GET request.

The OpenAPI Specification[4] defines a standard to describe RESTful APIs, using a schema document in JSON or YAML format. The schema allows both humans and machines to understand how to access the services and resources provided by the API. Figure 2 shows a snippet example of an OpenAPI/Swagger JSON definition for a RESTful API. As shown in the example, the `foo` resources can be retrieved with GET, created with POST, and removed with DELETE through a resource path `/foos/{x}`. To make a valid HTTP request, for each operation, parameters are required to be defined with their location (`in` body, query, or path), their type (`type` or `schema`), and optionality (`required`). Note that the type can be a primitive (e.g., integer for `x` and string for `y`) or structured (e.g., `Info Object` for `z`). Moreover, for the request, a set of possible responses are defined with HTTP response codes.

In the context of automated test generation, such a standard provides a clear machine-readable schema which can be used to create well-formatted HTTP requests to the SUT. When using search-based software testing techniques, test cases (i.e., HTTP requests) will be evolved following the constraints of the grammar defined by these schemas.

2.2 Hypermutation

The hypermutation operator can be regarded as one of the distinguishing features (compared with other evolutionary algorithms) of Immune Algorithms that employs mutation at a high rate [45, 52] (i.e., above the typical $1/n$ rate) and can explore the fitness landscape by introducing diversity into a population [22, 24].

The Immune Algorithms are inspired by biological immune systems. One kind of such algorithms follows the clonal selection principle [19, 25] which explains how cells in the immune system function in response to an antigenic stimulus (e.g., pathogens). In the immune system, there exist two kinds of cells, i.e., B-cells and T-cells, and both of the cells have receptors on them for recognizing antigens (the B cell receptor is also called an antibody) [21]. Affinity measures a degree of the recognition, i.e., binding between an antibody receptor and antigen [21]. The process can be simplified as [20]: (1) Selection: the B-cell with a better affinity is selected; (2) Proliferation: the selected B-cells produce many offspring by cloning themselves; (3) Affinity Maturation: clones are employed with a mutation at a high rate mutation (known as hypermutation) to differentiate them from their parents; and (4) Reselection: the mutated clones are reselected for ensuring that the strongest cells with a higher affinity are retained. The whole process is performed iteratively with the aim of evolving the antibodies with the highest affinity. Theoretical results on the effectiveness of the different features of Immune Algorithms have also been formally proved [23].

The use of higher mutation rates has also been explored in traditional evolutionary algorithms like (1+1) EA, in the so called "**Heavy-Tailed Mutations**" (**HTM**). For example, Friedrich

[1] https://developers.google.com/drive/v2/reference/.
[2] http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.
[3] https://dev.twitter.com/rest/public.
[4] https://swagger.io/.

```
"/api/foos/{x}": {
  "get": {..},
  "post": {
    "tags": [..],
    "summary": "createFoo",
    "operationId": "createFooUsingPOST",
    "consumes": ["application/json"],
    "produces": ["*/*"],
    "parameters": [
      {
        "in": "body",
        "name": "z",
        "description": "z",
        "required": true,
        "schema": {"$ref": "#/definitions/Info", "originalRef": "Info"}
      },{
        "name": "x",
        "in": "path",
        "description": "x",
        "required": true,
        "type": "integer",
        "format": "int32"
      },{
        "name": "y",
        "in": "query",
        "description": "y",
        "required": true,
        "type": "string"
      }],
    "responses": {
      "200": {..},"201": {..},"401": {..},"403": {..},"404": {..}
    },
    "deprecated": false
  },
  "delete": {..}
},
```

Fig. 2. Snippet example of OpenAPI/Swagger document in JSON for a RESTful API.

et al. [37, 38] studied HTM for (1+1) EA applied on some pseudo-Boolean artificial landscapes and combinatorial problems such as the Minimum Vertex Cover and the Maximum Cut. Mironovich and Buzdalov [58] studied (1+1) EA with HTM on the Maximum Flow problem. Lengler [50] studied higher mutation rates for different variants of Genetic Algorithms when applied to the optimization of the monotone function family HotTopic. Ye at al. [66] introduced a "normalized standard bit mutation", and investigate it in the context of (1+λ) EA applied on the functions OneMax and LeadingOnes. Antipov et al. [3] investigated the use of HTM on $(1 + (\lambda, \lambda))$ Genetic Algorithm applied to Jump functions.

Hypermutation, and Immune Algorithms in particular, have been applied to solve various optimization problems, including multi-objective optimization problems [20, 42, 52]. In this article, we developed a novel hypermutation specific to test generation for RESTful APIs which faces many objectives (e.g., thousands of lines and code branches) to be optimized. The hypermutation manages to determine an average number of genes to be mutated based on a number of candidates and time spent by the search. Some of the different hypermutation mechanisms discussed in the literature could be adapted and applied in this context as well. But whether they are going to be effective in the context of test data generation will be a matter of future empirical evaluations.

2.3 Parameter Control

In search algorithms, there can be several parameters that need to be configured, like for example population size and cross-over rate in Genetic Algorithms. There are different strategies to choose such settings [26, 32, 46], like a static setting of those parameters, or let them vary dynamically during the search.

Regarding static settings, no settings is best on all possible optimization problems [65], although default values recommended in the literature do seem to be performing relatively well on software engineering problems [11, 49, 62, 67]. Regarding dynamic parameter control, it is less used, and not just in the software engineering literature: *This overview revealed a great number of interesting publications with promising results. Meanwhile, we also noted a disappointing discrepancy. In theory, parameter control mechanisms have a great potential to improve evolutionary problem solvers. In practice, however, the evolutionary computing community did not adopt parameter control techniques as part of the standard machinery, controlling EA parameters on-the-fly is still a rather esoteric option, with self-adaptation in evolution strategies being the exception that confirms the rule* [46].

Another complementary approach is to base the parameter tuning on features of the addressed problem instance, in the so called "**Per Instance Algorithm Configuration**" (**PIAC**) [43, 44, 51]. Parameter models can be inferred via experimentation, and then used to decide which parameter settings to use on each new instance of the addressed problem.

In this article, we use an adaptive parameter control for some of the configuration settings, like the hypermutation. The idea is to have a high mutation rate to reward the *exploration* of the search landscape at the beginning of the search, and then gradually reducing it to rather reward the *exploitation*. The mutation rate is also updated per-gene based on fitness feedback. We do not build PIAC models, but some parameters depend on problem instance features, like the number of genes related to SQL data.

2.4 The MIO Algorithm

The MIO algorithm [6] is an evolutionary algorithm specialized for generating system tests, and its pseudo-code representation is shown in Algorithm 1. The algorithm was proposed for handling white-box system testing, where it aims at exploiting many specific characteristics of such a problem domain. For example, there are many targets to optimize for, possibly in the order of thousands. The final output is not a single *individual* (i.e., a test case), but rather a population (i.e., test suite). An individual can cover one or more targets, and its representation is of variable length (e.g., a sequence of HTTP calls in the context of RESTful APIs). The goal is to maximize the number of covered targets, where test length is a secondary objective to optimize for. There is a strong dependency relationship between targets (e.g., nested code blocks), and not all targets are feasible (e.g., code that cannot be reached). However, each target can be sought independently in different individuals. Furthermore, the input representation can be very variegated, including simple booleans, numbers, strings, regular expressions, arrays, objects, and so on. These are just some of the reasons why a new evolutionary algorithm like MIO [6] was designed.

MIO's fitness function is designed based on testing targets, such as lines, branches, and faults, to be maximized. Such testing targets are either "not-covered", "reached", or "covered". For example, when a `if(condition)` is executed but the `condition` is not satisfied, then in this case, the `if` true-branch is reached (but not covered) whereas the `else` branch is covered (as well as reached). For each testing target, there exists a population of candidate tests aimed at covering that corresponding target. Note that a test is usually related to multiple test targets (e.g., executing several lines/branches in the SUT, where each line/branch is a distinct test target).

ALGORITHM 1: Pseudo-Code of the MIO Algorithm [6]

```
Input  : Stopping condition C, Fitness function δ, Population size n, Probability for random sampling
         P_r, Start of focused search F
Output : Archive of optimised individuals A
1  T ← SetEmptyPopulations( )
2  A ← {}
3  while ¬C do
4      if P_r > rand( ) then
5          p ← RandomIndividual( )
6      else
           // feedback directed sampling
7          p ← SampleIndividual(T)
           // applying mutation on a sampled individual p
8          p ← Mutate(p)
       // updating archive A with evaluated individual p
9      foreach element k of ReachedTargets(p) do
10         if IsTargetCovered(k) then
11             UpdateArchive(A, p)
12             T ← T \ T_k
13         else
14             T_k ← T_k ∪ {p}
15             if |T_k| > n then
16                 RemoveWorstTest(T_k, δ)
       // exploration/exploitation control
17     UpdateParameters(F, P_r, n)
```

As shown in the depicted algorithm, the search is started with empty populations. MIO is inspired by the (1+1) Evolutionary Algorithm [30] and by the Genetic Algorithms [40]. During the search, MIO either samples a new test at random (line 5) or mutates an existing test from one of the populations (lines 7–8) with a probability P_r. The sampled/mutated test is evaluated based on its execution results. If it achieves any improvement on any testing target, the test will be added to all the active populations. However, each population per target has a limited size n. Once the size exceeds this limit, the test with the worst performance for that specific target will be removed.

When executing a test, one or more new testing targets might be identified. Then, a new population containing the executed test will be created for that target. Moreover, if a target is covered, MIO no longer considers that target in the following optimization search. The associated population size is shrunk to one, and no more sampling is allowed from that population. At the end of the search, the MIO algorithm outputs a set of test cases based on the best tests in each of the populations.

In the context of white-box testing, test coverage (e.g., line coverage) is a typical criterion to evaluate the effectiveness of the tests. Thus, it is more important to produce tests that cover targets rather than tests that are just heuristically close to covering such targets. Therefore, a *feedback-directed* sampling was developed in MIO that guides the search to focus the sampling on populations that have recent improvements. Such sampling enables an effective way to reduce the search budget spent on infeasible targets [6]. In addition, MIO introduces a *focused search* that starts to focus more on the exploitation of the search landscape by reducing the probability

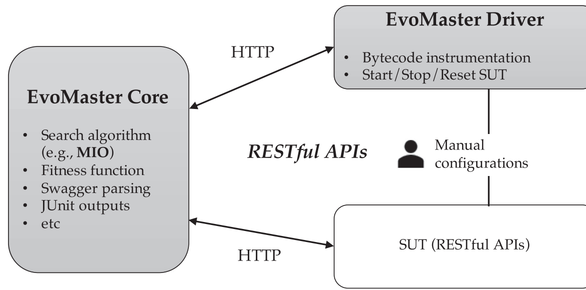


Fig. 3. An overview of EvoMaster.

of random sampling P_r once the search reaches a given certain time point F. This is done with a **deterministic parameter control** (***dpc***) to decrease P_r linearly to 0 based on the spent search budget. This allows the later stages of the search to focus more on trying to cover the currently reached targets, instead of exploring more areas of the search landscape with new random tests.

2.5 EvoMaster

EvoMaster is an open-source tool [15] (currently hosted on GitHub, as well as on Zenodo [14]) that can automatically generate system-level test cases for RESTful APIs, using evolutionary algorithms, e.g., MIO and MOSA. A high-level overview of EvoMaster is represented in Figure 3. It is composed of two main components: (1) *core* which implements different search algorithms to generate tests, defines fitness functions for maximizing testing coverage and fault finding, extracts and parses schemas for REST APIs, outputs executable tests with the specified language (e.g., Java and Kotlin), and so on; (2) *driver* which is developed to manipulate SUTs, e.g., for instrumenting the SUT and starting/stopping/resetting it. With such a *driver*, EvoMaster is capable of obtaining the testing coverage (e.g., covered statements) and search-based heuristics (e.g., branch distance) at runtime, which are used to evaluate the evolved tests during the search.

EvoMaster is now integrated with several novel techniques for testing REST APIs. To test REST APIs that interact with databases, SQL handling [12] was implemented in EvoMaster that defines heuristics on SQL queries, and further enables the search to employ such heuristics as objectives to optimize. In addition, the SQL handling is able to produce SQL data directly as part of the tests, to directly manipulate the state of the databases (if any) during the search. Moreover, to better guide the search in the context of white-box testing, EvoMaster employs a novel *testability transformation* [13] that is able to transform the code of SUT with the aim of providing better heuristic values. Instead of generating sequences of HTTP calls just at random, EvoMaster can also exploit dependencies among the resources in the OpenAPI schemas [68]. To make its adoption easier among practitioners, EvoMaster also supports black-box testing [8], which is easier to setup (as it does not require the manual configuration for the driver).

EvoMaster with those novel techniques serves as a comprehensive platform to test REST APIs.

3 RELATED WORK

For RESTful web services, most of the existing work related to automated test generation employs black-box testing. Recently, interest has been growing in the research community regarding the challenges of automatically generating tests for REST APIs using different variants of random testing, given an OpenAPI schema [16, 17, 31, 39, 47, 54, 63]. For instance, Atlidakis et al. [17] proposed the tool *RESTler* that is capable of inferring dependencies based on OpenAPI specifications,

analyzing dynamic feedback from responses (e.g., status code) during test execution. Further, Godefroid et al. [39] designed a set of techniques (including schema fuzzing rules, rules selection, and value rendering strategies) which is built on the top of *RESTler* for generating test data for requests. Viglianisi et al. [63] developed an approach to generate tests with considerations of data dependencies among operations and operation semantics. In such work, the dependencies are specified with an *Operation Dependency Graph* which is initialized with an OpenAPI schema and evolved during test execution. Karlsson et al. [47] developed a property-based test generation approach using OpenAPI schemas, and Ed-douibi et al. [31] introduced a model-based test generation approach to generate tests with models derived from those OpenAPI schemas. Lopez et al. [54] proposed the *RESTest* open-source tool, that can exploit inter-parameter dependencies in the HTTP calls when generating test cases.

By employing white-box information (e.g., coverage), Atlidakis et al. [16] developed *Pythia* which is composed of a grammar fuzzer for fuzzing the OpenAPI schema, and a strategy to mutate the grammar rules for introducing noise (e.g., GET → GKT) into the produced tests. Then, coverage is measured when executing the produced tests which is used a feedback to decide the tests to be mutated next. To obtain the coverage, *Pythia* requires performing a static analysis on the source code and further manual configuration before testing. With our approach, such coverage information can be obtained automatically by code instrumentation with EvoMaster.

Note that, for developers, it is not necessary to manually write an OpenAPI schema before using any of these tools on their web services. Depending on the applied libraries/frameworks for building RESTful web services (e.g., with the popular Spring in the Java ecosystem), the OpenAPI schemas can be automatically generated (e.g., using libraries such as SpringFox and SpringDoc). So, the lack of an existing OpenAPI schema in general does not hinder the use of these tools.

Our work is different, as EvoMaster uses evolutionary algorithms and support white-box testing in addition to black-box testing. Furthermore, to the best of our knowledge, it is the only tool that can directly generate test data for SQL databases as part of the test cases. The large number of decisions when generating SQL data (besides HTTP calls) complicates the test generation even further, and requires more efficient search algorithms. In this work, we improved the search-based engine of EvoMaster by designing an adaptive mutation specific to REST API white-box testing, by selecting and mutating genes based on their gene characteristics and fitness impact identified during the search. Last, but not least, another key difference with existing work on test generation for REST APIs is that, like *RESTest* [54] and, recently, *RESTler* [17] as well, our tool EvoMaster is open-source [15] and freely available on GitHub since 2016 (http://www.evomaster.org), and on Zenodo [14]. Furthermore, EvoMaster is actively supported, with extensive user manual documentation and video demonstrations. This is essential to enable replicated experiments and support validation from third-party researchers, in addition to making these research results usable and available among practitioners.

There is a large body of literature about the use of search algorithms to solve software engineering problems [41], in particular test case generation [1, 55]. There is no search algorithm which is best on all possible problems [65]. Therefore, for each specific problem domain, there has been research effort to evaluate and compare with existing search algorithms, and designing new custom ones that try to exploit as much domain knowledge as possible. For example, in the context of *unit* test generation, the open-source tool *EvoSuite* [34] has been used to compare with several different search algorithms [18], and evaluate new ones like MOSA [60]. Our work in this article is of similar nature, in which we designed a new variant of the MIO search algorithm, exploiting domain knowledge related to system test generation. We focus on the testing of REST APIs, which

```java
@RequestMapping(value = "/foos/{x}", method = RequestMethod.POST,
  consumes = MediaType.APPLICATION_JSON)
public ResponseEntity createFoo(@PathVariable(name = "x") Integer x,
  @RequestParam String y, @Valid @RequestBody Info z) {
  if (fooRepository.count() < 3) // -- a state of database
    return ResponseEntity.status(400).build();
  if (x < 0 || fooRepository.findById(x).isPresent()) // -- x
    return ResponseEntity.status(400).build();
  if (!y.toLowerCase().equals("foo")) // -- y
    return ResponseEntity.status(400).build();
  String response = "B0"; // else-branch as B0 -- z.c
  if (z.c == 100) // B1 -- z.c
    response = "B1";
  else if (z.c == 200) // B2 -- z.c
    response = "B2";
  else if (z.c == 300) // B3 -- z.c
    response = "B3";
  LocalDate date = LocalDate.parse(z.t);
  if (date.getYear() == 2020) // B4 -- z.t
    response += "B4";
  if (fooRepository.findById(42).isPresent()) // B5 -- a SQL INSERT
    response += "B5";
  FooEntity node = new FooEntity();
  node.setX(x);
  node.setY(y);
  node.setZ(z);
  fooRepository.save(node);
  return ResponseEntity.ok(response);
}
```

Fig. 4. `foos/{x}` endpoint with POST.

is a context that has not been investigated before in the literature of search-based software testing, and that is quite different from unit testing for example.

4 MOTIVATING EXAMPLE

In this section, we provide a simple REST API as an example (available online)[5] to illustrate our addressed problem, i.e., test generation for RESTful APIs in the context of white-box testing. That REST API provides a set of APIs to access `foo` and `bar` resources (e.g., POST for creation, GET for retrieving) which also interact with a SQL database. Figure 4 (complete code in `FooRestAPI.java`) represents an implementation of an endpoint with `POST` action that attempts to create a `foo` by a path `foos/{x}` with SpringBoot.[6] Note that with SpringBoot, a REST API schema can be automatically generated from the implementation, e.g., the snippet schema in Figure 2 was automatically generated corresponding to this implementation. That endpoint requires specifying three parameters, i.e., *path parameter* `x` with type `Integer`, *query parameter* `y` with type `String`,

[5]The full code of the REST API can be found in the `adaptivehypermutation` package, currently accessible at https://github.com/EMResearch/EvoMaster/tree/master/e2e-tests/spring-rest-openapi-v2/src/main/java/com/foo/rest/examples/spring/adaptivehypermutation.
[6]https://spring.io/projects/spring-boot.

```
1  @ApiModel
2  public class Info {
3    @ApiModelProperty(required = true)
4    public Integer c;
5    @ApiModelProperty(required = true)
6    @Pattern(regexp = "\\d{4}-\\d{1,2}-\\d{1,2}")
7    public String t;
8    @ApiModelProperty public String d1;
9    @ApiModelProperty public String d2;
10   @ApiModelProperty public String d3;
11 }
```

a. Info DTO class

```
1  @Entity @Table(name = "Foo")
2  public class FooEntity {
3    @Id @NotNull @Min(0)
4    private Integer x;
5    @NotNull private String y;
6    @NotNull private Integer zc;
7    @NotNull private Date zt;
8    private String zd1;
9    private String zd2;
10   private String zd3;
11 }
```

b. Info Entity class

Fig. 5. Info DTO class and Foo Entity class.

and *body payload* z with type Info object. For the z, it is restricted by a reference object with Info which is composed of five fields, and one of them t is constrained with a regular expression (see Info **Data Transfer Object** (**DTO**) in Figure 5(a), where the complete code is in Info.java). In the five fields, only two of them (i.e., c and t) are marked as ‘‘required’’. This means that the other three fields are *optional*, and can be omitted. In our context, the lines and branches in the figure can be regarded as *testing targets* to be optimized with the search. As seen in Figure 4, the targets are affected by the state of the interacted database and different values of the parameters, e.g., lines 5–6 by the state of the database, lines 7–8 by x, lines 9–10 by y, and lines 12–20 by z. With the search, such states and values of the parameters can be manipulated by the search operator, e.g., with mutations. Based on the execution sequence of the targets, there exist a potential order of priority of manipulating the state or the parameters. For instance, in order to enter this method, a valid z is required whose t field needs to follow the specified regular expression (see the use of the annotation @Valid). Otherwise, Spring would return a 400 HTTP status code without even the need to execute that method. After valid parameters are given as input, lines 7–end can be reached only if the else-branch at line 5 is covered. This means that, before that target is covered, it is useless to manipulate the input parameters (i.e., x, y, and z) except for the database state. The branch target is related to a global state of the SUT that requires a set of pre-actions which are to insert at least three foos into the database (e.g., with other POST requests on other endpoints that lead to the creation of such data, or with direct SQL INSERTs from the tests). An implementation for persisting foo to the database is represented in Figure 5(b) (complete code in FooEntity.java), that defines the table foo with 7 columns using a JPA entity. Note that involving the SQL insertions will result in more genes in an individual. Subsequently, the individual also needs to first handle x to get a value for reaching lines 9–end, then handle y to have a string value ‘‘foo’’ for reaching lines 11–end. Moreover, at different stages of the search, the optimal values of the parameters are also varied by the testing targets to be evaluated. For example, as one field of z related to the four branches (i.e., B0–B3) at lines 12–16 (one else branch is hidden), the search only focuses on the branch at line 16 (make z.c 300) once branches at lines 12 (a test which z.c’s value is 100) and 14 (a test which z.c’s value is 200) are covered. Furthermore, values in pre-SQL actions can also affect the testing target in this endpoint. For instance, there exist a branch (at line 21) that is independent of the parameters of the endpoint, and rather relies on the values specified in the SQL insertions (or previous successful POST calls with x equal to 42).

Figure 6 (complete code BarRestAPI.java) is implemented to retrieve (i.e., GET) a bar resource with a specified a by a path bars/{a} that is simple and depends on whether a specific bar exists in the database (see line 4).

```
1 @RequestMapping(value = "/bars/{a}",
2   method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON)
3 public ResponseEntity<Bar> getBarById(@PathVariable(name = "a") Integer a) {
4   if (barRepository.findById(a).isPresent())
5     return ResponseEntity.status(404).build();
6   Bar dto = barRepository.findById(a).get().getDto();
7   return ResponseEntity.ok(dto);
8 }
```

Fig. 6. `bars/{a}` endpoint with GET.

A test for the REST API can include more than one HTTP action for testing multiple endpoints. For instance, Figure 7 shows a test related to two endpoints, i.e., `POST foos/{x}` and `GET bars/{a}`, with RestAssured.[7] In the context of testing REST APIs, SQL commands can be applied for manipulating different states of the SUT [12] as it is needed for covering targets such as the branch target at line 5 shown in Figure 4. For instance, lines 4–14 are used to create a `bar` and four `foo` resources with SQL commands. Note that the complete code for the insertions is not shown due to space limitation, and more details can be found in `test_example()`.[8] Given such state, lines 15–37 are to test `POST foos/{x}` endpoint with expected response (i.e., cover B3, B4, and B5 branches) and lines 25–32 are to test `bars/{a}` endpoint with expected response. In our context, such a test with this test data can be treated as an individual.

From the point of view of the automated test generation, one challenge here is that there are a large number of genes introduced to handle the SQL insertions (due to line 5), but most of them do not have an impact on the control flow of the SUT (apart from line 21). However, those genes still have to be there, e.g., to satisfy the non-null constraints in the database. Without new techniques to handle these cases, the search can be severely hampered, by wasting time in mutating genes that have no impact on the phenotype. Furthermore, any such technique will have to handle these cases *dynamically* and *adaptively*. For example, the genes of `x`, `y`, and `z` have no impact until the `else` branch of line 5 is executed.

To better demonstrate the performance of our novel proposed techniques, we also created an End-to-End test[9] which employs different techniques (i.e., a baseline technique and the proposed technique) against the `foo` endpoints (i.e., GET ALL, GET, DELETE, and POST). For EvoMaster, being a piece of software itself, we need to write tests for it (unit, integration, and End-to-End), e.g., to avoid having our research results negatively affected by software faults. Each time we present a new technique with a motivating example, we add it as an End-To-End test (i.e., as part of the build, we automatically run EvoMaster on the motivating examples, and verify that all testing targets are covered in reasonable time).

Regarding the six branch targets (i.e., B0–B5 in Figure 4) that are the most difficult to solve in this REST API, within three attempts, the proposed approach is capable of producing tests that cover all of the branches by assigning a budget, i.e., $20k$ HTTP calls that take a couple of minutes, while the baseline technique is able to cover just 0 or 1 using the same budget. More details can be found in the link.[9]

[7] https://rest-assured.io/.
[8] The test can be found at https://github.com/EMResearch/EvoMaster/blob/master/e2e-tests/spring-rest-openapi-v2/src/test/java/org/evomaster/e2etests/spring/examples/adaptivehypermutation/ManualRestTest.java.
[9] The E2E test can be currently found at https://github.com/EMResearch/EvoMaster/blob/master/e2e-tests/spring-rest-openapi-v2/src/test/java/org/evomaster/e2etests/spring/examples/adaptivehypermutation/AHypermutationAWHTest.java.

```java
@Test
public void test_example() {
  //add 1 bar and 4 foo
  List<InsertionDto> insertions = sql().insertInto("BAR", 0L)
    .d("A", "0")
    .d("B", "\"bar\"")
    .d("C", "369")
    .and()...and().insertInto("FOO", 42L)
    .d("X", "42")
    .d("Y", "\"foo\"")
    .d("ZC", "0")
    .d("ZT", "\"1900-01-20\"")
    .dtos();
  controller.execInsertionsIntoDatabase(insertions);
  given().accept("*/*")
    .contentType("application/json")
    .body(" { " +
      " \"c\": 300.0, " +
      " \"d1\": \"d1\", " +
      " \"d2\": \"d2\", " +
      " \"d3\": \"d3\", " +
      " \"t\": \"2020-01-01\" " +
    " } ")
    .post(baseUrlOfSut + "/api/foos/22?y=foo")
    .then()
    .assertThat()
    .statusCode(HttpStatus.OK.value())
    .contentType(ContentType.TEXT)
    .body(equalTo("B3B4B5"));
  given().accept("*/*")
    .get(baseUrlOfSut + "/api/bars/0")
    .then()
    .assertThat()
    .statusCode(HttpStatus.OK.value())
    .contentType(ContentType.JSON)
    .body("'b'", equalTo("bar"))
    .body("'c'",equalTo(369));
}
```

Fig. 7. An example of a test manually written.

5 WEIGHT-BASED HYPERMUTATION

5.1 Individual and Gene Structure

In the evolutionary search, a test case will be defined as an *individual*. In a test case, there are several configurations that need to be set, like the choice of query parameters in the URLs, and body payloads (e.g., JSON data). The goal of the search is to find the best configuration to maximize the fitness function (e.g., code coverage and number of detected faults).

A typical representation for an individual in an evolutionary algorithm is a 0/1 bit-string. An individual can then be seen as a list of *n genes*, each one representing one binary choice (i.e., either 0 or 1). Genes can then be mutated (e.g., flipping a bit from 0–1, or vice-versa). The search space will then have a size of 2^n distinct individuals.

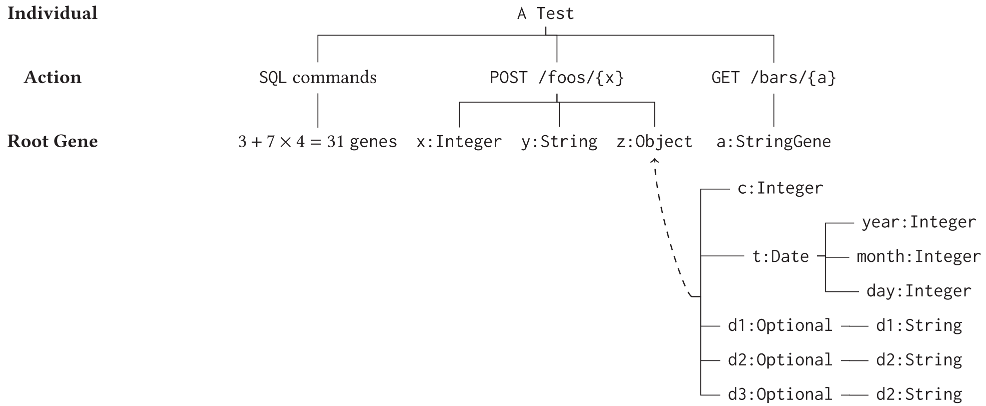


Fig. 8. An example to illustrate a representation of an individual for the test shown in Figure 7.

All search problems can use a bit-string representation, but it has limitations. For example, not all bit-string of a given length n might represent valid individuals (e.g., there can be constraints, like when needing to represent a string-date in a valid format and numbers in a given range), or the length of the representation must be variable (e.g., text strings of variable length). Customized representations (and mutation operations) are usually designed to exploit as much domain knowledge as possible.

When testing a RESTful API, with HTTP calls, and SQL commands, there is a large variation of data types, e.g., numbers, strings, dates, objects, arrays, regular expressions, and primary/foreign keys in SQL tables. EvoMaster uses a rich type system, in which each data type has its own customized gene representation, with specialized mutation operators on each different data type. As the test in Figure 7, its individual representation is illustrated with views of actions and genes in Figure 8. The individual is composed of two HTTP calls, i.e., the first HTTP call on `/foos/{x}` with POST has three genes (i.e., an `Integer` gene for a variable in the path element `x` of the URL, a `String` gene for a query parameter `y`, and a `Object` gene for a possible JSON body payload), and the second HTTP call on `/bars/{a}` with GET has an `String` gene representing a variable in the path element `a` of the URL.

Regarding the genes, to better handle the problem of test data generation, a key difference in EvoMaster compared with traditional evolutionary algorithms is that genes have a tree-like representation. In other words, each gene could have child genes, and so on recursively (e.g., to represents structured data that are objects with fields that are themselves objects). For the *body payload* `z` in Figure 8, it is represented by an `Object` gene with `Info` DTO class. As its implementation in Figure 5(a), the object gene includes five fields, where the field `t` is represented with a `Date` object. In addition, the `Date` object is internally represented with three `Integer` genes (one for the year, one for the month, and one for the day), each with its own constraints (e.g., a valid month can only be within the range of 1 and 12). To mutate such an object gene, its mutation operator will need to decide which of the internal genes (i.e., fields) to mutate (one or more), and then apply the specific mutation operators for the types of these internal genes (e.g., `Date` gene, `String` gene).

Moreover, as the test in Figure 7, there exists a set of SQL insertions for adding one `bar` resource and four `foo` resources that should also be represented along with contained genes. For instance, an insertion for adding one `foo` has seven root genes as Figure 5(b), and one of the genes is a primary key.

Note that when we refer to $|G| = n$, we consider only these top-level/root genes, and not the whole trees. For the example shown in Figure 8, we have $n = 31 + 3 + 1 = 35$ top-level genes.

Table 1. Weights and Number of Direct Child Genes (if any) for Each Type of Genes

GeneType	Weight	#Internal Genes
NumberGene	1	0
BooleanGene	1	0
StringGene	1	$\geq 0^{\dagger}$
DateGene	1	3
TimeGene	1	3
DateTimeGene	1	2
EnumGene	1	≥ 1
MapGene	*sum of weights of elements* + 1	≥ 1
ArrayGene	*sum of weights of elements* + 1	≥ 1
ObjectGene	*sum of weights of its fields*	≥ 1
DisruptiveGene	*weight of gene* $\times p + 1$	0, 1
OptionalGene	*weight of gene* +1	0, 1
AnyCharacterRxGene	1	0
CharacterClassEscapeRxGene	1	0
CharacterRangeRxGene	1	0
DisjunctionRxGene	*sum of weight of terms* + 1	≥ 1
DisjunctionListRxGene	*sum of weight of disjunctions* + 1	≥ 1
QuantifierRxGene	*sum of weight of atoms* + 1	≥ 1
RegexGene	*weight of disjunctionListRxGene*	1

Note that † the *String* gene might contain a set of internal genes when applying testability transformations [13].

5.2 Gene and Mutation Weight

In a traditional mutation, it is a common practice to have an equal mutation rate that is applied to all the genes, e.g., $1/n$. In our context, considering the structured genes, e.g., an object gene containing 50 fields versus a gene for a simple boolean, however, having the same mutation probability would be very unbalanced. To address this issue, it would make sense to define a customized mutation rate for each of the genes that would be dependent on the characteristics of the gene.

To customize the mutation rate, we define a *mutation weight* for each of the mutable genes, denoted as a positive value w that is used in calculating the mutation rate, i.e., the higher the weight, the higher the chances to be mutated. Such a weight for a gene can be decided in many different ways, e.g., based on its complexity and the number of possible configurations. Considering the example in Figure 8, because the `Object` gene `z` contains much more information than the `String` gene `y`, then the weight for `z` should be more than the weight of y. Therefore, we define a *mutation weight* based on each type of mutable genes, and the weights are represented in Table 1 as follows:

- For basic types, i.e., *Number,*[10] *Boolean* and *String*, we set their weights as 1.
- For time related genes, i.e., *Date*, *Time,* and *DateTime*, because their structures are fixed and they only refer to time/date information, we define their weights as 1.
- For *Enum*, the mutation is to select one of defined items in the *Enum* that can be regarded as a scoped *Integer*, thus the weight is defined as 1.
- For *Collection* genes, i.e., *Map* and *Array*, their weights are defined by considering weights of included elements and possible size mutation, i.e., $(\sum_{e=1}^{n_e} w_e) + 1$ where w_e is a weight of an included element and n_e is a number of included elements.

[10] *Number* includes *Integer*, *Long Float,* and *Double* in EvoMaster.

- For *Object*, the mutation is cumulative based on weights of its fields, i.e., $\sum_{f=1}^{n_f} w_f$ where w_f is a weight of a field and n_f is a number of its fields.
- For *Disruptive*,[11] it is a gene containing another gene. The mutation is implemented to mutate the contained gene with a probability $p \in [0, 1]$. The weight is calculated as $w_g \times p + 1$ where w_g is a weight of the contained gene and +1 is for the additional property.
- For *Optional*,[12] it is a gene containing another gene with *present* property defined by a boolean value, i.e., the gene is present (`True`) or absent (`False`). The weight is calculated as $w_g + 1$ where w_g is a weight of the contained gene and +1 is regarding the mutation of the *present* property. For instance, `d1` is not required as defined in Figure 5(a), then `d1` is formalized as an *Optional* gene.
- For regex related genes,[13]
  — *AnyCharacter*, *CharacterClassEscape,* and *CharacterRange* are used to define characters in the regular expression. *AnyCharacter* is to represent any character, *CharacterClassEscape* is defined for representing special characters (e.g., $\backslash d$ and $\backslash s$), and *CharacterRange* is to represent ranges of characters, e.g., [*a-z*]. Those genes can be regarded as scoped chars, thus their weights are defined as 1.
  — *DisjunctionList* defines a (sub)expression connected with the disjunction operator, that is composed of a sequence of disjunctions. The weight is defined based on the internal disjunctions, i.e., $(\sum_{d=1}^{n_d} w_d) + 1$ where w_d is a weight of a disjunction and n_d is the number of these disjunctions.
  — *Disjunction* defines an element in a (sub)expression connected with the disjunction operator. The element is composed of a sequence of terms, e.g., *AnyCharacter.* The weight is defined based on the terms plus whether appending a prefix or postfix (to handle the cases of the delimiters \$ and ^), i.e., $(\sum_{t=1}^{n_t} w_t) + 1$ where w_t is a weight of a term and n_t is the number of all these terms.
  — *Quantifier* is used to represent a repeated pattern (e.g.,$(foo)\{2\}$,$\backslash d+$) that is defined with an atom template (i.e., a term) and the number of repeated times. Since the number of repetitions could be defined as an unbounded range, e.g., + and $*$, the gene could produce a very large number of atoms each one requiring their own genes. Therefore, EvoMaster defined a limit for the gene, and the mutation is implemented to mutate one of repeated genes or add/remove one [12]. The weight is defined based on the atoms plus whether an atom is added or removed, i.e., $(\sum_{a=1}^{n_a} w_a) + 1$ where w_a is a weight of an atom, n_a is a number of all these atoms, +1 is for the possible removal or addition of an atom.
  — *Regex* is used to define a complete regular expression that contains a *DisjunctionList*, then its weight is equal to the weight of the contained *DisjunctionList.*

5.3 Hypermutation with Deterministic Parameter Control

Specialized evolutionary algorithms for test case generation, such as MIO, do not use the crossover operator. Defining meaningful crossover operators for test cases is a challenge in itself. The downside is that relying on a typical *1/n* mutation rate would lead to just 1 mutated gene on average. This would put more emphasis on the exploitation rather than the exploration of the search landscape. This is a problem for system testing, where the search space is huge, and search budget is

[11]In the context of testing RESTful API, *Disruptive* gene is used to handle URL path parameters which should not be mutated once set to handle different calls working on the same resource paths [7].
[12]In the context of testing RESTful API, *Optional* gene is needed to handle optional query parameters [7].
[13]Genes are defined for handling strings in EvoMaster which should match some specified regular expressions [12]. As *PatternCharacterBlock* is not mutable, it is not discussed in this article.

usually small (due to the high cost of running system tests). To offset this issue, algorithms such as MIO have high probabilities of sampling new individuals at random, rather than just mutating existing ones in the evolving populations.

A complementary approach would be to increase the mutation rate, in what it is called *hypermutation*. Instead of doing 1 mutation on average (i.e., mutation rate of $\frac{1}{n}$ for each gene), there could be ℓ mutated genes on average. This would be simply achieved by using a mutation rate of $\frac{\ell}{n}$, where $1 \leq \ell \leq n$. As n would vary from individual to individual, having a constant ℓ for the whole search would not be appropriate. The parameter ℓ should rather be based on a percentage ρ of n, i.e., $\ell = max(1, \rho n)$, with $\rho \in [0, 1]$.

Having hypermutation at the beginning of the search can be useful for exploration of the search landscape. However, at later stages, it can be too disruptive. The MIO algorithm already includes a *dpc* system that, for example, will linearly decrease the probability of sampling new random individuals (instead of mutating existing ones) throughout the search. The same can be done for the parameter ρ, e.g., start at 0.5 and decrease it linearly down to $\frac{1}{n}$. In particular, $\rho = dpc_\rho(s, e)$, where s is the starting value at the beginning of the search (e.g., $\rho = 0.5$), and e is the value when MIO's *focused search* starts (e.g., $\rho = \frac{1}{n}$). Note that there can be other strategies [27–29], besides a simple linear decrease of these parameters. However, whether they would give better results would be a matter of future empirical investigations.

5.4 Mutation Rate

Given the weights and a number of genes to be mutated, we define a weight-based *mutation rate*. Suppose that there exist n ($n \geq 1$) top-level mutable genes $G = \{g_1, \ldots, g_n\}$ in an individual, and each gene g_i has a weight w_i. As previously discussed, as n would vary from individual to individual, having a constant number of genes to be mutated (i.e., ℓ) for the whole search would not be appropriate. The parameter ℓ should rather be based on a percentage ρ of n, i.e., $\ell = max(1, \rho n)$, with $\rho \in [0, 1]$. Thus, with the weights, a mutation rate m_i for the g_i is calculated as

$$m_i(t, W) = \ell \times \left(d \times \frac{1}{n} + (1 - d) \times \frac{w_i}{\sum_{j=1}^{n} w_j} \right), \quad (1)$$

where ℓ is an average number of genes to mutate in one operation ($\ell \leq n$), i.e., with those mutation rates for all n genes, then $\sum_{i=1}^{n} m_i(t) = \ell$ genes on average are selected to mutate; $W = \{w_x | x = 1..n\}$ is a set of weights for G; $d \in [0, 1]$ is a tunable value, e.g., when $d = 1$, the weight w_i contributes nothing to the m_i, and so the mutation rate would be a simple $\frac{\ell}{n}$. Note that hypermutation would be represented by $\ell > 1$; standard mutation $\frac{1}{n}$ would be represented by $\ell = 1$ and then $d = 1$ and/or all $w_i = 1$.

The motivation for the design of Equation (1) is as follows. We want ℓ top-genes mutated on average. However, not all top-genes have the same importance (e.g., a simple boolean versus an object with tens of internal fields), and so the mutation rate should be based on the weights w_i for each top-gene, divided by the sum of all weights $\sum_{j=1}^{n} w_j$ (i.e., proportional to weight). Note that, if all weights w_i are equal, then $\frac{w_i}{\sum_{j=1}^{n} w_j} = \frac{1}{n}$. However, we do not want w_i to lead in some cases to very low probabilities. Especially for REST APIs, we could have low-weight genes (e.g, booleans and numbers) for URL query parameters, and high-weight object fields (e.g., JSON and XML data) for the HTTP body payloads. These latter should be mutated more often, but still, query parameters might have a high chance of impacting the fitness, and so we should avoid decreasing their mutation rate too much if the HTTP body payloads have a lot of internal fields. Therefore, the use of the parameter d is to offset possible side effects of using the weights w_i, by giving each gene at least a minimum mutation rate of d/n. Note the use of the multiplicative $(1 - d)$, as

we need to guarantee that the sum of the probabilities for all top-genes should be equal to ℓ, i.e., $\sum_{i=1}^{n} m_i(t) = \ell$. How to best choose the right value for d is a matter of empirical investigation, and it depends on the problem domain.

Such mutation rate can be treated as a strategy for gene selection that, in our context, is applied on not only the individuals (to select a subset of the n top-level genes) but also on the structured genes which have more than one internal gene. For instance, to mutate an object gene, a selection should be applied for deciding which field is to be mutated. The same applies when a mutated gene has a selection of G' internal genes to mutate (e.g., an array gene that has $|G'|$ child genes, each one representing an element in the array). However, in this case, the value ℓ should be based on the cardinality of $|G'|$ and not $|G|$.

5.5 SQL Gene

An individual might also contain SQL commands, e.g., SQL insertions for manipulating the states of the SUT and the database in Figure 8. The SQL commands might lead to have many new more genes as part of the chromosome in an individual. As in that example, to make the SUT have at least 3 `foo` resources for covering the branch (see line 5 in Figure 4), the test (Figure 7) includes four SQL insertions for `foo` that will lead to $7 \times 4 = 28$ more top-level genes, and in total there are 31 genes from SQL commands (see Figure 8). Compared with the 31 genes, the four genes from the HTTP actions might result in a relative low chance to be mutated.

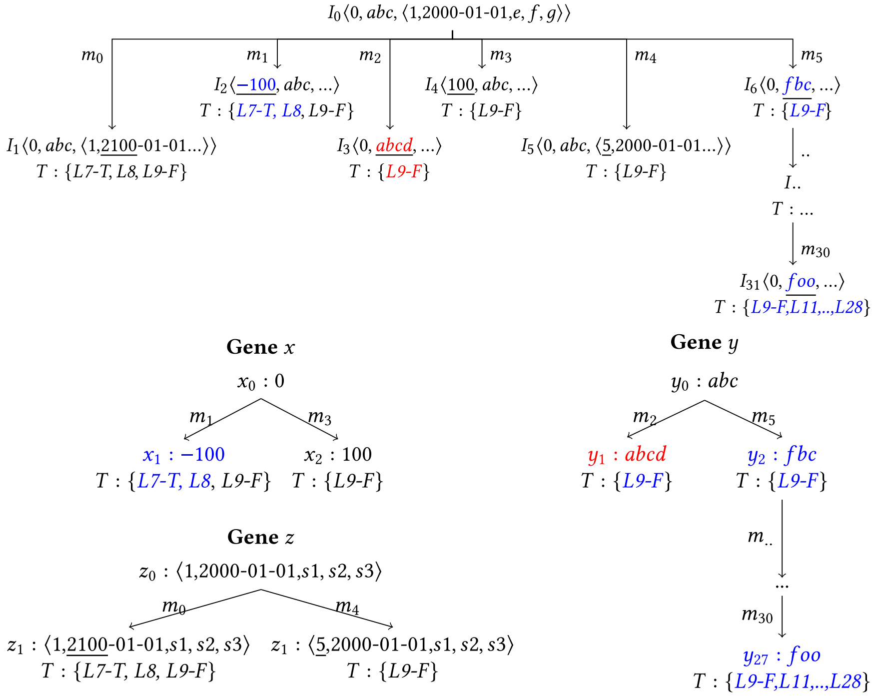
The genes for HTTP actions could have more impacts than the SQL genes on covering more targets. Much of the data for the SQL insertions could have been added only to satisfy constraints in the database (e.g., when the elements in a column must not be null), and have no effect on the control flow in the SUT. Therefore, we design a special handling of SQL genes by distinguishing them from the HTTP action genes when deciding the mutation rate.

Suppose that G is a set of mutable genes in an individual, $Z \subseteq G$ is a subset of G representing a set of non-SQL genes, and hence $G \setminus Z$ represents the genes from SQL actions. Then, when selecting genes to mutate, we apply weight-based selection within the HTTP action gene set and the SQL gene set, respectively, i.e., Z and $G \setminus Z$. To make the two sets have the same overall mutation probability, an average number of genes t to mutate is distributed to the two sets equally. Note that when *focused search* in MIO starts, the t for the two sets is 0.5 to allow only small modifications.

6 ADAPTIVE MUTATION

When mutating an individual in our context, the mutation is applied to try to evolve new individuals that cover the currently reached targets (recall our definition in Section 2.4), e.g., lines and branches. Regarding the implementation shown in Figure 4, each statement is defined as a target labelled as *L<number>*(*-T/F*), e.g., *L7-T* represents the `if-condition` at line 7 where the `condition` is true, and *L9* represents the statement at line 9. Thus, lines *L5–L28* are the testing targets for `POST /foos/{x}`. Note: even if statements are consecutive in the same block, they are treated as separate targets, as statements could throw exceptions (and so the following statements would not be executed in those cases).

Given a set of targets to optimize for, however, not all of the genes in an individual are related to such targets. For instance, the example in Figure 9 shows the evolution of an individual to test the endpoint `POST /foos/{x}` (see its implementation in Figure 4). In the example, the individual to be mutated is composed of the four `foo` SQL insertions (to cover *L5-F*) and an HTTP action `POST /foos/{x}`. For Non-SQL genes, the individual includes three top-level root genes $\{x, y, z\}$ (see genes representation for `POST /foos/{x}` in Figure 8), and z is an `Object` that further includes a set of internal genes, i.e., $\{c, t, d1, d2, d3\}$. In the figure, at the beginning of the



Note that I represents an individual which is composed of an HTTP action, `POST /foos/{x}` (see its implementation in Figure 4 and its genes representation in Figure 8); $I_i\langle x, y, z\rangle$ represents three root genes of the individual I over a course of the search, where i is an index in the evolution (e.g., generation number); $m_j(I, T)$ represents the mutation of I for aiming at targets T, where j is an index of applying mutation on I; value represents the mutated genes; regarding the evaluation of the mutated individual, for the mutated genes and the targets, blue indicates a fitness improvement, red indicates a fitness decrease, and black indicates no change.

Fig. 9. An example to illustrate the evolution of an individual through mutations during the search.

search, Non-SQL genes in the individual I_0 are initialized, e.g., by random sampling, as $I_0\langle 0, abc, \langle 1{,}2000\text{-}01\text{-}01\rangle, e, f, g\rangle$. With the SQL actions, *L7–L10* are reachable, and the I_0 can cover targets {*L7-F, L9-T, L10*}, thus the targets to be optimized are {*L7-T, L8, L9-F*}. Note that, once some targets are covered, in MIO, we do not consider them in the fitness function for the rest of search. Covering one target may also additional targets to be reached, e.g., once *L9-F* is covered, *L11–L28* are reached at m_{30}. Based on that implementation of the SUT, there are two genes (i.e., `x` and `y`) which are related to the execution of {*L7-T, L8, L9-F*}. Thus, applying any mutation on `z` will lead to a waste of budget, e.g., m_0 in Figure 9, as that mutation has no impact on those testing targets. To reduce such waste, one possible solution is to identify possibly related genes for the targets, and then increase the chances to mutate them. Therefore, in this article, we propose an *impact-based weight* that decides gene weights adaptively based on collected impact on targets throughout the search (Sections 6.1 and 6.2). In addition, we developed an *adaptive gene mutation* to mutate the genes based on their fitness impacts and latest mutation history (Section 6.3).

6.1 Gene Impact

Recall the example in Figure 9. Assume that *L9-F* is the target to be optimized (after m_1), mutating `x` (as m_3) and `z` (as m_4) genes does not lead to any impact on the fitness of the target. Regarding

Table 2. Impact Information Collection for Each Type of Genes

GeneType	Additional Impacts	Description
StringGene	*employSpecialization*	F: impact when the specialization is not employed T: impact when the specialization is employed
	specializationImpacts	a map of impacts for each existing specialization in the gene
DateGene	*yearImpact*	impact when mutating the year gene
	monthImpact	impact when mutating the month gene
	dayImpact	impact when mutating the day gene
TimeGene	*hourImpact*	impact when mutating hour gene
	minuteImpact	impact when mutating minute gene
	secondImpact	impact when mutating second gene
DateTimeGene	*dateImpact*	impact when mutating date gene
	timeImpact	impact when mutating time gene
EnumGene	*valuesImpact*	a map of impacts for each value
MapGene, ArrayGene	*lengthImpact*	impact when removing or adding elements
ObjectGene	*fieldsImpact*	a map of impacts for each field
DisruptiveGene	*geneImpact*	impact for the contained gene
OptionalGene	*presentImpact*	F: impact when the present is false T: impact when the present is true
	geneImpact	impact for the contained gene
DisjunctionRxGene	*termsImpact*	a map of impacts for each contained term
DisjunctionListRxGene	*disjunctionsImpact*	a map of impacts for each contained DisjunctionRxGenes
RegexGene	*listRxGeneImpact*	impact of contained DisjunctionListRxGene

m_2 and m_5, mutating the `y` gene shows impacts on the target *L9-F* that can be positive (e.g., m_5) or negative (e.g., m_2). As such, we define *gene impact* and collect the impacts over each gene mutation history, per individual. The *gene impact* is described with

- m_t represents the number of times there was an impact on the given target t, i.e., the fitness of t was changed with the mutation of the gene, and
- n_t represents the number of times there was no impact on the given target t, i.e., the fitness of t is not changed with the mutation of the gene.

In addition, a gene may have internal genes, and the impacts of the internal genes also needs to be collected. Moreover, a gene might also have additional characteristics which are employed during mutation. For instance, regarding the *Optional* gene, there exists an additional property, i.e., *present*, indicating whether the gene is present (i.e., whether this gene should contribute to the phenotype of the individual). During the mutation of the gene, we can mutate either the contained gene (when the `present` is `True`) or the property by flipping the `present` boolean. In this case, it might be helpful to collect an impact regarding the `present` property. Table 2 presents a list of genes which are defined with additional internal impacts. For each type of gene, we have

- *Number*, *AnyCharacterRxGene*, *CharacterClassEscapeRxGene*, *CharacterRangeRxGene*. There is no need to collect additional impacts because they do not have internal genes and additional properties that are used in the mutation.
- *QuantifierRxGene*. It might produce a very large number of atoms. Collecting impacts for such atoms might not be effective to distinguish genes. Therefore, we do not collect internal impacts for it currently.
- *Date*, *Time*, *DateTime*, *Enum*, *Object*, *Object*, *Disruptive*, *DisjunctionRx*, *DisjunctionListRx* and *Regex*. Impacts with respects to their internal genes are collected.
- *String*. A *String* might be specified with regular expressions. For example, in `Info` DTO shown in Figure 5(a), one of its properties named `t` is defined with `\d{4}-\d{1,2}-\d{1,2}`.

ALGORITHM 2: Collect and Update Impacts After Each Mutation (impactCollect)

```
Input  : Targets T, Mutated Genes S, Original form of the Mutated Genes O, Impacts of the Mutated
         Genes M
Output : Updated Impacts M
1 ET ← {}
2 foreach element t of T do
3   if isTargetChanged(t) then
4     ET ← ET ∪ {t}
5 foreach element g of S do
    // Update g impact, for every ET, m_t += 1.0/|S|, for every T \ ET, n_t += 1.0
6   m_g ← updateImpacts(1.0/|S|, ET, 1.0, T \ ET)
    /* if g has internal genes, extract the mutated internal genes S_gc, their orignal
       form O_gc and impacts M_gc.                                                    */
7   O_gc, S_gc, M_gc ← extractInternalMutatedGenes(o_g, g, m_g)
8   if |S_gc| > 0 then
9     m_g ← impactCollect(T, S_gc, O_gc, M_gc)
```

With the support of testability transformations [13], the *String* can be transferred into a *Regex* and *Date* that can be regarded as its specializations. A *String* might have many (≥ 0) specializations. During the mutation, we can change a different specialization, mutate the current specialization, or mutate the value. As such, we collect additional impacts, for *employSpecialization* and *specializationImpacts*.

- *Map* and *Array*. We collect an impact for the *length* that is used to decide about mutating an element or adding/removing an element. Regarding each of the elements, we currently do not collect impacts for them because the collection might have many elements, and impacts among them might be not much different.
- *Optional*. Impacts for the contained gene and `present` property are collected.

In the context of hypermutation, multiple genes (i.e., S and $|S| > 1$) might be mutated at one time. Given a target t, after applying a mutation on S, there exist two conditions: (1) if there is no change on a fitness of t, then n_t for all genes in S should be increased by 1; (2) if there is a change on the fitness, then we treat all mutated genes equally by increasing their m_t with $1/|S|$. During hypermutation, there might be a misleading counting for m_t when *impactful* genes and *noimpactful* genes are mixed together in S. However, the counting for n_t would be correct, and an increase of n_t is greater or equal than an increase of m_t, i.e., $1 \geq 1/|S|$. Therefore, by comparing an *impactful* gene (g_m) with a *noimpactful* gene (g_n) with m_t and n_t, the impactful indicator (m_t) of g_m is never less than g_n, i.e., $m_t^{g_m} \geq m_t^{g_n}$, and the noimpactful indicator (n_t) of g_m is never greater than g_n, i.e., $n_t^{g_m} \leq n_t^{g_n}$. Therefore, it is still possible to distinguish *impactful* genes using this impact collection.

Algorithm 2 defines the impact collection for the mutated genes. Line 6 presents impact counting for each of the gene. Because there might exist internal genes, lines 7–9 deal with the collection of additional impacts for them (see Table 2).

6.2 Adaptive Impact-Based Weight

With the gene impacts discussed in Section 6.1, we define adaptive weights based on them, which are calculated as

$$w_i(T) = 1.0 + \sum_{t}^{T} w_i(t) \times 100, \tag{2}$$

$$w_i(t) = \begin{cases} 1.0 & \text{if } n_t = 0 \text{ and } m_t = 0 \\ \dfrac{c \times m_t}{n_t + c \times m_t} & \text{otherwise} \end{cases}, \tag{3}$$

where T is a set of targets evaluated for the mutation, $w_i(T)$ is an impact-based weight for a gene $g_i \in G = \{g_1, ..., g_n\}$ for all targets T, $w_i(t)$ is an impact-based weight for g_i regarding a target $t \in T$, and c is a configurable parameter. For instance, $c = 1$ means that we treat n_t and m_t equally. In this article, we set c 1.5 to give more emphasis to m_t. Note that, when a gene is never mutated, and so n_t and m_t are 0, then we prioritize mutation of such genes to collect its impact information, and thus we set its weight to 1.

For the example shown in Figure 9, $G = \{x, y, z\}$ at the time point m_5, $T = \{L9\text{-}F\}$ based on the last 5 mutations (i.e., $m_0..m_4$), the weights for G can be calculated as: $w_x(T) = 1.0 + \frac{1.5\times0.0}{2.0+1.5\times0.0} \times 100$, $w_y(T) = 1.0 + \frac{1.5\times1.0}{0.0+1.5\times1.0} \times 100$, and $w_z(T) = 1.0 + \frac{1.5\times0.0}{2.0+1.5\times0.0} \times 100$. Compared with `x` and `z`, `y` has a higher weight, and so it will have more chances to be mutated. The weights can be used in Equation (1) to calculate a mutation rate to select a subset from a set of genes.

6.3 Adaptive Gene Value Mutation

When we need to mutate an individual, we not only need to select which genes to mutate, but also *how* to mutate them. Regarding gene value mutation, even if it is a structured gene, ultimately, there would be the need to modify a value of a *Number*, *String*, or *Boolean*. For instance, to mutate the `Object` gene `z`, m_0 modifies a `year` (`Integer`) of a `Date` gene, which is just one of the fields of `z`.

Based on the archived fitness evaluations, we manage to collect impacts (see Section 6.1) that are used to guide the selection of which gene to mutate. In addition, we can employ the impact-based strategy to mutate genes when the mutation is a decision-making problem, e.g., for *Enum*, selecting a value based on its impact. However, it is unlikely that such a strategy would be viable to use to mutate *Number* and *String* genes, due to the extremely large number of possible values they can have. Inspired by the impact collection, in this article, we provide a basic approach to handle *Number* and *String* mutation by deriving a possibly feasible boundary for each specific target based on its collected mutation history. Note that we do not handle *Boolean* mutation because its mutation is very simple, i.e., just flipping the current value.

6.3.1 Number. The valid range of values of *Number* genes are often specified in the schema with boundary values (i.e., *MIN* and *MAX*), indicating that the value should range from *MIN* to *MAX*. In the context of test case generation, mutations for *Number* type genes have been implemented in EvoMaster as modifying a current value (v_c) with a *delta* [7] (named *standard value mutation*), i.e.,

- For *Integer* and *Long*, modifying the current value with a $\pm 2^i$ delta (δ), where i is decided at random within an adaptive range $[0, MAX_\delta]$, and the MAX_δ is decreased during the search, e.g., from 30 to 10.
- *Double* and *Float*, modifying the current value with a $\pm 2^i \times g$ delta, where g is generated with a Gaussian distribution, and i is handled similarly as for *Integer* and *Long*.

If the modified value (v_m based on v_c) is out of the range, then one of the two boundary values (i.e., either *MIN* or *MAX*) will be selected as the modified value. In this mutation, the modified value is based only on δ and the specified range.

Inspired by the impact collection, the gene mutation history with their achieved fitness might be used to narrow down the range of the numerical gene, i.e., an adaptive boundary (MIN_d and MAX_d) derived with the mutation history. Such a derived boundary can be employed by the

mutation for restricting the value modification (i.e., $\delta \leq max(|MAX_d - v_c|, |MIN_d - v_c|)$ and $v_m \in [MIN_d, MAX_d]$), thus a value within the derived boundary would have a high probability to be the modified value. The strategy to update the boundary with a mutation history is represented in Algorithm 3. Suppose that, an integer gene x ranges from $MIN = 0$ to $MAX = 122$, and its target is to satisfy a condition `IF-condition` (`x == 111`). At the beginning, considering that range, x could have an initial sampled value like $x_0 = 0$. Afterwards, x_0 might be selected by the mutation operator, which produces a modified value, e.g., $x_1 = 64$. For the given target, by comparing the fitness values (in this case, based on branch distance for integer comparisons) achieved by x_0 and x_1, we might derive a boundary for the feasible solution. In this case, $x_1 > x_0$ and x_1 has a better fitness than x_0, i.e., $F_{x_1=64} \succ F_{x_0=0}$, thus a feasible solution could be located at $[\frac{x_0+x_1}{2} = 32, MAX_d = 122]$. The boundary could be employed for the next mutation once x_0 or x_1 is selected, e.g., modify the value from x_1 to $x_2 = 48$. Note that we track an evolution history of an individual along with its fitness, and this history is shared among the evolved individuals. For example, for x, its evolution history is $(\langle x_0, F_{x_0}\rangle, \langle x_1, F_{x_1}\rangle)$, when x_0 is selected for applying the next mutation, the derived boundary is still same as x_1 is selected, i.e., $[32, 122]$. Based on the fitness value (F_{x_2}) of x_2, the boundary could be further narrowed down. In this example, compared with x_1, x_2 has a worse fitness value, i.e., $F_{x_1=64} \succ F_{x_2=48}$, then the boundary could be updated to $[\frac{x_1+x_2}{2} = 56, MAX_d = 122]$.

Regarding the target `IF-condition` `x==111`, it is static and linear, and it can be covered when `x` is 111 (assuming the input x is not modified). However, in our context, the targets may be nonlinear (e.g., $x^2 + 4x + 60 == 120$), dynamic or dependent on other variables (e.g., $x == y$). The derived boundaries are just heuristics to handle the most common cases, and so might be less useful in more complex cases. For instance, regarding $x == y$, during the search, `y` might be modified, leading to a different feasible solution for `x`. Therefore, at some point, the feasible solution for `x` may be out of the current derived boundary. To handle these targets, before narrowing down the range, we examine the applicability of the mutation with the derived boundaries for the gene by checking whether there exist a $x_i \notin [MIN_d, MAX_d]$ (produced by the standard mutation) that has a better fitness. If such x_i exists, the derived boundary would be reset to the defined boundary $[MIN, MAX]$ (see lines 6–8 in Algorithm 3). Thus, the modified value can be outside from the narrowed boundary, and the mutation becomes equivalent with the standard mutation. In our context, there often exist genes which are dependent with others. However, if the dependent genes are not mutated (e.g., a period between the two mutations on y), the derived boundary (e.g., for x) is still effective to be used by the mutation. Therefore, in our implementation, we employ recent mutations (e.g., 10 recent mutations) to derive the boundary for a specific target.

As a further optimization, the boundary can be quickly narrowed down by using an intermediate value of the range as the mutated value (e.g., similarly to a binary search algorithm [64]), and this might be helpful at the beginning of the search. However, the mutation with the intermediate value might lead to a big delta compared with the current value, and that may lead to a side effect in the later stages when the exploitation of the search landscape would be preferable (i.e., focused search). Therefore, we develop a strategy to control such mutation with a probability φ which is decided based on three aspects, i.e., number of mutations on the value, recent mutation results, and exploitation control. Note that the parameter control can be implemented in various ways. In this implementation, we control φ with three levels, i.e., *high*, *medium*, and *low* (*high* indicates a high demand to employ such mutation, i.e., a condition when there is a lack of mutations or recent improvements, and *low* indicates a low demand to employ such mutation, i.e., a condition when there exist enough mutation history or recent improvements). With these three levels, the probability φ is calculated as

$$\varphi = \begin{cases} dpc_{\varphi}(p_h, p_m) & \text{if } u < 3 \text{ or } r > 3 \ (high) \\ dpc_{\varphi}(p_m, p_l) & \text{otherwise if } u \in [3, 5] \text{ or } r > 1 \ (medium) \\ p_l & \text{otherwise } (low) \end{cases}, \tag{4}$$

where u is the number of updates on the value for a target; r is a counter for the gene value mutation keeping track of how often it does not lead to a fitness improvement, i.e., r is set to 0 once there is an improvement; $dpc\ (s, e)$ is a function to decrease the value from s to e linearly throughout the search [6], where s is a value when search starts and e is a value when *focused search* starts; For the experiments in this article, we set $p_h = 0.8$, $p_m = 0.5$, and $p_l = 0.1$. With such parameter control, if there exist few mutations applied on the gene or we experience ineffective mutations multiple times, we allow a higher probability to use the intermediate value of the range. Note that, if the intermediate value strategy is not selected (i.e., $\varphi \leq rand()$), then we use the standard way to modify the value, i.e., with $\pm 2^i$ or $\pm 2^i \times g$ delta.

The motivation for this equation is that the number of the mutations and their results would help to infer the performance of recent mutation operations. We want to apply a high φ in two cases: either a gene has just started to be mutated (e.g., $u < 3$), and so we want to collect info on it (i.e., the use of the chosen boundary and its impact on the fitness), or it has been mutated often but without any recent fitness improvement (i.e., $r > 3$), and so it might be worth trying a different kind of mutation operators (e.g., a larger jump) to possibly escape from local optima. Otherwise, we still want to apply a not too low φ (i.e., p_m) when either not too many mutations (i.e., $u \in [3, 5]$, and so we still want to collect more fitness impact info) have been applied so far, or there was no immediate improvement in its latest mutation (i.e., $r > 1$). In all other cases, e.g., when there has been a recent improvement (i.e., $r = 0$) after at least a few mutations (i.e., $u > 5$, and so no major need to collect further fitness impact info), then we want to avoid having large jumps (and so a low p_l), as those could be too disruptive.

In addition, such mutation would still need to be controlled with a consideration of the current stages of the search. For instance, if *focused search* starts, there might be a need to reduce the probability (i.e., φ) of applying the mutation, to better reward the exploitation of the search landscape. Note that, in this case, $dpc_{\varphi}(p_m, p_l) = p_l$ and $dpc_{\varphi}(p_h, p_m) = p_m$.

The choice of having three levels (i.e., *low*, *medium,* and *high*), and the choices for the u and r ranges, were based on some basic preliminary experiments during the development and design of the techniques presented in this article. However, a proper empirical analysis would be needed to further tune them.

In some aspects, our value mutation approach has some similarities with the **Alternating Variable Method** (**AVM**) [57], which it takes as inspiration. AVM is a local search strategy, where modifications are increased (e.g., double the added delta) each time the fitness improves, and reset to the starting delta (e.g., ± 1) otherwise. Due to the constraint of system testing (i.e., extremely large search landscape, large number of objectives to optimize for, high cost of fitness evaluations, and possibly a non-negligible amount of genes with no effects on the phenotype), local search strategies might not be suitable, which led to the design of our hypermutation with history.

6.3.2 String. Integrated with testability transformations, mutating a *String* needs to be handled with respects to *specialization*s and its *value*. Regarding *specialization*s, we apply collected impacts to guide the mutation. More specifically, *employSpecialization* is used to decide whether to mutate the *specialization* or the *value*. In addition, once the specialization mutation is selected, the mutation can either replace the currently applied *specialization* or mutate the current *specialization*. Moreover, the replacement of *specialization* can employ the impact-based solution to select one

ALGORITHM 3: Update Boundary for a Variable Regarding a Specific Target (boundaryUpdate)

```
Input  : Default boundary (min, max), Derived boundary (dmin, dmax, t, r), Latest value l, Mutated
         value m, Is mutated better? b
Output : Updated derived boundary (dmin, dmax)
1  t ← t + 1
2  if b then
3  |  r ← 0
4  else
5  |  r ← r + 1
6  if b & (m < dmin || m > dmax) then
7  |  dmin ← min
8  |  dmax ← max
9  else
10 |  if (b & m > l) || (!b & m < l) then
11 |  |  dmin ← (m + l)/2.0
12 |  else
13 |  |  dmax ← (m + l)/2.0
```

from existing ones based on *specializationImpacts*. Furthermore, mutating the current *specialization* can be converted to the mutation of the type of *specialization*, e.g., *Date*.

Regarding *value*, a *String* value can be formalized as a sequence of chars, i.e., $v = \{c_i | c_i \in characters \text{ and } i = 0..n\}$ where n is the length of the value and *characters* is a set of available candidates of characters. In addition, the char can be read as an integer based on its binary representation, and so *characters* can be treated as an integer boundary. Thus, mutating the value of *String* can be converted to *Number* mutation (as discussed in Section 6.3.1), i.e., either modify the length n or modify an integer c_i from the sequence. With the history, we update the boundary of *length* and boundaries for each of *character*. For the *value*, its mutation strategy strongly depends on the employed fitness function for *String*. For instance, currently in EvoMaster the Left-Alignment distance [34] is employed as the fitness function as follows:

$$f(v, v^*) = \frac{1.0}{1.0 + |n - n^*| \times 65536 + \sum_i^{min(n,n^*)} |c_i - c_i^*|}, \tag{5}$$

that calculates the distance by comparing the length and chars at same index from the left. Therefore, to mutate the *length*, we remove or append characters from the right. To mutate a char, we employ weight-based selection to choose the index, and the weights are calculated based on the boundary, i.e., fewer candidates, higher weights. Once the index is decided, we treat the char as an integer with the updated boundary. Note that if a char at an index might achieve its optimal value (i.e., an empty boundary), then we will not mutate this char. Pseudo-code of adaptively mutating a *String* gene is represented in Algorithm 8 in the Appendix.

7 ADAPTIVE WEIGH-BASED HYPERMUTATION

In this section, we present our novel *adaptive weigh-based hypermutation*, which integrates all of the above proposed techniques. Our novel mutation can be employed by many-objective search algorithms, such as MIO, for white-box system test generation (e.g., for REST APIs).

Algorithm 4 presents the pseudo-code of mutating an individual (also referred to as a test) with our hypermutation. The mutation mainly addresses three problems, i.e., selecting genes to mutate (lines 1–2), mutating the selected genes (lines 4–6), and evaluating the impact of the mutated genes

ALGORITHM 4: Pseudo-Code of Mutating an Individual with Adaptive Hypermutation

Input : Candidates Genes to Mutate G, Targets T, Evaluated Individual I, SQL special handling SQL, A percentage of candidates to mutate ρ, Enabling of gene weights EGW, Probability of enabling of archive-based solution P_A, Enabling of adaptive gene value mutation EAM

Output : Mutated Gene S^*

```
1 Z ← extractNonSqlGenes(G)
2 S ← selectGene(G, Z, T, I, EGW, P_A > rand(), SQL, ρ)        // refer to Algorithm (5)
3 EA ← P_A > rand()
4 foreach element g of S do
5 |  g* ← mutate(g, T, I, EGW, EGW & EA, EAM & EA)             // refer to Algorithm (7)
6 if P_A > 0 then
7 |  M ← extractImpacts(G, I)
8 |  M* ← impactCollect(T, S*, S, M)                          // refer to Algorithm (2)
```

ALGORITHM 5: Pseudo-Code of Selecting a Subset of Genes S from G to Mutate (*SelectGene*)

Input : Candidate Genes to Mutate G, Non-SQL Genes Z, Targets T, Evaluated Individual I, Enabling of Gene Weight EGW, Enable of impact-based adaptive weights EAW, SQL special handling SQL, A percentage of candidates to mutate ρ

Output : Selected Genes S

```
1  S ← {}
2  if ¬EGW then
3  |  if SQL then
4  |  |  k ← |Z|
5  |  else
6  |  |  k ← |G|
7  |  m ← 1/k
8  |  while |S| = 0 do
9  |  |  foreach element g of G do
10 |  |  |  if m > rand() then
11 |  |  |  |  S ← S ∪ {g}
12 else
13 |  while |S| = 0 do
14 |  |  if SQL & |Z| > 0 & |G| − |Z| > 0 then
15 |  |  |  S ← S ∪ SelectSubsetByWeight(Z, T, I, EAW, 2)
16 |  |  |  S ← S ∪ SelectSubsetByWeight(G \ Z, T, I, EAW, 2)
17 |  |  else
18 |  |  |  S ← S ∪ SelectSubsetByWeight(G, T, I, EAW, 1)
```

with respects to the targets (lines 7–8). In addition, the mutation can be configured with whether to select genes with SQL special handling SQL (Section 5.5), whether to enable weight-based hypermutation to select genes to mutate EGW (Section 5), a probability of employing adaptive mutation P_A, and whether to apply adaptive gene value mutation EAM (Section 6). The adaptive solution is controlled with the P_A parameter, as lines 2–3 ($P_A > rand()$) shows in the algorithm.

Algorithm 5 shows the pseudo-code of selecting genes from the individual to mutate. Lines 2–11 implement the default selection with a standard mutation rate in EvoMaster, i.e., $1/k$. For the default implementation, the mutation rate is calculated only with a number of genes from rest actions if the SQL special handling is applied (as done in [12]). Lines 13–18 implement our novel

ALGORITHM 6: Pseudo-Code of Selecting Genes to Mutate with Weight-Based Solutions (SelectSubSetByWeights)

```
Input :Candidate Genes to Select X, Targets T, Evaluated Individual I or Weights W, Enabling of
       impact-based adaptive weights EAW, Number of Set of Genes N, A percentage of candidates
       to mutate ρ
Output:Selected Genes S
1  S ← {}
2  if |W| = 0 then
3  |  W ← {1, . . . , 1}
4  |  foreach element g of X do
5  |  |  if EAW then
6  |  |  |  W_g ← adaptiveWeight(g, T, I)            // refer to Equation (2)
7  |  |  else
8  |  |  |  W_g ← geneWeight(g)                       // refer to Table (1)
9  ℓ ← 1/N
10 if ρ > 0 then
11 |  s ← max(1, ρ × |X|)
12 |  t ← dpc_t(s, ℓ)
13 foreach element g of X do
14 |  m ← ℓ × (d/|X| + (1 − d) × W_g/ ∑ W)         // refer to Equation (1)
15 |  if m > rand( ) then
16 |  |  S ← S ∪ {g}
```

ALGORITHM 7: Pseudo-Code of Mutating a Selected Gene (mutate)

```
Input :A Gene to Mutate g, Targets T, Evaluated Individual I, Enabling of gene weights EGW,
       Enabling of impact-based adaptive weights EAW,Enabling of adaptive gene mutation EAM
Output:Mutated Gene g*
1  C ← extractInternalGenes(g)
2  if |C| = 0 then
3  |  if EAM then
4  |  |  g* ← adaptiveMutation(g)                    // refer to Table 3
5  |  else
6  |  |  g* ← standardMutation(g)
7  else
8  |  g_s ← { }
9  |  if EWG then
10 |  |  g_s ← g_s ∪ selectSubSetByWeights(C, T, I, EAW, 1, 0)
11 |  else
12 |  |  g_s ← g_s ∪ C
13 |  g_c ← random(g_s)                              // select one of g_s at random
14 |  g*_c ← mutate(g_c, T, I, EGW, EAW, EAM)
```

proposed weight-based selection. If the SQL special handling (Section 5.5) is enabled (line 14), the selection is divided into two independent selections for Non-SQL genes (line 15) and SQL genes (line 16), respectively.

The weight-based selection for the hypermutation is implemented as Algorithm 6. At lines 2–8 in the algorithm, weights can be from W, or calculated based on EAW configuration. For the

Table 3. Proposed Techniques for Each Type of Genes that Includes Applicable Selection Strategies when Mutating a Structured gene, and Value Mutation

GeneType	#Internal Genes	Selection Strategy for Internal Genes	Value Mutation
NumberGene	0	-	Standard, AWM
BooleanGene	0	-	Standard, AWM
StringGene	≥ 0	-	Standard, AWS+AWM
DateGene	3	Random, AWS	-
TimeGene	3	Random, AWS	-
DateTimeGene	2	Random, AWS	-
EnumGene	0	-	Standard, AWS
MapGene	≥ 1	Random, WS, AWS	AWS for length
ArrayGene	≥ 1	Random, WS, AWS	AWS for length
ObjectGene	≥ 1	Random, WS, AWS	-
DisruptiveGene $\times p + 1$	0, 1	-	-
OptionalGene	0, 1	-	AWS for present
AnyCharacterRxGene	0	-	-
CharacterClassEscapeRxGene	0	-	-
CharacterRangeRxGene	0	-	-
DisjunctionRxGene	≥ 1	Random, WS, AWS	-
DisjunctionListRxGene	≥ 1	Random, WS, AWS	-
QuantifierRxGene	≥ 1	Random, WS, AWS	-
RegexGene	1	-	-

Note that *Random* means to select one of the internal genes at random; *WS* is to select an internal with the weights based on their types; *AWS* is to select an internal with the weights based on their impacts; *Standard* is to apply a standard value mutation on the gene; and *AWM* is to apply our adaptive value mutation on the gene. For the value mutation, *AWS* is also applicable for some genes, e.g., *Enum* or whether to mutate a length of *ArrayGene* (i.e., remove/add elements).

calculation, the weights can be decided with either pre-defined weights based on gene types (in Table 1) or adaptive weights based on collected impacts. An average number of genes ℓ to mutate is calculated at lines 9–12. ℓ is determined with a configuration ρ which is a percentage of a number of all mutable genes, and with a *dpc* for controlling exploration/exploitation (Section 5.3). At the beginning of the search, a higher mutation rate (led by higher ℓ) would be helpful for the exploration of the search landscape. However, as discussed in Section 2.4, in order to cover more targets, the algorithm should enable a *focused search*, i.e., focus on exploiting promising areas (i.e., reached targets), which is achieved with a lower random sampling probability and a lower mutation rate (i.e., $\ell = 1$). Lines 13–16 deal with the selection of genes with weight-based mutation rate, calculated as discussed in Section 5.4.

Algorithm 7 is used to mutate the genes selected with Algorithm 5. A gene might consist of a set of internal genes, and so weight-based selection can be applied to determine an internal gene to mutate (at lines 8–13). Note that hypermutation is only applied on root gene selection in terms of an individual, i.e., if a selected root gene has internal genes, we only select one of the internal genes to be mutated. Regarding gene value mutation (lines 2–6), either adaptive or standard mutation can be employed, as controlled by the parameter *EAM*.

In Table 3, we also present the applicable mutation techniques for each type of gene. For *String* value mutation, its adaptive mutation implementation is presented in Algorithm 8. Note that adaptive gene selection and **adaptive gene value mutation (AWM)** are controlled with the configurable parameter P_A. This means that other available techniques can also be selected to mutate the gene.

Note that all of our novel techniques, presented in this article, do increase the computational cost of the search algorithm. Collecting and analyzing the history of the mutations is not free. On one hand, this could be a problem for *unit* testing, where the fitness evaluations could be relatively

Table 4. Descriptive Statistics of the Case Studies with a **Number of Classes** (**#Classes**), **Lines of Codes** (**LOCs**), a **Number of Endpoints** (**#Endpoints**), a Number of Genes Extracted from REST Actions, and Interactions with Database

Name	#Classes	LOCs	#Endpoints	#Rest Genes Root(IG = 0, #O/#$IG \geq 1$)	Database N/Y (#T, #C)
rest-ncs	9	602	6	14 (0, 0/14)	N
rest-scs	13	859	11	26 (0, 0/26)	N
rest-news	10	718	7	10 (1, 2/9)	Y (1, 5)
catwatch	69	5,442	13	34 (2, 1/32)	Y (5, 45)
feature-service	23	2,347	18	33 (0, 4/33)	Y (6, 20)
proxyprint	68	7,534	74	64 (4, 14/60)	Y (15, 92)
scout-api	75	7,479	49	108 (0, 19/108)	Y (14, 70)

Regarding #Rest Genes, we further show the genes based on a number of its contained **internal genes** (**#IG**). For the genes which contain more than one gene, we also report a number of gene which is `Object` or contains `Object`. Regarding Database, a **number of tables** (**#T**) and a **total number of columns** (**#C**) are also reported.

cheap compared with such overhead. On the other hand, in *system* testing most of the computation cost is in the fitness evaluation. For example, when doing an HTTP call to a REST API, not only there is the cost of (de)serializing data sent over a TCP connection, but also for all the computation done in the SUT (e.g., accessing databases). In our context of testing REST APIs, such overheads introduced by our techniques are simply negligible (not even 1% compared with the cost of a fitness evaluation, albeit this depends on the employed hardware and SUT).

8 EMPIRICAL STUDY

In this article, we have carried out an empirical study aimed at answering the following **research questions** (**RQs**):

RQ1: How does SQL special handling affect mutation for testing RESTful web services?
RQ2: How does weight-based hypermutation perform in terms of code coverage?
RQ3: How does adaptive mutation perform in terms of code coverage?
RQ4: How does exploration/exploration control affect the adaptive hypermutation?
RQ5: How much improvement (if any) in coverage and fault detection do our novel techniques achieve compared with existing work?

8.1 Case Studies

To evaluate the proposed approach, we conducted an empirical study with seven RESTful APIs[14] that we have used in our previous work [6, 7, 12, 13, 68]. All of the case studies are Java/Kotlin open-source projects that can be compiled to JVM bytecode.

Table 4 shows descriptive statistics of these case studies, including their number of Java/Kotlin class files (#Class), LOCs, #Endpoints, and whether they interact with a database (Y/N for #Database). In addition, to study the proposed mutation approach for the genes, we also report the number of mutable root genes that exist for the endpoints. Moreover, the number of root genes are categorized into two groups in terms of the number of their internal genes, i.e., none (#IG = 0), and not less than one (#$IG \geq 1$). For #$IG \geq 1$, we also report the **number of object** (#O) genes.

Regarding the case studies, three of them are artificial, i.e., **REST Numerical Case Study** ***(rest-ncs)***, **REST String Case Study** ***(rest-scs)***, and *rest-news*. *rest-ncs* and *rest-scs* are based on code

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

Table 5. Description of the Experiment Settings

	EGW	*SQL*	*d*	ρ	P_A	*EAM*	P_r	*FS*	Count
E1	*F*	$\{F, T\}$	-	-	-	-	0.5	0.5	2
E2	*T*	$\{F, T\}$	{0.2, 0.5, 0.8}	0.0	0.0	*F*	0.5	0.5	6
E3	*T*	E2	{0.2, 0.5, 0.8}	{0.2, 0.5}	0.0	*F*	0.5	0.5	6
E4	*T*	E2	E3	$E2 \vee E3$	{0.5, 0.8}	$\{F, T\}$	0.5	0.5	8
E5	*T*	E2	E3	E4	$E3 \oplus E4$	E4	{0.2, 0.5}	{0.5, 0.8}	4
E6	*F*	E1	-	-	-	-	{0.2, 0.5}	{0.5, 0.8}	4

Table 6. Description of Experiment Tasks with Respects to RQs, along with Corresponding Techniques, Settings, and Employed Case Studies

RQs	Tasks	Techniques	Settings	Case Studies
RQ1	study an effect of SQL special handling for mutation **config. parameter**: *SQL*	Base MIO-WH	E1 E2	5 (Database = Y)
RQ2	study performance of weight-based hypermutation **config. parameter**: d and ρ	MIO-WH	E3	7
RQ3	study performance of adaptive weight-based hypermuation **config. parameter**: P_A and *EAM*	MIO-WH*	E4	
RQ4	study effects of different settings on exploration/exploitation control for mutation **config. parameter**: P_r and *FS*	Base MIO-WH*	E5 E6	
RQ5	compare proposed technique with baseline technique	MIO-WH* vs. Base	Best config.	

The detailed configuration for the settings can be found in Table 5.

examples previously introduced for experiments on solving numerical [9] and string [2] problems in the context of unit testing. *rest-news* was developed for educational purposes on enterprise development in a university course of one of the authors.[15] The remaining four case studies are real RESTful web service projects, i.e., *features-service*, *proxyprint*, *scout-api*, and *catwatch*. These APIs were selected by analyzing projects on the widely used open-source repository GitHub. In previous work, we searched for projects that included OpenAPI schemas, and that were possible to compile, build, and run without problems (e.g., due to missing dependencies, compilation errors, or requiring third-party services/tools that no longer exist). This was a time-consuming manual effort, as GitHub does not provide an easy way to list all projects that are RESTful APIs. Every time we found a new suitable API, we added it to our selection of APIs for experimentation.

8.2 Experiment Design

To evaluate the proposed techniques and answer the five RQs, we conducted an empirical study as shown in Table 5 (for experiment settings) and Table 6 (for experiment design).

In the study, we employed our novel weight-based hypermutation (WH^*) on MIO (named MIO-WH^*), and further compared the MIO-WH^* with default MIO (named *Base*) in terms of three metrics, i.e., a **number of covered targets** (**#Target**), **line coverage** (**%Lines**), and **branch coverage** (**%Branch**). Table 6 presents the experiment tasks with related configurable parameters, experiment settings and case studies for each of the RQs. The configurable parameters are explained in detail as follows:

- *EGW* represents whether to enable the weight-based mutation rate that controls the prerequisite for weight-based mutation;
- *SQL* represents whether to enable SQL special handling when selecting genes to mutate;

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

- $d \in [0, 1]$ is a tunable value (see Equation (1)) that can be used to control a ratio of weights for the mutation rate calculation, e.g., $d = 1$ means that weights are not used to calculate the mutation rate;
- $\rho \in [0, 1]$ is a parameter to configure hypermutation, e.g., $\rho = 0$ implies the mutation of only one gene on average, whereas $\rho = 0.5$ would lead to mutate 50% of genes in an individual on average;
- P_A is the probability to control the application of adaptive mutation;
- *EAM* represents whether to mutate gene value adaptively, and enabled adaptive mutation (i.e., $P_A > 0$) is its prerequisite;
- P_r and *FS* are used to control the tradeoff between exploration and exploitation in MIO [6].

In the context of RESTful web services, we propose SQL special handling that distinguishes SQL genes from all mutable genes when selecting genes to mutate. To study the effects of applying such handling, we conducted an experiment for *SQL* with default MIO and the newly proposed technique (RQ1 in Table 6). In addition, regarding proposed techniques, we defined *WH* and WH^* by whether adaptive strategies are enabled/controlled by P_A. To investigate the performance of different configurations of *WH*, we study the tuning of d regarding mutation rate and ρ regarding hypermutation. Moreover, we performed experiments with different settings of P_A and *EAM* in order to identify the best way to involve adaptive strategies in gene mutation. Furthermore, budgets assigned to exploration and exploitation of the search landscape might have an impact on adaptive mutation since it requires to collect impacts with mutations. Therefore, we performed further experiments with different settings on exploration and exploitation control in MIO (RQ4). The idea is that, with hypermutation, it might be beneficial to decrease the probability of random sampling (exploration), and also delay the start of focus search (exploitation).

Last, but not the least, we compared our new technique with the selected baseline technique MIO (RQ5), which is the current default setting in EvoMaster. We do not compare with other tools, because, as of the current moment, to the best of our knowledge EvoMaster is the only tool that can do white-box testing of REST APIs, and that can handle SQL data insertions directly from the tests.

Regarding the experiment settings, for $EGW = T \wedge SQL \in \{F, T\} \wedge d \in \{0.2, 0.5, 0.8\} \wedge \rho \in \{0, 0.2, 0.5\} \wedge P_A \in \{0, 0.5, 0.8\} \wedge EAM \in \{F, T\} \wedge P_r \in \{0.2, 0.5\} \wedge FS \in \{0.5, 0.8\}$ there would be $1 \times 2 \times 3 \times 3 \times 3 \times 2 \times 2 \times 2 = 432$ possible configurations. In addition, due to the stochastic nature of the search algorithms, each of the configurations should be repeated several times, and 30 is a typically recommended number of repetitions [10], which we used in this study. So, assuming that we use a search budget of 100,000 HTTP calls on each of the seven case studies, it would require running $432 \times 30 \times 100k \times 7 = 9072m$ HTTP calls. However, it would not be viable to conduct such experiments, even considering the cluster of computers we have access to for experimentation. Therefore, to reduce the number of configurations to experiment with, we defined the experiment settings as shown in Table 5. Each of the settings can be employed to study the best configuration, which can then be used in the following experiments. For example, the choice of whether to activate SQL handling (T) or not (F) in $E3$ is based on the setting that gives best results in the previous set of experiments $E2$. Similarly, the choice of d in $E4$ is based on the best one out of the three values experimented in the previous $E3$, and so on, as shown in Table 5. With such design, there exist 30 configurations in total, which is a more manageable number, i.e., repeat each of the configurations 30 times with $100k$ HTTP calls on each of the used case studies (for a total of 630 million HTTP calls).

Due to the very high cost of running experiments on system test generation, there is a limit on the number of configuration settings we could experiment with. The choice of values for the

Table 7. Average #Targets, %Lines, and %Branches and Their Ranks with $SQL \in \{F, T\}$ in E1 Experiment Setting

SUT	#Targets		%Lines		%Branches	
	F	T	F	T	F	T
rest-news	**344.50(1)**	342.14(2)	**53.51%(1)**	52.89%(2)	26.31%(2)	**26.37%(1)**
catwatch	1178.20(2)	**1182.92(1)**	31.55%(2)	**31.67%(1)**	17.24%(2)	**17.54%(1)**
features-service	**486.83(1)**	▼451.07(2)	**42.15%(1)**	38.84%(2)	**13.47%(1)**	11.38%(2)
proxyprint	1545.47(2)	▲**1909.67(1)**	21.73%(2)	**26.39%(1)**	6.92%(2)	**10.59%(1)**
scout-api	1819.57(2)	**1838.83(1)**	38.45%(2)	**38.79%(1)**	20.36%(2)	**20.40%(1)**
Average rank	1.60	**1.40**	1.60	**1.40**	1.80	**1.20**
Friedmantest	χ^2 = 0.2, p-value = 0.655		χ^2 = 0.2, p-value = 0.655		χ^2 = 1.8, p-value = 0.180	

Rank with the value 1 represents the highest achievement, and values in bold are the highest in the case study. In addition, we also report the χ^2 and p-value of the Friedman test for variance analysis by the ranks. Moreover, we compared *SQL* settings (*T* vs. *F*) using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney effect sizes ($\hat{A}_{12}$), and results in details are reported in Table 16. In this table, statistical significant comparison results (p-values < 0.05) are denoted with ▲ and ▼. ▲ indicates that $SQL = T$ is statistically better than $SQL = F$ ($\hat{A}_{12} > 0.5$), and ▼ indicates that $SQL = F$ is statistically better than $SQL = T$ ($\hat{A}_{12} < 0.5$).

different parameters was based on our experience in this kind research, where we tried to have a balance between *low* (i.e., 0.2), *medium* (i.e., 0.5), and *high* (i.e., 0.8) values, to give the readers a high-level overview of how those values may impact the search effectiveness. But other options could have been possible (e.g., 0.1 for representing *low* values). At the end, any given choice is bound to be rather arbitrary. Fine tuning of those parameters might lead to better results in our experiments, but there is always the possible issue of overfitting to the used case study [11].

8.3 Experiment Results

8.3.1 Results for RQ1. To answer RQ1, we compared the performance of two configurations ($SQL = F/T$) of MIO and MIO-WH on five case studies which interact with a database.

Regarding MIO with *E1* setting, Table 7 reports effectiveness of the two configurations with #Targets, %Lines, and %Branches, and comparisons between the two configurations are shown in Table 7. Based on the results, for the artificial case study *rest-news*, the configuration $SQL = F$ is slightly better than $SQL = T$ in terms of #Targets and %Lines, but not for %Branches. For the other SUTs, in *catwatch* and *scout-api*, we observed better performance on $SQL = T$ than $SQL = F$, i.e., positive *relative* improvement, but the difference is not significant $p > 0.05$. In *proxyprint*, there exists a significantly improvement with $SQL = T$ compared with $SQL = F$ (i.e., high $\hat{A}_{xy} = 0.96$, high *relative* = +23.57% and low $p < 0.001$), but the results for *feature-service* are the opposite.

Regarding MIO-WH with *E2* setting, we conducted the same analysis as for *MIO*, and results are shown in Tables 8 and 17. With these results, we found that, except for *catwatch*, the configuration $SQL = T$ is overall better than $SQL = F$ in the other four case studies, especially for *proxyprint* case study.

The main idea for proposing *special SQL handling* was to avoid a low mutation rate for HTTP genes when the number of SQL genes is large. Based on the obtained results, with $SQL = T$ configuration we found that both *Base* and *MIO-WH* achieved a strong improvement on *proxyprint*. As can be seen in Table 4, *proxyprint* has the most endpoints to be tested that interact with a database, which has the most tables and columns. Thus, for *proxyprint*, there might exist a relative high probability of including a large number of SQL genes in a test. This might be a reason for the strong improvement with the enabled *special SQL handling*. However, with *Base*, the SQL handling shows its limitation in *feature-service*. In *Base*, the handling is implemented by only

Table 8. Average #Targets, %Lines, and %Branches and Their Ranks with $d \in \{0.2, 0.5, 0.8\}$ and $SQL \in \{F, T\}$ in E2 Experiment Setting

SUT	*d*	#Targets		%Lines		%Branches	
		F	T	F	T	F	T
rest-news	0.2	341.60(5)	342.40(3)	53.52%(4)	53.61%(3)	26.02%(5)	26.14%(4)
	0.5	343.70(2)	342.04(4)	53.74%(2)	53.49%(5)	26.22%(3)	26.32%(2)
	0.8	340.18(6)	**344.07(1)**	53.34%(6)	**53.75%(1)**	26.00%(6)	**26.42%(1)**
catwatch	0.2	**1181.29(1)**	1160.44(6)	31.50%(2)	31.04%(6)	17.37%(2)	17.00%(5)
	0.5	1180.21(2)	▼1164.00(4)	**31.55%(1)**	31.08%(5)	**17.39%(1)**	16.98%(6)
	0.8	1177.57(3)	1163.03(5)	31.47%(3)	31.12%(4)	17.34%(3)	17.06%(4)
features-service	0.2	478.18(5)	**498.66(1)**	41.40%(5)	**43.09%(1)**	13.11%(5)	**13.91%(1)**
	0.5	477.93(6)	480.96(4)	41.33%(6)	41.52%(4)	13.01%(6)	13.23%(4)
	0.8	487.27(2)	486.00(3)	42.10%(2)	42.03%(3)	13.31%(3)	13.59%(2)
proxyprint	0.2	1626.20(5)	▲1874.79(2)	22.76%(4)	25.97%(2)	7.63%(5)	10.52%(2)
	0.5	1629.83(4)	▲**1885.96(1)**	22.76%(5)	**26.13%(1)**	7.89%(4)	**10.53%(1)**
	0.8	1544.40(6)	▲1825.61(3)	21.65%(6)	25.29%(3)	6.75%(6)	9.61%(3)
scout-api	0.2	1800.90(6)	1811.50(5)	37.96%(6)	38.12%(5)	20.30%(6)	20.47%(3)
	0.5	1823.30(2)	1819.13(3)	38.36%(2)	38.29%(3)	20.33%(5)	20.54%(2)
	0.8	1815.03(4)	**1833.37(1)**	38.18%(4)	**38.67%(1)**	20.41%(4)	**20.77%(1)**
Average rank	0.2	3.86	3.14	4.00	3.43	4.29	3.14
	0.5	**3.00**	**3.00**	**3.00**	3.29	3.14	**2.57**
	0.8	4.57	3.43	4.29	**3.00**	4.71	3.14
Friedmantest		χ^2 = 3.91, p-value = 0.562		χ^2 = 2.91, p-value = 0.714		χ^2 = 6.84, p-value = 0.233	

Rank with the value 1 represents the highest achievement, and values in bold are the highest in the case study. In addition, we also report the χ^2 and p-value of the Friedman test for variance analysis by the ranks. Moreover, we compared *SQL* settings (*T* vs. *F*) using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney effect sizes ($\hat{A}_{12}$), and results in details are reported in Table 17. In this table, statistical significant comparison results (p-values < 0.05) are denoted with ▲ and ▼. ▲ indicates that $SQL = T$ is statistically better than $SQL = F$ ($\hat{A}_{12} > 0.5$), and ▼ indicates that $SQL = F$ is statistically better than $SQL = T$ ($\hat{A}_{12} < 0.5$).

employing HTTP genes to calculate a mutation rate that results in a higher exploration and limits the performance of focused search. Thus, such handling might underperform on the case study which requires more exploitation such as *feature-service*, which has the second lowest number of endpoints, second lowest number of SQL tables and columns. Besides, since *MIO-WH* controls the mutation rate when *focused search* starts (recall Section 5.5), the limitation on *feature-service* is not shown in *MIO-WH*. Regarding *MIO-WH*, the special SQL handling shows a significant limitation on *catwatch* with $d = 0.5$, but the relative decrease in performance of target coverage is small, i.e., −1.37% (see Table 17 in Appendix). As shown in Table 4, *catwatch* has the lowest number of tables and second lowest number of columns, which might be a possible reason for this small underperformance.

Based on these overall results, we can conclude that:

RQ1: Our SQL special handling is effective overall in improving mutation for testing RESTful web services.

8.3.2 Results for RQ2. RQ2 is used to study the performance of weight-based hypermutation (*MIO-WH*), which is related to the tunable d and ρ. Table 9 presents the overall performance in terms of average #Targets, %Lines, and %Branches, with their ranks. Given that table, we can see that test cases produced by *MIO-WH* are capable of achieving up to 87.80% line coverage and

Table 9. With $SQL = T$, Average #Targets, %Lines, and %Branches and Their Ranks with $d \in \{0.2, 0.5, 0.8\}$ and $\rho \in \{0.2, 0.5\}$ in E3 Experiment Setting

SUT	*d*	#Targets		%Lines		%Branches	
		$\rho = 0.2$	$\rho = 0.5$	$\rho = 0.2$	$\rho = 0.5$	$\rho = 0.2$	$\rho = 0.5$
rest-ncs	0.2	619.73(4)	**622.57(1)**	87.40%(4)	87.70%(2)	65.23%(5)	▲**65.72%(1)**
	0.5	619.73(4)	622.50(2)	87.39%(6)	87.69%(3)	65.23%(5)	▲65.70%(2)
	0.8	619.70(6)	622.50(2)	87.40%(4)	**87.73%(1)**	65.23%(5)	▲65.69%(3)
rest-scs	0.2	790.87(4)	▲828.17(2)	76.02%(4)	▲79.11%(2)	46.00%(4)	▲48.10%(2)
	0.5	780.27(6)	▲826.73(3)	75.25%(6)	▲78.96%(3)	45.08%(6)	▲47.77%(3)
	0.8	781.30(5)	▲**830.80(1)**	75.34%(5)	▲**79.40%(1)**	45.19%(5)	▲**48.36%(1)**
rest-news	0.2	**345.10(1)**	▼334.63(6)	**53.86%(1)**	▼51.37%(6)	26.50%(2)	25.92%(6)
	0.5	338.15(4)	335.37(5)	51.67%(4)	51.46%(5)	**26.63%(1)**	26.02%(5)
	0.8	344.10(2)	339.17(3)	53.75%(2)	53.01%(3)	26.37%(3)	26.05%(4)
catwatch	0.2	1,193.71(2)	▼1,161.07(6)	31.08%(2)	▼30.30%(6)	16.93%(3)	16.93%(4)
	0.5	1,167.75(3)	1,163.57(4)	30.40%(3)	30.37%(4)	16.79%(6)	16.98%(2)
	0.8	**1,260.50(1)**	▼1,162.06(5)	**32.64%(1)**	▼30.36%(5)	**17.37%(1)**	▼16.89%(5)
features-service	0.2	497.52(4)	499.95(3)	43.07%(4)	43.20%(3)	14.11%(4)	14.45%(3)
	0.5	484.77(6)	503.82(2)	41.85%(6)	▲43.62%(2)	13.39%(6)	▲14.75%(2)
	0.8	**505.83(1)**	490.68(5)	**43.79%(1)**	42.32%(5)	**14.89%(1)**	13.94%(5)
proxyprint	0.2	1,849.20(6)	1,854.15(5)	25.61%(6)	25.66%(5)	9.94%(4)	9.78%(6)
	0.5	1,860.50(4)	1,882.65(2)	25.75%(4)	26.06%(3)	9.92%(5)	10.12%(3)
	0.8	1,882.33(3)	▲**1,951.37(1)**	26.06%(2)	▲**26.95%(1)**	10.45%(2)	▲**11.29%(1)**
scout-api	0.2	1,794.13(6)	1,810.35(5)	37.81%(6)	38.11%(5)	20.45%(3)	20.42%(4)
	0.5	1,815.77(4)	1,828.70(2)	38.26%(4)	38.44%(2)	20.29%(6)	20.59%(2)
	0.8	1,824.30(3)	▲**1,858.17(1)**	38.38%(3)	**39.02%(1)**	20.32%(5)	**20.65%(1)**
Average rank	0.2	3.93	4.00	3.93	4.14	3.57	3.71
	0.5	4.50	2.93	4.71	3.14	5.00	**2.71**
	0.8	3.00	**2.64**	2.64	**2.43**	3.14	2.86
Friedmantest		$\chi^2 = 5.53$, p-value $= 0.354$		$\chi^2 = 8.2$, p-value $= 0.146$		$\chi^2 = 7.03$, p-value $= 0.218$	

In each of the case studies, Rank with the value 1 represents the highest achievement which is also in bold. In addition, we also report the χ^2 and p-value of the Friedman test for variance analysis by the ranks. Moreover, we compared ρ settings (0.5 vs. 0.2) using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney effect sizes ($\hat{A}_{12}$), and results in details are reported in Table 19. In this table, statistical significant comparison results (p-values < 0.05) are denoted with ▲ and ▼. ▲ indicates that $\rho = 0.5$ is statistically better than $\rho = 0.2$ ($\hat{A}_{12} > 0.5$), and ▼ indicates that $\rho = 0.2$ is statistically better than $\rho = 0.5$ ($\hat{A}_{12} < 0.5$).

65.87% branch coverage for the artificial SUTs, and up to 43.79% line coverage and 20.57% branch coverage for other SUTs.

Regarding the 6 configurations (3 × 2), based on results of Friedman test, there does not exist a significant best configuration on all case studies. We also applied pair comparison for d and ρ using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney effect sizes ($\hat{A}_{12}$). The detailed comparison results for d and ρ are reported in Tables 18 and 19, respectively. Regarding the tunable d, based on the results in Table 18, $d = 0.8$ is either statistically better than or equal to $d = 0.2$ and $d = 0.5$. Thus, we can choose 0.8 for d. Regarding the parameter ρ for the hypermutation, based on the results shown in Tables 9 and 19, we can see that an higher hypermutation setting ($\rho = 0.5$) is more effective than lower one ($\rho = 0.2$) on *rest-ncs*, *rest-scs*, *feature-service*, *proxyprint*, and *scout-api*, while the lower hypermutation setting ($\rho = 0.2$) is more effective on *rest-news* and *catwatch*. Therefore, we select 0.5 for ρ, because $\rho = 0.5$ is better on more SUTs than $\rho = 0.2$. In addition, to

Table 10. Pair Comparison on whether to Enable Hypermutation with #Target, %Lines, and %Branches using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{12}$)

SUT	$T(\rho=0.5)$ $F(\rho=0.0)$	#Targets $\hat{A}_{12}$	#Targets *p*-value	#Targets relative	%Lines $\hat{A}_{12}$	%Lines *p*-value	%Lines relative	%Branches $\hat{A}_{12}$	%Branches *p*-value	%Branches relative
rest-ncs	T vs. F	0.56	0.421	+0.12%	0.43	0.321	+0.05%	0.58	0.314	+0.40%
rest-scs	T vs. F	**0.99**	**<0.001**	**+14.64%**	**1.00**	**<0.001**	**+12.68%**	**0.95**	**<0.001**	**+15.52%**
rest-news	T vs. F	0.38	0.097	−1.42%	0.38	0.087	−1.37%	0.43	0.330	−1.40%
catwatch	T vs. F	0.64	0.069	−0.08%	0.64	0.054	+0.01%	0.61	0.156	+1.03%
features-service	T vs. F	0.56	0.396	+0.96%	0.52	0.726	+0.69%	0.53	0.707	+2.60%
proxyprint	T vs. F	**0.76**	**<0.001**	**+6.89%**	**0.76**	**<0.001**	**+6.56%**	**0.72**	**0.003**	**+17.48%**
scout-api	T vs. F	**0.67**	**0.016**	**+1.35%**	**0.65**	**0.038**	**+0.91%**	0.54	0.564	−0.57%

T indicates that the hypermutation is enabled with $\rho = 0.5$, while F indicates that the hypermutation is not enabled with $\rho = 0.0$. Values in bold indicate that the setting with T is significantly better than the setting with F (i.e., p-value < 0.05 and $\hat{A}_{12} > 0.5$).

further investigate the effectiveness of the hypermutation, we compare the selected hypermutation setting $\rho = 0.5$ with the disabled hypermutation setting $\rho = 0.0$. The results are reported in Table 10. With the results for all of the SUTs, the mutation with the enabled hypermutation shows a statistically better or equivalent performance in all coverage metrics.

Thus, we can conclude that:

> *RQ2: MIO-WH is capable of automatically generating tests that cover up to 43.79% of lines in real REST APIs and 87.80% of lines in the artificial REST APIs. Our recommended configuration for weight-based mutation is with $d = 0.8$ and $\rho = 0.5$.*

8.3.3 Results for RQ3. RQ3 is used to investigate the performance of the proposed adaptive mutation, which can be configured with P_A and *EAM*. Table 11 reports the overall performance in terms of average #Targets, %Lines, and %Branches, with their ranks. Considering that table, test cases produced by MIO-WH are capable of achieving up to 87.72% line coverage and 66.64% branch coverage for the artificial SUTs, and up to 42.88% line coverage and 21.41% branch coverage for other SUTs.

Regarding different configurations, in terms of %Branches, there exist a best configuration $P_A = 0.5$ and $EAM = T$ based on significant p-value (i.e., < 0.05) with Friedman test and the best average rank over all case studies (i.e., 1.57). In addition, we report pair comparisons for the related parameters, i.e., P_A and *EAM* in Tables 20 and 21. Regarding a probability of enabling adaptive strategies P_A, with results in Table 20, we select $P_A = 0.5$ because $P_A = 0.5$ is equal or better in performance than $P_A = 0.8$. Regarding *EAM* used to control whether to enable adaptive gene mutation (recall Section 6.3), with the results in Table 21, we can observe that adaptive gene mutation achieved a significant better coverage in four out of the seven SUTs, i.e., *rest-ncs*, *catwatch*, *proxyprint,* and *scout-api*. In *rest-scs* and *feature-services*, significant differences do not exist between the two settings. In *rest-news*, adaptive strategies shows a decrease in performance in target coverage and line coverage with $P_A = 0.8$. However, $P_A = 08$ is not our selected configuration for P_A. In addition, to study the effectiveness of adaptive handling, we compared MIO-WH and MIO-WH* based on the test cases produced by their best configurations for the coverage metrics. Comparison results are shown in Table 12. Based on those results, MIO-WH* shows a clear improvement on *rest-ncs*, *catwatch*, *proxyprint,* and *scout-api*, based on low p-values and high effect sizes in the coverage metrics. For the remaining SUTs, there does not exist significant difference in their performance. Therefore, we can conclude that:

Table 11. Average #Targets, %Lines, and %Branches and Their Ranks with $P_A \in \{0.5, 0.8\}$ and $EAM \in \{F, T\}$ in E4 Experiment Setting

SUT	P_A	#Targets F	#Targets T	%Lines F	%Lines T	%Branches F	%Branches T
rest-ncs	0.5	621.83(4)	▲**627.55(1)**	87.65%(4)	87.70%(2)	65.55%(4)	▲**66.64%(1)**
	0.8	622.29(3)	▲627.27(2)	**87.72%(1)**	87.69%(3)	65.59%(3)	▲66.54%(2)
rest-scs	0.5	824.47(4)	828.21(3)	78.83%(4)	79.19%(2)	47.32%(4)	47.67%(2)
	0.8	**829.53(1)**	829.17(2)	79.10%(3)	**79.79%(1)**	**48.12%(1)**	47.45%(3)
rest-news	0.5	337.73(3)	341.66(2)	53.19%(2)	53.11%(3)	25.56%(4)	26.48%(2)
	0.8	**346.31(1)**	▼333.15(4)	**53.80%(1)**	▼52.25%(4)	**26.66%(1)**	25.86%(3)
catwatch	0.5	1,184.00(3)	▲**1,217.03(1)**	30.91%(3)	▲**31.73%(1)**	17.08%(4)	▲**17.44%(1)**
	0.8	1,182.86(4)	1,203.72(2)	30.87%(4)	31.34%(2)	17.21%(3)	17.41%(2)
features-service	0.5	495.11(3)	495.60(2)	42.78%(3)	42.81%(2)	14.05%(2)	**14.21%(1)**
	0.8	**497.74(1)**	484.83(4)	**42.88%(1)**	41.79%(4)	14.04%(3)	13.56%(4)
proxyprint	0.5	1,949.65(3)	▲2,079.35(2)	26.88%(3)	▲28.56%(2)	11.17%(3)	▲12.16%(2)
	0.8	1,914.53(4)	▲**2,089.82(1)**	26.45%(4)	▲**28.67%(1)**	10.48%(4)	▲**12.17%(1)**
scout-api	0.5	1,854.54(4)	▲**1,949.51(1)**	39.00%(4)	▲**40.68%(1)**	20.53%(4)	▲21.28%(2)
	0.8	1,864.76(3)	▲1,940.95(2)	39.23%(3)	▲40.04%(2)	20.85%(3)	▲**21.41%(1)**
Average rank	0.5	3.43	**1.71**	3.29	**1.86**	3.57	**1.57**
	0.8	2.43	2.43	2.43	2.43	2.57	2.29
Friedmantest		χ^2 = 6.26, p-value = 0.100		χ^2 = 4.37, p-value = 0.224		χ^2 = 8.66, $\boldsymbol{p}$**-value = 0.034**	

In the case study, Rank with the value 1 represents the highest achievement with different configuration which is also in bold. In addition, we also report the χ^2 and p-value of the Friedman test for variance analysis by the ranks. Moreover, we compared *EAM* settings (*T* vs. *F*) using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney effect sizes ($\hat{A}_{12}$), and results in details are reported in Table 21. In this table, statistical significant comparison results (p-values < 0.05) are denoted with ▲ and ▼. ▲ indicates that $EAM = T$ is statistically better than $EAM = F$ ($\hat{A}_{12} > 0.5$), and ▼ indicates that $EAM = F$ is statistically better than $EAM = T$ ($\hat{A}_{12} < 0.5$).

Table 12. Pair Comparisons on whether to Employ Adaptive Handling with #Target, %Lines, and %Branches using Mann–Whitney–Wilcoxon U-Tests (p-$value$) and Vargha–Delaney Effect Sizes ($\hat{A}_{12}$)

SUT		#Targets $\hat{A}_{12}$	#Targets p-value	#Targets relative	%Lines $\hat{A}_{12}$	%Lines p-value	%Lines relative	%Branches $\hat{A}_{12}$	%Branches p-value	%Branches relative
rest-ncs	MIO-WH* vs. MIO-WH	**0.85**	**<0.001**	**+0.81%**	0.54	0.432	−0.03%	**0.78**	**<0.001**	**+1.44%**
rest-scs	MIO-WH* vs. MIO-WH	0.46	0.481	−0.31%	0.46	0.490	−0.27%	0.42	0.198	−1.42%
rest-news	MIO-WH* vs. MIO-WH	0.55	0.332	+0.73%	0.53	0.589	+0.19%	0.59	0.125	+1.65%
catwatch	MIO-WH* vs. MIO-WH	**0.81**	**<0.001**	**+4.73%**	**0.81**	**<0.001**	**+4.54%**	**0.74**	**<0.001**	**+3.30%**
features-service	MIO-WH* vs. MIO-WH	0.52	0.676	+1.00%	0.54	0.420	+1.17%	0.53	0.538	+1.90%
proxyprint	MIO-WH* vs. MIO-WH	**0.91**	**<0.001**	**+6.56%**	**0.91**	**<0.001**	**+5.99%**	**0.87**	**<0.001**	**+7.73%**
scout-api	MIO-WH* vs. MIO-WH	**0.92**	**<0.001**	**+4.92%**	**0.90**	**<0.001**	**+4.25%**	**0.74**	**<0.001**	**+3.06%**

MIO-WH* indicates that the adaptive handling is employed with its recommended configuration, while MIO-WH indicates that the adaptive handling is not employed with its recommended configuration (see RQ2). Values in bold indicate that MIO-WH* is significantly better than MIO-WH (i.e., p-value < 0.05 and $\hat{A}_{12} > 0.5$).

> *RQ3: MIO-WH* is capable of automatically generating tests that cover up to 42.88% of lines in real-world REST APIs and 87.72% of lines in artificial REST APIs. Our recommended configuration for weight-based mutation with adaptive handling (i.e., MIO-WH*) is with $P_A = 0.5$ and $EAM = T$. Besides, adaptive handling helps the mutation operator to achieve a better performance in coverage.*

8.3.4 *Results for RQ4.* RQ4 is used to study the effects of exploration/exploitation control on the mutation operator, with the Base and MIO-WH* configurations. With the two techniques,

Table 13. Applying Base Technique, Average #Targets, %Lines, and %Branches and Their Ranks with $P_{rand} \in \{0.2, 0.5\}$ and $FS \in \{0.5, 0.8\}$ in E5 Experiment Setting

SUT	FS	#Targets 0.2	#Targets 0.5	%Lines 0.2	%Lines 0.5	%Branches 0.2	%Branches 0.5
rest-ncs	0.5	623.14(2)	**623.38(1)**	**87.79%(1)**	87.74%(2)	65.55%(2)	**65.73%(1)**
	0.8	622.03(4)	622.17(3)	87.68%(4)	87.69%(3)	65.45%(4)	65.55%(3)
rest-scs	0.5	735.37(4)	**751.28(1)**	71.15%(4)	**72.40%(1)**	42.37%(4)	**44.39%(1)**
	0.8	745.33(2)	744.43(3)	72.24%(2)	71.85%(3)	42.94%(3)	43.67%(2)
rest-news	0.5	**343.44(1)**	341.83(2)	**53.72%(1)**	53.04%(2)	26.16%(3)	26.21%(2)
	0.8	335.29(4)	340.75(3)	51.47%(4)	52.39%(3)	26.06%(4)	**26.63%(1)**
catwatch	0.5	1,151.80(4)	**1,181.76(1)**	30.02%(4)	**30.86%(1)**	16.68%(4)	**17.19%(1)**
	0.8	1,159.27(3)	1,172.13(2)	30.25%(3)	30.61%(2)	16.87%(3)	16.95%(2)
features-service	0.5	464.00(2)	456.15(4)	40.02%(2)	39.28%(4)	12.01%(2)	11.61%(4)
	0.8	461.83(3)	**464.80(1)**	39.92%(3)	**40.14%(1)**	11.98%(3)	**12.12%(1)**
proxyprint	0.5	**1,953.70(1)**	1,898.58(2)	**27.01%(1)**	26.24%(2)	**11.21%(1)**	10.35%(3)
	0.8	1,882.86(4)	1,895.36(3)	26.12%(4)	26.15%(3)	10.37%(2)	10.26%(4)
scout-api	0.5	1,789.67(3)	1,839.95(2)	37.86%(3)	38.75%(2)	19.52%(4)	20.52%(2)
	0.8	1,785.87(4)	**1,853.50(1)**	37.63%(4)	**38.90%(1)**	19.73%(3)	**20.58%(1)**
Average rank	0.5	2.43	**1.86**	2.29	**2.00**	2.86	**2.00**
	0.8	3.43	2.29	3.43	2.29	3.14	**2.00**
Friedmantest		χ^2 = 5.57, p-value = 0.134		χ^2 = 5.06, p-value = 0.168		χ^2 = 4.37, p-value = 0.224	

Rank 1 represents the highest achieved coverage, and values in bold are the highest in the case study.

exploration/exploitation is controlled by the probability of applying random sampling P_r and the percentage of budget used to start a focused search *FS*.

Regarding Base, Table 13 shows the average #Target, %Lines, and %Branches produced by all settings ($P_r \in \{0.2, 0.5\}$ combined with $FS \in \{0.5, 0.8\}$) for each of the SUTs. The results show that the best configuration is $P_r = 0.5$ with $FS = 0.5$ based on attainment of the best ranks in all coverage metrics, but the variance among the case studies is not significant with the Friedman test. In addition, as seen in the pair comparison results for P_r and *FS*, we found that, (1) for P_r, compared with $P_r = 0.2$, $P_r = 0.5$ shows an equal or better performance in all coverage metrics (see Table 22), and (2) for F_s, compared with $FS = 0.8$, $FS = 0.5$ shows an equal or better performance in all coverage metrics. Therefore, for Base, a balanced exploration/exploitation setting (i.e., $P_r = 0.5$ and $FS = 0.5$) obtains consistently equal or better performance in all coverage metrics for all SUTs.

Regarding MIO-WH*, Table 14 shows the average #Target, %Lines, and %Branches produced by all settings ($P_r \in \{0.2, 0.5\}$ combined with $FS \in \{0.5, 0.8\}$) for each of the case studies. Based on the average ranks, $P_r = 0.5$ with $FS = 0.5$ is the best for all of the three coverage metrics, but the variance among the case studies is not significant based on Friedman test. In addition, to study P_r and *FS* on the case studies, we further report the pair comparison results among their different configurations in Tables 25 and 24. Based on these comparison results, we found that only for *catwatch*, the decreased probability of random sampling ($P_r = 0.2$) achieved a better performance. Regarding *FS*, with $P_r = 0.5$, the delayed focused search setting $FS = 0.8$ shows a clear limited performance on *feature-service* and *proxyprint* case studies, and there does exist any positive upside. Therefore, based on the overall results, we would recommend that:

> *RQ4: With Base and MIO-WH*, a balanced exploration/exploitation setting ($P_r = 0.5$ and $FS = 0.5$) suits most of case studies for maximizing the coverage metrics.*

Table 14. Applying MIO-WH* Technique, Average #Targets, %Lines, and %Branches and Their Ranks with $P_r \in \{0.2, 0.5\}$ and $FS \in \{0.5, 0.8\}$ in E6 Experiment Setting

SUT	FS	#Targets 0.2	#Targets 0.5	%Lines 0.2	%Lines 0.5	%Branches 0.2	%Branches 0.5
rest-ncs	0.5	626.57(4)	627.42(3)	87.62%(4)	87.67%(3)	66.46%(4)	66.64%(3)
	0.8	**628.46(1)**	627.90(2)	**87.86%(1)**	87.68%(2)	66.67%(2)	**66.77%(1)**
rest-scs	0.5	830.76(2)	**834.86(1)**	79.39%(2)	**79.45%(1)**	47.74%(3)	**48.59%(1)**
	0.8	826.17(3)	826.17(4)	79.16%(3)	78.96%(4)	47.30%(4)	47.74%(2)
catwatch	0.5	**1,247.76(1)**	1,197.67(3)	**32.55%(1)**	31.30%(3)	17.51%(2)	17.38%(3)
	0.8	1,231.83(2)	1,194.79(4)	32.11%(2)	31.06%(4)	**17.63%(1)**	17.26%(4)
features-service	0.5	**508.03(1)**	505.60(2)	**43.84%(1)**	43.77%(2)	14.79%(2)	**15.08%(1)**
	0.8	480.62(4)	488.07(3)	41.48%(4)	42.15%(3)	13.38%(4)	13.56%(3)
proxyprint	0.5	2,071.67(2)	**2,117.30(1)**	28.47%(2)	**29.02%(1)**	12.20%(2)	**12.62%(1)**
	0.8	2,054.53(4)	2,056.20(3)	28.25%(4)	28.30%(3)	11.78%(4)	11.82%(3)
scout-api	0.5	1,924.66(4)	**1,961.32(1)**	40.33%(4)	**40.86%(1)**	21.06%(4)	**21.38%(1)**
	0.8	1,948.17(3)	1,951.27(2)	40.74%(2)	40.62%(3)	21.31%(2)	21.27%(3)
Average rank	0.5	2.33	**1.83**	2.33	**1.83**	2.83	**1.67**
	0.8	2.83	3.00	2.67	3.17	2.83	2.67
Friedmantest		$\chi^2 = 3$, p-value = 0.392		$\chi^2 = 3.4$, p-value = 0.334		$\chi^2 = 3.4$, p-value = 0.334	

Rank 1 represents the highest achieved coverage, and values in bold are the highest in the case study.

8.3.5 Results of RQ5. Table 15 shows the results of comparing our best configuration *MIO-WH** with the baseline technique (*Base* = default MIO) in terms of three coverage metrics. Based on the results on average coverage and ranks, compared with *Base*, except *rest-ncs* in %Lines, *MIO-WH** achieves the best rank for the rest of case studies in the coverage metrics. In addition, the significant variance by Friedman test demonstrates the effectiveness of the proposed techniques in target coverage and branch coverage. In Table 15, we also report the results with Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney effect sizes ($\hat{A}_{xy}$). As it can be seen in that table, except *rest-news*, MIO-WH* shows a clear and strong improvement in six out of the seven case studies, based on high effect sizes, high positive relative improvements, and low *p*-values. For *rest-news*, the difference between the two techniques is not significant.

For automated testing approaches, fault detection is another important metric to evaluate. In our context, faults can be identified based on the HTTP status codes (i.e., 500) and unexpected responses. In Table 15, we report a further metric, i.e., the number of "potential" faults detected by Base and MIO-WH* (note that, in HTTP, not all 500 responses are due to software faults). Results demonstrate that MIO-WH* achieved a significant improvement in all of the four non-artificial REST APIs with p-values < 0.01 and $\hat{A}_{12} > 0.8$. For the three artificial REST APIs, MIO-WH* was better only on *rest-scs*. For the *rest-ncs* and *rest-news*, there appear to be some downsides, but the differences were not statistically significant.

To provide more details on the performance of the techniques, Figure 10 plots the average number of covered targets over time during the search, for all the seven case studies. Compared with *Base*, *MIO-WH** has a clear large margin, except for *catwatch* in the first 5% of the budget, and for *rest-news* in the last 5% of the budget. Besides, with MIO-WH*, the covered targets before focused search starts (i.e., the first 50% of the budget) grow faster for all of the case studies. This can be used to demonstrate the effectiveness of our novel hypermutation to solve our addressed problem. Moreover, during the focused search, on *catwatch* and *feature-service*, MIO-WH* maintains a steady growth of the covering targets.

Table 15. Results of Comparing MIO-WH* with the Selected Baseline Technique (Base = default MIO) in Three Coverage Metrics (i.e., #Target, %Lines, and %Branches) and Fault detection

SUT	Coverage	Techniques		A=MIO-WH* vs. B=Base		
		Base	MIO-WH*	$\hat{A}_{ab}$	p-value	relative$_{a-b/b}$
rest-ncs	#Target	623.4(2)	**627.4(1)**	**0.81**	**<0.001**	**+0.65%**
	%Lines	**87.74%(1)**	87.67%(2)	0.52	0.786	−0.07%
	%Branches	65.73%(2)	**66.64%(1)**	**0.77**	**<0.001**	**+1.39%**
	#Faults	**5.1(1)**	5.0(2)	0.46	0.154	−1.71%
rest-scs	#Target	751.3(2)	**834.9(1)**	**0.95**	**<0.001**	**+11.13%**
	%Lines	72.40%(2)	**79.45%(1)**	**0.97**	**<0.001**	**+9.73%**
	%Branches	44.39%(2)	**48.59%(1)**	**0.88**	**0.002**	**+9.45%**
	#Faults	9.3(2)	**10.6(1)**	**0.79**	**0.016**	**+13.97%**
rest-news	#Target	342.1(2)	**342.9(1)**	0.50	1.000	+0.21%
	%Lines	52.89%(2)	**53.40%(1)**	0.61	0.193	+0.96%
	%Branches	26.37%(2)	**26.54%(1)**	0.54	0.673	+0.67%
	#Faults	**7.1(1)**	6.8(2)	0.40	0.138	−5.41%
catwatch	#Target	1,182.9(2)	**1197.7(1)**	**0.68**	**0.022**	**+1.25%**
	%Lines	31.67%(2)	**31.30%(1)**	**0.67**	**0.027**	**+1.33%**
	%Branches	17.54%(2)	**17.38%(1)**	0.61	0.154	+1.12%
	#Faults	19.8(2)	**21.8(1)**	**0.93**	**<0.001**	**+10.08%**
feature-service	#Target	451.1(2)	**505.6(1)**	**0.86**	**<0.001**	**+12.09%**
	%Lines	38.84%(2)	**43.77%(1)**	**0.86**	**<0.001**	**+12.69%**
	%Branches	11.38%(2)	**15.08%(1)**	**0.85**	**<0.001**	**+32.51%**
	#Faults	33.6(2)	**34.4(1)**	**0.80**	**<0.001**	**+2.28%**
proxyprint	#Target	1,909.7(2)	**2117.3(1)**	**0.98**	**<0.001**	**+10.87%**
	%Lines	26.39%(2)	**29.02%(1)**	**0.98**	**<0.001**	**+9.94%**
	%Branches	10.59%(2)	**12.62%(1)**	**0.96**	**<0.001**	**+19.24%**
	#Faults	102.4(2)	**109.8(1)**	**0.88**	**<0.001**	**+7.19%**
scout-api	#Target	1,838.8(2)	**1961.3(1)**	**0.96**	**<0.001**	**+6.66%**
	%Lines	38.79%(2)	**40.96%(1)**	**0.94**	**<0.001**	**+5.57%**
	%Branches	20.40%(2)	**21.38%(1)**	**0.85**	**<0.001**	**+4.78%**
	#Faults	102.9(2)	**113.8(1)**	**0.91**	**<0.001**	**+10.54%**
Average Rank	#Target	2.00	**1.00**	χ^2 = 7, ***p*-value = 0.008**		
	%Lines	1.86	**1.14**	χ^2 = 3.57, p-value = 0.059		
	%Branches	2.00	**1.00**	χ^2 = 7, ***p*-value = 0.008**		
	#Faults	1.71	**1.29**	χ^2 = 1.29, p-value = 0.257		

The results are reported with average coverage metrics, average number of detected fault, and their rank. Rank 1 represents the highest achieved coverage, and values in bold are the highest in the case study. We also represent comparison results, i.e., if better than baseline (i.e., $\hat{A}_{12}$ > 0.5 and p-values < 0.05), and the χ^2 and p-value of the Friedman test.

Thus, we can conclude that:

> *RQ5: Our proposed technique MIO-WH* (i.e., weight-based hypermutation with adaptive handling), with our best configuration settings, significantly outperformed the selected baseline technique in target coverage, line coverage, branch coverage, and fault detection.*

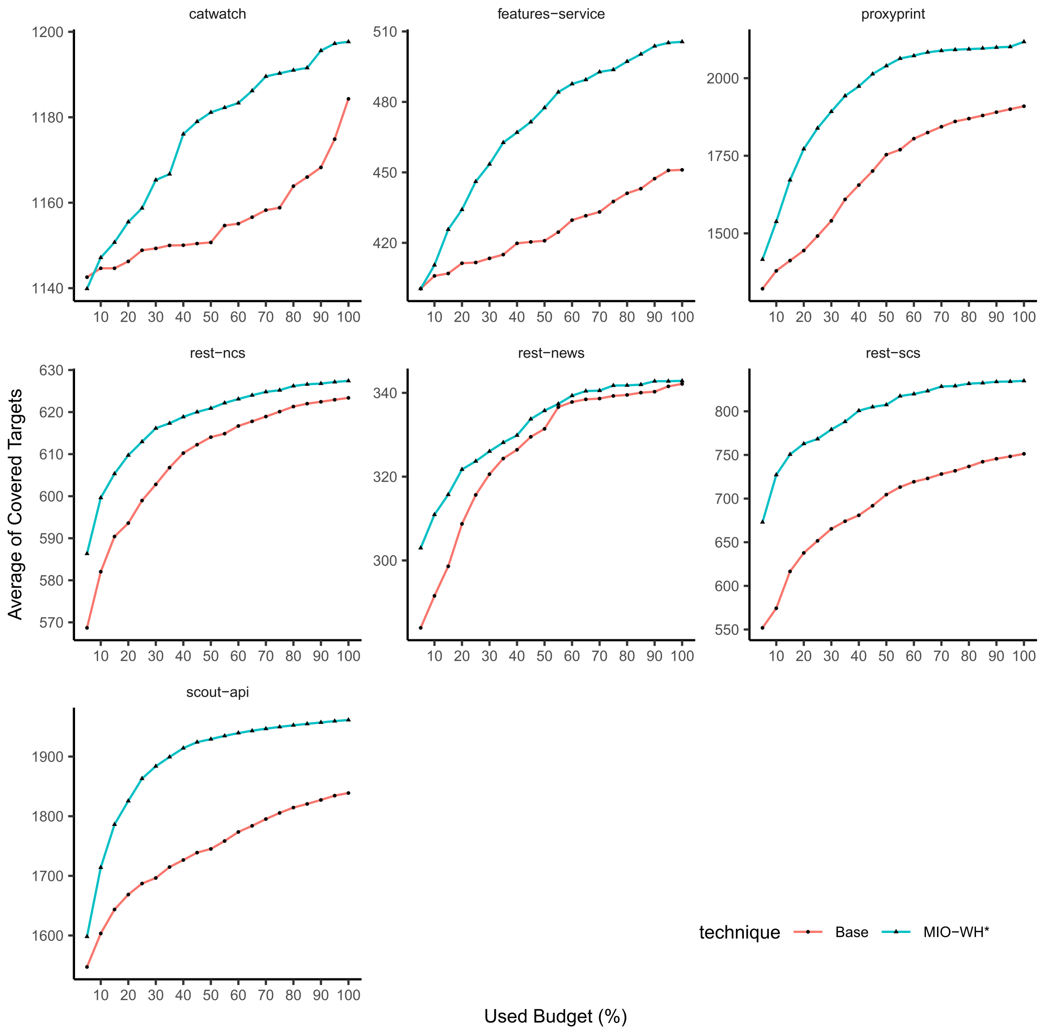


Fig. 10. Average covered targets (*y-axis*) with Base and MIO-WH* throughout the search, reported at 5% intervals of the used budget allocated for the search (*x-axis*).

8.4 Results Discussion

The experiments presented in this article show with high confidence that our novel techniques do improve performance significantly. These good results can be explained if our initial hypothesis is indeed true, i.e., if in this problem domain a non-negligible amount of genes in the evolved individuals have little to no impact on their phenotype. Our adaptive hypermutation seems effective at identifying those types of gene, mutate them less often, while concentrating on the important genes.

For practitioners, besides higher code coverage and higher fault detection, there is no direct impact. These novel techniques are now the default in EvoMaster, and not something that needs to be activated manually. In other words, this kind of algorithmic improvements should be something transparent to users of test case generation tools.

Our novel techniques were evaluated in the context of system test generation, specifically for RESTful APIs. However, they could be used for other kinds of web services as well, like for example SOAP, GraphQL, and gRPC. But they could be used as well in other system test generation contexts, like for web and mobile applications. For example, when having to fill an HTML form as part of an interaction with a web page, there could be many fields that need to be filled before the form can be submitted to the server. But many of these fields might not impact much the control flow in the SUT. Furthermore, when testing web applications, might still need to generate SQL data directly in their databases (if any). However, without a sound empirical investigations, we cannot claim for sure that our novel techniques would be still effective in these other problem domains.

In future work, the scalability of our approach will need to be further investigated, with a larger selection of SUTs. For example, with 92 columns in 15 different SQL tables, *proxyprint* is the SUT with the most SQL data in our empirical study (recall Table 4). But how would our techniques fare on SUTs with even more SQL tables? Without empirical investigations, it is hard to say for sure. However, RESTful APIs, especially when used in microservice applications (which is a common practice in industry), tend to be small [59]. It is a common practice nowadays to avoid large "monolith" applications, and rather split them in separated smaller components (e.g., RESTful APIs) that communicate with each other (i.e., a microservice architecture). In this context, it is a common practice that each REST API will have its own databases, which are not directly accessible from the other APIs in the microservice. Therefore, when testing services in isolation, such scalability would likely be not a major concern. But, if one had to test a whole microservice (which could be composed of hundreds of RESTful APIs [59]) then the scalability of test case generation techniques would be of paramount importance.

9 THREATS TO VALIDITY

Conclusion validity. This threat is related to results obtained by our experiments for answering the RQs. The experiments are in the context of search-based software engineering. By following the guidelines to conduct experiments in such context [10], first, we repeated the experiments for each configuration at least 30 times. This is done to prevent the drawn conclusions being negatively affected by chance, as the search algorithms are stochastic in nature. Furthermore, we illustrated the results by reporting average values, ranks, and line plots. Moreover, to properly analyze the data and draw sound conclusions, we use statistical methods, i.e., the Friedman test for analyzing variance of case studies by technique, and the Mann–Whitney U-test at significance level $\alpha = 0.05$ for analyzing differences pairwise, with effect size computed with the Vargha–Delaney $\hat{A}_{12}$.

Construct validity. The threat is related to the generalization of the outputs of the employed techniques. In our context, the outputs can be regarded as test coverage and fault findings achieved by the generated tests. To prevent bias in such outputs by the search algorithms, we set the same stopping criterion for all the algorithms and configurations, as suggested in the literature (e.g., [1]). Besides, in our context, executing HTTP calls are required for evaluating tests during the search, and such executions can be quite expensive (compared with the computation cost of the algorithms themselves). Thus, time spent by the execution may vary by operating environment, e.g., hardware and OS, while the number of HTTP calls is independent of the execution environment. To make the outputs replicable, we employed a fixed number of HTTP calls (i.e., $100k$) to define the stopping criterion for the search.

Internal validity. This threat is related to our implementation. First, most of our proposed approaches are illustrated in this article with pseudo-code, which makes them assessable. Besides, all of our code implementation and case studies are open-source and freely available online, which allows anyone to review them and replicate the experiments.

External validity. The threat is related to the generalization of our results to other case studies. In this work, we conducted our empirical study with seven SUTs that include three artificial REST APIs and four real-world REST APIs from Github. More REST APIs would help to better generalize our results. However, experiments on system testing are computational expensive, which limit the number of SUTs that can be used in a viable amount of time for experimentation. Furthermore, although REST APIs are widely popular in industry, they are less common in open-source repositories, which makes finding new suitable SUTs for experimentation harder.

10 CONCLUSION

REST web services are widely used in industry. However, testing such services is challenging. In this article, we proposed an adaptive weight-based hypermutation to enhance the mutation of evolutionary algorithms for generating system-level white-box tests for RESTful web services. In the context of testing REST APIs, considering the large amount of genes and their different types, we designed *weight-based hypermutation* to determine which genes to mutate, based on their characteristics and impacts on the evaluated testing targets. Besides, we developed an *adaptive* gene value strategy to mutate the selected genes with consideration of the collected impacts on their internal genes and mutation history.

Our approach takes advantage of EvoMaster which is capable of analyzing coverage at runtime, provides different evolutionary algorithms for test generation and is integrated with advanced techniques for testing, e.g., SQL handling and testability transformations. To assess our techniques, we implemented our novel mutation operator (named MIO-WH*) in the MIO algorithm designed for system test generation, whose effectiveness for REST APIs has been demonstrated in our recent work. Then, we empirically compared MIO-WH* with the default version of MIO in EvoMaster, on seven open-source RESTful APIs. Results showed that MIO-WH* achieved a significant improvement in coverage on six out of the seven case studies. Relative improvements are up to +12.09% for target coverage, +12.69% for line coverage, and +32.51% for branch coverage.

In the future, to study the generalization of our approach, we plan to conduct additional experiments with more evolutionary algorithms on more case studies. In addition, to further investigate our approach with different budgets, we plan to conduct experiments with more budget settings and study the performance with different budgets on different case studies. Our novel techniques were evaluated in the context of test generation for REST APIs, but they might be effective as well in other system-level testing scenarios (e.g., search-based test generation for mobile and web frontend applications).

To enable replicated studies and future tool comparisons, EvoMaster and the employed benchmark are freely available online as open-source. For more details, visit our webpage at `www.evomaster.org`.

APPENDIX

This appendix contains additional algorithms and several tables which show results in detail for the empirical analyses carried out in this article.

ALGORITHM 8: Pseudo-Code of Mutating a String (adaptiveStringMutation)

```
Input  : Targets T, String gene to mutate g, Impacts of the specializations M, Impacts of
         whether to employ a specialization E, Evaluated Individual I, Updated boundary for
         the length ul, Update boundaries for chars UC, Other genes as raw strings O
Output : Mutated Gene g*
1  S ← getSpecializations(G)
2  e ← getEmployedSpecialization(G)
   /* select True or False based on their impacts with Algorithm 6, and getF
      and getT are to get impacts for False and True respectively              */
3  w ← random(selectSubSetByWeights({False, True}, T, {getF(E), getT(E)}, True, 1, 0))
4  if |S| > 0 & w then
5  |  if e < 0 then
6  |  |  e ← random(0, |S| − 1)// select one at random
7  |  else
   |  |  /* select a specialization from S with weights Algorithm 6              */
8  |  |  g_s ← random(selectSubSetByWeights(S, T, M, True, 1, 0))
9  |  |  if g_s == S[e] then
10 |  |  |  g*_s ← mutate(g_s, T, I, True, True, True)
11 |  |  else
12 |  |  |  e ← indexOf(S, g_s)
13 else
14 |  g_v ← getValue(g)
15 |  if e ≥ 0 then
16 |  |  e ← −1
17 |  else
18 |  |  r ← random()
19 |  |  if 0.02 > r & |O| > 0 then
20 |  |  |  g*_v ← random(O)
21 |  |  else
   |  |  |  /* a probability to mutate the length with dpc where there is a
   |  |  |     relative high probablity to mutate a length at the begining of
   |  |  |     the search                                                      */
22 |  |  |  p ← dpc(0.6, 0.2)
23 |  |  |  MC ← extractMutableChars(g_v)
24 |  |  |  if isOptimal(L) || (r < (1.0 − p) & |g_v| > 0 & |MC| > 0) then
   |  |  |  |  // weights are defined based on dmax − dmin
25 |  |  |  |  W_MC ← extractWeights(MC, UC)
26 |  |  |  |  i ← random(selectSubSetByWeights(MC, W_MC, True, 1, 0))
27 |  |  |  |  g_v[i]* ← getChar(sample(UC_i))
28 |  |  |  else
29 |  |  |  |  l ← getInt(sample(ul))
30 |  |  |  |  if l < |g_v| || |g_v| == maxLength || l == |g_v| & r < 1.0 − p/2.0 then
31 |  |  |  |  |  g*_v ← dropLast(g_v, 1)
32 |  |  |  |  else
33 |  |  |  |  |  g*_v ← append(g_v, randomChar())
```

Table 16. **RQ1**: Pair Comparison for $SQL \in \{F, T\}$ Setting with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

projects	A (SQL = F), B (SQL = T)								
	#Targets			%Lines			%Branches		
	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-news	0.45	0.537	−0.69%	0.37	0.127	−1.16%	0.51	0.938	+0.23%
catwatch	0.55	0.535	+0.40%	0.55	0.536	+0.36%	0.59	0.301	+1.72%
features-service	0.25	0.001	−7.35%	0.24	0.002	−7.85%	0.26	0.003	−15.51%
proxyprint	**0.96**	**<0.001**	**+23.57%**	**0.96**	**<0.001**	**+21.48%**	**0.95**	**<0.001**	**+53.00%**
scout-api	0.61	0.131	+1.06%	0.61	0.162	+0.91%	0.57	0.546	+0.19%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 17. **RQ1**: For Each Setting of d, Pair Comparison for $SQL \in \{F, T\}$ Settings with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

projects	d	A (SQL = F), B (SQL = T)								
		#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-news	0.2	0.52	0.383	+0.23%	0.52	0.738	+0.16%	0.52	0.577	+0.46%
	0.5	0.49	0.893	−0.49%	0.49	0.936	−0.47%	0.51	0.933	+0.35%
	0.8	0.58	0.353	+1.14%	0.55	0.392	+0.76%	0.58	0.210	+1.63%
catwatch	0.2	0.35	0.052	−1.76%	0.36	0.063	−1.46%	0.32	0.020	−2.09%
	0.5	0.34	0.039	−1.37%	0.34	0.037	−1.50%	0.33	0.028	−2.36%
	0.8	0.36	0.078	−1.23%	0.34	0.036	−1.11%	0.37	0.089	−1.60%
features-service	0.2	0.63	0.093	+4.28%	0.59	0.227	+4.08%	0.60	0.177	+6.14%
	0.5	0.52	0.682	+0.64%	0.51	0.948	+0.45%	0.52	0.744	+1.63%
	0.8	0.52	0.975	−0.26%	0.51	0.977	−0.19%	0.53	0.615	+2.12%
proxyprint	0.2	**0.89**	**<0.001**	**+15.29%**	**0.88**	**<0.001**	**+14.09%**	**0.87**	**<0.001**	**+37.78%**
	0.5	**0.91**	**<0.001**	**+15.72%**	**0.91**	**<0.001**	**+14.78%**	**0.86**	**<0.001**	**+33.33%**
	0.8	**0.92**	**<0.001**	**+18.21%**	**0.92**	**<0.001**	**+16.81%**	**0.92**	**<0.001**	**+42.39%**
scout-api	0.2	0.60	0.376	+0.59%	0.58	0.371	+0.42%	0.57	0.242	+0.84%
	0.5	0.46	0.537	−0.23%	0.47	0.581	−0.18%	0.49	0.927	+1.01%
	0.8	0.56	0.299	+1.01%	0.57	0.150	+1.29%	0.56	0.144	+1.75%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 18. **RQ2**: With $SQL = T$, Pair Comparison for d Settings for Each Setting of ρ with #Targets using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

projects	ρ	A (d = 0.2), B (d = 0.5), C (d = 0.8)								
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ca}$	p-value$_{ca}$	relative$_{c-a/a}$	$\hat{A}_{cb}$	p-value$_{cb}$	relative$_{c-b/b}$
#Targets										
rest-ncs	0.2	0.50	1.000	+0.00%	0.50	1.000	−0.01%	0.50	1.000	−0.01%
	0.5	0.49	0.572	−0.01%	0.49	0.786	−0.01%	0.49	0.943	+0.00%
rest-scs	0.2	0.37	0.199	−1.34%	0.38	0.116	−1.21%	0.49	0.813	+0.13%
	0.5	0.49	0.902	−0.17%	0.53	0.629	+0.32%	0.55	0.276	+0.49%
rest-news	0.2	0.30	0.010	−2.02%	0.51	0.849	−0.29%	**0.66**	**0.037**	**+1.76%**
	0.5	0.51	0.741	+0.22%	0.64	0.174	+1.36%	0.59	0.263	+1.13%
catwatch	0.2	0.30	0.038	−2.17%	**0.87**	**<0.001**	**+5.60%**	**0.96**	**<0.001**	**+7.94%**
	0.5	0.51	0.829	+0.22%	0.56	0.456	+0.09%	0.54	0.562	−0.13%
features-service	0.2	0.44	0.430	−2.56%	0.60	0.189	+1.67%	**0.63**	**0.034**	**+4.35%**
	0.5	0.51	0.729	+0.78%	0.45	0.398	−1.86%	0.44	0.130	−2.61%
proxyprint	0.2	0.51	0.861	+0.61%	0.57	0.360	+1.79%	0.55	0.477	+1.17%
	0.5	0.58	0.241	+1.54%	**0.72**	**<0.001**	**+5.24%**	**0.64**	**0.038**	**+3.65%**
scout-api	0.2	0.59	0.156	+1.21%	0.65	0.116	+1.68%	0.57	0.365	+0.47%
	0.5	0.59	0.209	+1.01%	**0.76**	**<0.001**	**+2.64%**	0.65	0.121	+1.61%
%Lines										
rest-ncs	0.2	0.49	1.000	−0.01%	0.50	NaN	+0.00%	0.51	1.000	+0.01%
	0.5	0.49	1.000	−0.01%	0.52	0.346	+0.03%	0.53	0.149	+0.04%
rest-scs	0.2	0.40	0.165	−1.02%	0.40	0.147	−0.90%	0.50	0.991	+0.12%
	0.5	0.47	0.688	−0.20%	0.55	0.544	+0.36%	0.58	0.091	+0.56%
rest-news	0.2	0.10	<0.001	−4.07%	0.47	0.720	−0.21%	**0.88**	**<0.001**	**+4.02%**
	0.5	0.52	0.761	+0.18%	**0.79**	**<0.001**	**+3.20%**	**0.77**	**<0.001**	**+3.02%**
catwatch	0.2	0.32	0.027	−2.17%	**0.87**	**<0.001**	**+5.04%**	**0.96**	**<0.001**	**+7.37%**
	0.5	0.53	0.676	+0.20%	0.61	0.156	+0.17%	0.56	0.450	−0.03%
features-service	0.2	0.41	0.237	−2.82%	0.57	0.341	+1.68%	**0.65**	**0.049**	**+4.64%**
	0.5	0.53	0.547	+0.96%	0.43	0.325	−2.04%	0.39	0.053	−2.97%
proxyprint	0.2	0.51	0.918	+0.56%	0.57	0.411	+1.77%	0.55	0.393	+1.20%
	0.5	0.59	0.154	+1.55%	**0.73**	**<0.001**	**+5.03%**	0.62	0.073	+3.42%
scout-api	0.2	0.59	0.112	+1.17%	0.64	0.141	+1.49%	0.55	0.360	+0.32%
	0.5	0.59	0.301	+0.85%	**0.74**	**<0.001**	**+2.39%**	0.64	0.125	+1.52%
%Branches										
rest-ncs	0.2	0.50	NaN	+0.00%	0.50	NaN	+0.00%	0.50	NaN	+0.00%
	0.5	0.50	1.000	−0.02%	0.48	1.000	−0.04%	0.49	1.000	−0.02%
rest-scs	0.2	0.40	0.133	−2.00%	0.40	0.132	−1.75%	0.54	0.729	+0.25%
	0.5	0.46	0.829	−0.68%	0.53	0.847	+0.54%	0.55	0.471	+1.23%
rest-news	0.2	0.53	0.674	+0.48%	0.49	0.909	−0.51%	0.46	0.596	−0.98%
	0.5	0.52	0.663	+0.37%	0.53	0.745	+0.51%	0.51	0.837	+0.14%
catwatch	0.2	0.44	0.293	−0.79%	**0.74**	**0.003**	**+2.61%**	**0.77**	**<0.001**	**+3.43%**
	0.5	0.52	0.532	+0.32%	0.48	0.788	−0.23%	0.46	0.581	−0.55%
features-service	0.2	0.42	0.271	−5.13%	0.60	0.172	+5.47%	**0.65**	**0.024**	**+11.18%**
	0.5	0.54	0.592	+2.05%	0.45	0.368	−3.52%	0.40	0.098	−5.46%
proxyprint	0.2	0.49	0.861	−0.20%	0.59	0.327	+5.09%	0.59	0.186	+5.30%
	0.5	0.61	0.097	+3.50%	**0.74**	**<0.001**	**+15.46%**	0.62	0.067	+11.55%
scout-api	0.2	0.42	0.380	−0.79%	0.47	0.726	−0.65%	0.57	0.588	+0.14%
	0.5	0.55	0.653	+0.83%	0.63	0.135	+1.11%	0.58	0.925	+0.28%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 19. **RQ2**: With $SQL = T$, Pair Comparison for ρ Settings for Each Setting of d with #Targets using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

projects	d	A (ρ = 0.2), B (ρ = 0.5) #Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.2	0.64	0.071	+0.46%	0.61	0.109	+0.35%	**0.62**	**0.012**	**+0.75%**
	0.5	0.64	0.082	+0.45%	0.61	0.106	+0.35%	**0.62**	**0.013**	**+0.73%**
	0.8	0.63	0.079	+0.45%	0.62	0.085	+0.37%	**0.60**	**0.012**	**+0.70%**
rest-scs	0.2	**0.87**	**<0.001**	**+4.72%**	**0.89**	**<0.001**	**+4.06%**	**0.72**	**0.005**	**+4.56%**
	0.5	**0.93**	**<0.001**	**+5.96%**	**0.90**	**<0.001**	**+4.93%**	**0.77**	**<0.001**	**+5.96%**
	0.8	**0.95**	**<0.001**	**+6.34%**	**0.94**	**<0.001**	**+5.39%**	**0.90**	**<0.001**	**+7.00%**
rest-news	0.2	0.20	<0.001	−3.04%	0.09	<0.001	−4.63%	0.39	0.122	−2.21%
	0.5	0.43	0.337	−0.82%	0.45	0.473	−0.41%	0.39	0.142	−2.31%
	0.8	0.39	0.114	−1.43%	0.37	0.060	−1.37%	0.44	0.380	−1.20%
catwatch	0.2	0.28	0.007	−2.73%	0.28	0.008	−2.49%	0.50	0.970	−0.01%
	0.5	0.52	0.797	−0.36%	0.51	0.912	−0.12%	0.58	0.342	+1.11%
	0.8	0.00	<0.001	−7.81%	0.00	<0.001	−7.01%	0.26	<0.001	−2.78%
features-service	0.2	0.58	0.243	+0.49%	0.54	0.561	+0.30%	0.55	0.504	+2.37%
	0.5	0.62	0.083	+3.93%	**0.64**	**0.038**	**+4.21%**	**0.64**	**0.034**	**+10.13%**
	0.8	0.43	0.345	−3.00%	0.40	0.135	−3.37%	0.40	0.141	−6.36%
proxyprint	0.2	0.51	0.904	+0.27%	0.51	0.923	+0.21%	0.50	1.000	−1.67%
	0.5	0.57	0.325	+1.19%	0.58	0.238	+1.20%	0.60	0.166	+1.98%
	0.8	**0.67**	**0.017**	**+3.67%**	**0.66**	**0.021**	**+3.41%**	**0.65**	**0.036**	**+8.03%**
scout-api	0.2	0.54	0.533	+0.90%	0.52	0.776	+0.79%	0.45	0.457	−0.14%
	0.5	0.54	0.585	+0.71%	0.53	0.643	+0.48%	0.57	0.286	+1.49%
	0.8	**0.64**	**0.042**	**+1.86%**	0.63	0.057	+1.68%	0.61	0.119	+1.63%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 20. **RQ3**: Pair Comparison for P_A Settings for Each Setting of EAM with #Targets using Mann–Whitney–Wilcoxon U-tests (p-$value$) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

projects	EAM	A (P_A = 0.5), B (P_A = 0.8) #Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	F	0.52	0.763	+0.07%	0.51	0.832	+0.09%	0.51	0.884	+0.06%
	T	0.46	0.405	−0.05%	0.47	0.445	−0.01%	0.46	0.399	−0.15%
rest-scs	F	0.56	0.333	+0.61%	0.55	0.647	+0.35%	0.57	0.205	+1.68%
	T	0.52	0.891	+0.12%	0.62	0.345	+0.76%	0.46	0.757	−0.46%
rest-news	F	0.67	0.133	+2.54%	0.64	0.193	+1.16%	0.67	0.124	+4.29%
	T	0.32	0.004	−2.49%	0.35	0.015	−1.61%	0.39	0.069	−2.34%
catwatch	F	0.58	0.295	−0.10%	0.58	0.267	−0.11%	0.56	0.402	+0.76%
	T	0.43	0.259	−1.09%	0.42	0.191	−1.25%	0.49	0.828	−0.17%
features-service	F	0.53	0.687	+0.53%	0.49	0.925	+0.24%	0.50	0.972	−0.07%
	T	0.41	0.136	−2.17%	0.41	0.115	−2.39%	0.42	0.193	−4.55%
proxyprint	F	0.48	0.803	−1.80%	0.49	0.875	−1.63%	0.43	0.352	−6.13%
	T	0.56	0.253	+0.50%	0.54	0.454	+0.37%	0.44	0.243	+0.10%
scout-api	F	0.57	0.376	+0.55%	0.60	0.193	+0.60%	0.61	0.149	+1.53%
	T	0.39	0.013	−0.44%	0.20	<0.001	−1.57%	0.54	0.392	+0.59%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 21. **RQ3**: Pair Comparison for *EAM* Settings for Each Setting of P_A with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

		A (EAM = F), B (EAM = T)								
projects	P_A	#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.5	**0.89**	**<0.001**	**+0.92%**	0.56	0.223	+0.06%	**0.85**	**<0.001**	**+1.65%**
	0.8	**0.86**	**<0.001**	**+0.80%**	0.52	0.709	−0.03%	**0.79**	**<0.001**	**+1.44%**
rest-scs	0.5	0.55	0.450	+0.45%	0.56	0.343	+0.46%	0.53	0.668	+0.73%
	0.8	0.49	0.966	−0.04%	0.60	0.456	+0.87%	0.40	0.467	−1.39%
rest-news	0.5	0.58	0.300	+1.16%	0.50	1.000	−0.14%	0.65	0.053	+3.59%
	0.8	0.25	0.010	−3.80%	0.23	0.005	−2.88%	0.35	0.124	−3.00%
catwatch	0.5	**0.68**	**0.002**	**+2.79%**	**0.68**	**0.002**	**+2.68%**	**0.66**	**0.005**	**+2.12%**
	0.8	0.64	0.066	+1.76%	0.64	0.054	+1.51%	0.60	0.189	+1.17%
features-service	0.5	0.51	0.824	+0.10%	0.51	0.888	+0.08%	0.52	0.707	+1.08%
	0.8	0.40	0.169	−2.59%	0.42	0.290	−2.55%	0.44	0.403	−3.45%
proxyprint	0.5	**0.92**	**<0.001**	**+6.65%**	**0.92**	**<0.001**	**+6.25%**	**0.88**	**<0.001**	**+8.87%**
	0.8	**0.94**	**<0.001**	**+9.16%**	**0.94**	**<0.001**	**+8.40%**	**0.88**	**<0.001**	**+16.10%**
scout-api	0.5	**0.92**	**<0.001**	**+5.12%**	**0.91**	**<0.001**	**+4.31%**	**0.80**	**<0.001**	**+3.64%**
	0.8	**0.97**	**<0.001**	**+4.09%**	**0.81**	**<0.001**	**+2.07%**	**0.72**	**<0.001**	**+2.68%**

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 22. **RQ4**: With E5 Experiment Setting, for Each of *FS*, Pair Comparison P_r with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

		A (P_r = 0.2), B (P_r = 0.5)								
projects	FS	#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.5	0.53	0.465	+0.04%	0.54	0.559	−0.05%	0.56	0.239	+0.27%
	0.8	0.52	0.758	+0.02%	0.59	0.288	+0.01%	0.53	0.543	+0.15%
rest-scs	0.5	0.59	0.265	+2.16%	0.59	0.258	+1.76%	**0.66**	**0.038**	**+4.78%**
	0.8	0.49	0.880	−0.12%	0.46	0.607	−0.54%	0.56	0.510	+1.72%
rest-news	0.5	0.47	0.713	−0.47%	0.37	0.084	−1.27%	0.51	0.923	+0.20%
	0.8	0.64	0.076	+1.63%	**0.69**	**0.014**	**+1.78%**	0.60	0.210	+2.21%
catwatch	0.5	**0.68**	**0.024**	**+2.60%**	**0.66**	**0.044**	**+2.78%**	**0.71**	**0.007**	**+3.04%**
	0.8	0.56	0.234	+1.11%	0.54	0.269	+1.18%	0.53	0.605	+0.47%
features-service	0.5	0.45	0.440	−1.69%	0.44	0.413	−1.85%	0.43	0.303	−3.29%
	0.8	0.53	0.459	+0.64%	0.52	0.580	+0.55%	0.52	0.685	+1.18%
proxyprint	0.5	0.40	0.156	−2.82%	0.39	0.124	−2.84%	0.43	0.292	−7.67%
	0.8	0.53	0.678	+0.66%	0.51	0.886	+0.12%	0.50	0.955	−1.10%
scout-api	0.5	**0.69**	**0.008**	**+2.81%**	**0.65**	**0.029**	**+2.34%**	**0.69**	**0.008**	**+5.15%**
	0.8	**0.88**	**<0.001**	**+3.79%**	**0.87**	**<0.001**	**+3.39%**	**0.72**	**0.007**	**+4.33%**

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 23. **RQ4**: With E5 Experiment Setting, for Each of P_r, Pair Comparison *FS* with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

		A (FS = 0.5), B (FS = 0.8)								
projects	P_r	#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.2	0.45	0.507	−0.18%	0.48	0.761	−0.12%	0.48	0.806	−0.15%
	0.5	0.45	0.474	−0.19%	0.54	0.587	−0.05%	0.45	0.513	−0.27%
rest-scs	0.2	0.58	0.289	+1.36%	0.60	0.218	+1.53%	0.55	0.466	+1.34%
	0.5	0.48	0.808	−0.91%	0.47	0.733	−0.76%	0.44	0.452	−1.62%
rest-news	0.2	0.28	0.009	−2.37%	0.08	<0.001	−4.19%	0.48	0.812	−0.39%
	0.5	0.47	0.697	−0.32%	0.38	0.098	−1.24%	0.57	0.329	+1.60%
catwatch	0.2	0.52	0.469	+0.65%	0.51	0.459	+0.77%	0.57	0.197	+1.14%
	0.5	0.40	0.220	−0.81%	0.38	0.114	−0.80%	0.39	0.166	−1.38%
features-service	0.2	0.46	0.758	−0.47%	0.48	0.820	−0.25%	0.49	0.891	−0.24%
	0.5	0.56	0.406	+1.90%	0.57	0.316	+2.18%	0.59	0.208	+4.38%
proxyprint	0.2	0.33	0.025	−3.63%	0.33	0.025	−3.31%	0.34	0.034	−7.51%
	0.5	0.47	0.718	−0.17%	0.46	0.596	−0.37%	0.44	0.442	−0.92%
scout-api	0.2	0.48	0.711	−0.21%	0.44	0.530	−0.63%	0.55	0.819	+1.08%
	0.5	0.59	0.236	+0.74%	0.54	0.586	+0.39%	0.54	0.631	+0.30%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 24. **RQ4**: With E6 Experiment Setting, for Each of P_r, Pair Comparison *FS* with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

		A (P_r = 0.2), B (P_r = 0.5)								
projects	FS	#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.5	0.60	0.148	+0.13%	0.56	0.334	+0.06%	0.59	0.154	+0.27%
	0.8	0.55	0.507	−0.09%	0.49	0.861	−0.21%	0.56	0.460	+0.14%
rest-scs	0.5	0.56	0.660	+0.49%	0.49	0.952	+0.07%	0.61	0.389	+1.79%
	0.8	0.52	0.838	−0.00%	0.46	0.632	−0.26%	0.57	0.391	+0.93%
catwatch	0.5	0.28	0.003	−4.01%	0.30	0.005	−3.84%	0.43	0.317	−0.78%
	0.8	0.30	0.011	−3.01%	0.29	0.006	−3.27%	0.37	0.104	−2.09%
features-service	0.5	0.47	0.699	−0.48%	0.50	0.975	−0.16%	0.50	1.000	+2.02%
	0.8	0.50	0.994	+1.55%	0.51	0.894	+1.61%	0.52	0.837	+1.31%
proxyprint	0.5	**0.81**	**<0.001**	**+2.20%**	**0.79**	**<0.001**	**+1.92%**	**0.75**	**<0.001**	**+3.45%**
	0.8	0.52	0.934	+0.08%	0.53	0.905	+0.18%	0.53	0.847	+0.34%
scout-api	0.5	**0.75**	**<0.001**	**+1.90%**	**0.68**	**0.006**	**+1.30%**	0.61	0.082	+1.48%
	0.8	0.52	0.577	+0.16%	0.45	0.432	−0.28%	0.49	0.470	−0.21%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

Table 25. **RQ4**: With E6 Experiment Setting, for Each of *FS*, Pair Comparison P_r with #Targets using Mann–Whitney–Wilcoxon U-tests (*p-value*) and Vargha–Delaney Effect Sizes ($\hat{A}_{xy}$)

		A (FS = 0.5), B (FS = 0.8)								
projects	P_r	#Targets			%Lines			%Branches		
		$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$	$\hat{A}_{ba}$	p-value$_{ba}$	relative$_{b-a/a}$
rest-ncs	0.2	**0.65**	**0.019**	**+0.30%**	0.58	0.115	+0.27%	0.62	0.082	+0.32%
	0.5	0.54	0.561	+0.08%	0.50	0.970	+0.01%	0.54	0.556	+0.19%
rest-scs	0.2	0.43	0.445	−0.55%	0.46	0.709	−0.29%	0.47	0.489	−0.91%
	0.5	0.37	0.295	−1.04%	0.40	0.402	−0.62%	0.38	0.351	−1.74%
catwatch	0.2	0.44	0.432	−1.28%	0.43	0.322	−1.35%	0.54	0.492	+0.66%
	0.5	0.49	0.850	−0.24%	0.44	0.386	−0.76%	0.46	0.629	−0.67%
features-service	0.2	0.32	0.026	−5.40%	0.35	0.041	−5.37%	0.34	0.045	−9.49%
	0.5	0.37	0.165	−3.47%	0.36	0.094	−3.70%	0.37	0.037	−10.11%
proxyprint	0.2	0.54	0.734	−0.83%	0.55	0.837	−0.78%	0.52	0.800	−3.48%
	0.5	0.28	0.002	−2.89%	0.32	0.008	−2.47%	0.31	0.005	−6.38%
scout-api	0.2	**0.67**	**0.028**	**+1.22%**	0.64	0.076	+1.01%	0.59	0.212	+1.17%
	0.5	0.43	0.258	−0.51%	0.38	0.076	−0.58%	0.45	0.431	−0.51%

Values in **bold** mean the setting x is statistically significant better than the setting y, whereas values in red mean y is statistically significant better than x.

REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.

[2] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175–203. DOI: https://doi.org/10.1002/stvr.v16:3

[3] Denis Antipov and Benjamin Doerr. 2020. Runtime analysis of a heavy-tailed (1+(λ, λ)) genetic algorithm on jump functions. In *International Conference on Parallel Problem Solving from Nature*. Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann (Eds.) Springer, 545–559.

[4] Andrea Arcuri. 2018. EvoMaster: Evolutionary multi-context automated system test generation. In *IEEE 11th International Conference on Software Testing, Verification and Validation*. IEEE. DOI: 10.1109/ICST.2018.00046

[5] Andrea Arcuri. 2017. An experience report on applying software testing academic results in industry: We need usable automated test generation. *Empirical Software Engineering* 23, 2 (2017), 1–23.

[6] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206. DOI: https://doi.org/10.1016/j.infsof.2018.05.003

[7] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019), 1–37.

[8] Andrea Arcuri. 2021. Automated blackbox and whitebox testing of RESTful APIs with EvoMaster. *IEEE Software* 38, 3 (2021), 72–78.

[9] A. Arcuri and L. Briand. 2011. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 11th International Symposium on Software Testing and Analysis*. ACM, New York, NY, 265–275.

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

[11] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[12] Andrea Arcuri and Juan Pablo Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 1–31.

[13] Andrea Arcuri and Juan Pablo Galeotti. 2020. Testability transformations for existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*. IEEE, 153–163. DOI: 10.1109/ICST46399.2020.00025

[14] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2020. *EvoMaster: A Search-Based System Test Generation Tool*. Zenodo. DOI: https://doi.org/10.5281/zenodo.4300745

[15] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A search-based system test generation tool. *Journal of Open Source Software* 6, 57 (2021), 2153.

[16] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. arXiv:2005.11498. Retrieved from https://arxiv.org/abs/2005.11498.

[17] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* . IEEE Press, 748–758. DOI: https://doi.org/10.1109/ICSE.2019.00083

[18] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235. DOI: https://doi.org/10.1016/j.infsof.2018.08.010

[19] Leandro Nunes Castro, Leandro Nunes De Castro, and Jonathan Timmis. 2002. *Artificial Immune Systems: A new Computational Intelligence Approach.* Springer Science & Business Media.

[20] Jun Chen and Mahdi Mahfouf. 2006. A population adaptive based immune algorithm for solving multi-objective optimization problems. In *International Conference on Artificial Immune Systems.* H. Bersini and J. Carneiro (Eds.), Springer, 280–293.

[21] Chien-Wei Chu, Min-Der Lin, Gee-Fon Liu, and Yung-Hsing Sung. 2008. Application of immune algorithms on solving minimum-cost problem of water distribution network. *Mathematical and Computer Modelling* 48, 11-12 (2008), 1888–1900.

[22] Helen G. Cobb. 1990. *An Investigation into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuous, Time-Dependent Nonstationary Environments.* Naval Research Lab Washington DC.

[23] Dogan Corus, Pietro S. Oliveto, and Donya Yazdani. 2020. When hypermutations and ageing enable artificial immune systems to outperform evolutionary algorithms. *Theoretical Computer Science* 832 (2020), 166–185. DOI: https://doi.org/10.1016/j.tcs.2019.03.002

[24] Vincenzo Cutello, Giuseppe Nicosia, and Mario Pavone. 2004. Exploring the capability of immune algorithms: A characterization of hypermutation operators. In *Artificial Immune Systems.* Giuseppe Nicosia, Vincenzo Cutello, Peter J. Bentley, and Jon Timmis (Eds.). Springer, Berlin, 263–276.

[25] Leandro N. De Castro and Fernando J. Von Zuben. 2002. Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation* 6, 3 (2002), 239–251.

[26] Benjamin Doerr and Carola Doerr. 2020. Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices. *Theory of Evolutionary Computation* (2020), 271–321. DOI: 10.1007/978-3-030-29414-4_6

[27] Benjamin Doerr, Carola Doerr, and Johannes Lengler. 2019. Self-adjusting mutation rates with provably optimal success rules. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 1479–1487.

[28] Carola Doerr and Markus Wagner. 2018. Sensitivity of parameter control mechanisms with respect to their initialization. In *International Conference on Parallel Problem Solving from Nature.* A. Auger, C. Fonseca, N. Lourenco, P. Machado, L. Paquete, and D. Whitley (Eds.), Springer, 360–372.

[29] Carola Doerr and Markus Wagner. 2018. Simple on-the-fly parameter selection mechanisms for two classical discrete black-box optimization benchmark problems. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 943–950.

[30] S. Droste, T. Jansen, and I. Wegener. 1998. On the optimization of unimodal functions with the (1 + 1) evolutionary algorithm. In *Proceedings of the International Conference on Parallel Problem Solving from Nature.* 13–22.

[31] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference.* 181–190. DOI: 10.1109/EDOC.2018.00031

[32] Á. E. Eiben, R. Hinterding, and Z. Michalewicz. 1999. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3, 2 (1999), 124–141.

[33] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-Based Software Architectures.* Ph.D. Dissertation. University of California, Irvine. UMI Order Number: AAI 9980887.

[34] Gordon Fraser and Andrea Arcuri. 2011. 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.* 416–419.

[35] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite at the SBST 2013 Tool Competition. In *2013 IEEE 6th International Workshop on Search-Based Software Testing.* 406–409.

[36] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[37] Tobias Friedrich, Andreas Göbel, Francesco Quinzan, and Markus Wagner. 2018. Heavy-tailed mutation operators in single-objective combinatorial optimization. In *International Conference on Parallel Problem Solving from Nature.* A. Auger, C. Fonseca, N. Lourenco, P. Machado, L. Paquete, and D. Whitley (Eds.), Springer, 134–145.

[38] Tobias Friedrich, Francesco Quinzan, and Markus Wagner. 2018. Escaping large deceptive basins of attraction with heavy-tailed mutation operators. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 293–300.

[39] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 725–736. DOI: https://doi.org/10.1145/3368089.3409719
[40] D. E. Goldberg. 1989. *Genetic Algorithms in Search and Optimization*. Addison-wesley.
[41] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* 45, 1 (2012), 11.
[42] Zhengxin Huang and Yuren Zhou. 2020. Runtime analysis of somatic contiguous hypermutation operators in MOEA/D Framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 2359–2366.
[43] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *International Conference on Principles and Practice of Constraint Programming*. F. Benhamou (Eds.), Springer, 213–228.
[44] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206 (2014), 79–111. DOI: https://doi.org/10.1016/j.artint.2013.10.003
[45] T. Jansen and C. Zarges. 2014. Reevaluating immune-inspired hypermutations using the fixed budget perspective. *IEEE Transactions on Evolutionary Computation* 18, 5 (2014), 674–688. DOI: https://doi.org/10.1109/TEVC.2014.2349160
[46] Giorgos Karafotias, Mark Hoogendoorn, and Ágoston E. Eiben. 2014. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation* 19, 2 (2014), 167–187.
[47] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI described RESTful APIs. In *IEEE 13th International Conference on Software Testing, Verification and Validation*. IEEE.
[48] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.
[49] Anton Kotelyanskii and Gregory M. Kapfhammer. 2014. Parameter tuning for search-based test-data generation revisited: Support for previous results. In *2014 14th International Conference on Quality Software*. IEEE, 79–84. DOI: 10.1109/QSIC.2014.43
[50] Johannes Lengler. 2019. A general dichotomy of evolutionary algorithms on monotone functions. *IEEE Transactions on Evolutionary Computation* 24, 6 (2019), 995–1009.
[51] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. 2002. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *International Conference on Principles and Practice of Constraint Programming*. P. Van Hentenryck (Eds.), Springer, 556–572.
[52] Q. Lin, J. Chen, Z. Zhan, W. Chen, C. A. C. Coello, Y. Yin, C. Lin, and J. Zhang. 2016. A hybrid evolutionary immune algorithm for multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation* 20, 5 (2016), 711–729. DOI: https://doi.org/10.1109/TEVC.2015.2512930
[53] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.
[54] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing*. E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari (Eds.), Springer.
[55] P. McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
[56] Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun, and Joachim Wegener. 2011. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering* 38, 2 (2011), 453–477.
[57] Phil McMinn and Gregory M. Kapfhammer. 2016. AVMf: An open-source framework and implementation of the alternating variable method. In *International Symposium on Search Based Software Engineering*. F. Sarro and K. Deb (Eds.), Springer, 259–266.
[58] Vladimir Mironovich and Maxim Buzdalov. 2017. Evaluation of heavy-tailed mutation operator on maximum flow test generation problem. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1423–1426.
[59] Sam Newman. 2015. *Building Microservices*. "O'Reilly Media, Inc.".
[60] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2018. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.
[61] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.
[62] Abdel Salam Sayyad, Katerina Goseva-Popstojanova, Tim Menzies, and Hany Ammar. 2013. On parameter tuning in search based software engineering: A replicated empirical study. In *2013 3rd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, 84–90.

[63] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated black-box testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation*. IEEE. DOI: 10.1109/ICST46399.2020.00024

[64] Louis F. Williams. 1976. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th Annual Southeast Regional Conference*. ACM, New York, NY, 95–101. DOI: https://doi.org/10.1145/503561.503582

[65] D. H. Wolpert and W. G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82.

[66] Furong Ye, Carola Doerr, and Thomas Bäck. 2019. Interpolating local and global search by controlling the variance of standard bit mutation. In *2019 IEEE Congress on Evolutionary Computation*. IEEE, 2292–2299.

[67] Shayan Zamani and Hadi Hemmati. 2019. Revisiting hyper-parameter tuning for search-based test data generation. In *International Symposium on Search Based Software Engineering*. S. Nejati and G. Gay (Eds.), Springer, 137–152.

[68] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1426–1434.

Received December 2020; revised March 2021; accepted May 2021