UNIVERSIDADE Ð COIMBRA

Carlos Francisco Fernandes Santos

EVOLUTIONARY ROBUSTNESS TESTING OF REST SERVICES

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Nuno Laranjeiro and Professor Nuno Lourenço and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2022

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Carlos Francisco Fernandes Santos

Evolutionary Robustness Testing of REST services

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Nuno Laranjeiro and Professor Nuno Lourenço and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2022

Acknowledgements

This work is funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020; it is also supported by VALU3S project, which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

Regarding personal acknowledgments, I would like to thank Professor Nuno Laranjeiro and Professor Nuno Lourenço for the constant meetings, reviews, and their **time** to help improve my work throughout the whole year. I also would like to sincerely thank my whole group of friends, which stayed together while helping each other in challenging times. To my Ritas in life, thank you for always staying by my side and supporting me. Finally, as an ending note, I could not miss mentioning my mother, who does everything for me. Thank you for everything, especially for wanting my best at all costs.

Abstract

Today's companies, including Google, Facebook (Meta), Instagram, and Twitter, depend heavily on REST-based services. In this type of environment, these services are highly exposed to unexpected scenarios, which may lead to service failures. Robustness, therefore, is a crucial feature of REST services.

Robustness is the degree to which a particular system or component can operate correctly in the presence of invalid input or stressful conditions. Due to these services' increasing use, interconnection, and complexity, acquiring assurances concerning their robustness has become an essential part of their development process. Even more so, when these services support critical systems, where a failure can have significant consequences for the business or even for people's lives.

Unlike SOAP services, which have been widely tested for robustness, REST services have not undergone the same scrutiny. Despite its extensive use, little research has been done on the topic. As a result, only a few approaches for black-box testing of REST services have emerged, and all face the problem of generating high-quality workloads (e.g., inputs that allow good code coverage), which is an open and difficult challenge, especially from a black-box perspective.

In this dissertation, we present an evolutionary mechanism called EvoReFuzz for robustness testing of REST services. Although several approaches and software testing tools have been studied and applied to a wide range of problems, REST services need new practices in the intelligent generation of quality inputs and the improvement of the exhaustive process of verification and validation. Also, the potential of using evolutionary computation for this purpose has been mostly disregarded. Therefore, we fill in this gap by proposing EvoReFuzz, a tool that uses an evolutionary algorithm to automatically generate valid and invalid inputs solely based on the OpenAPI interface description and the observed external behavior of the service.

We used EvoReFuzz to evaluate 12 different services, 11 public real-world APIs, and one private. The experimental results on the 11 real-world RESTful APIs demonstrate the effectiveness and efficiency of EvoReFuzz, where we were able to disclose 28 unique robustness problems in the GitLab and Microsoft Bing Maps services, such as Run Time errors. These results depict that REST services are being deployed online, holding software bugs. In addition, the lack of parameter validation is one of the most common implementation errors, and flawed practices while specifying the OpenAPI files are widespread. Private services are also included in this group, where we could find four different bugs and bad implementation practices in an implemented API for a framework. Moreover, we made a code coverage comparison between EvoReFuzz and the state-of-the-art testing tool EvoMaster, in which both approaches had a relatively close performance.

Keywords

Software testing, robustness evaluation, RESTful APIs, Evolutionary Testing, Genetic algorithms

Resumo

As empresas de atualmente, incluindo Google, Facebook (Meta), Instagram e Twitter, dependem fortemente de serviços baseados em REST. Consequentemente, os serviços web, em particular, estão constantemente expostos a cenários inesperados, que podem ou não levar a falhas no serviço. Fazendo com que a Robustez seja, portanto, uma propriedade essencial dos serviços REST.

Robustez é o grau em que um determinado sistema ou componente pode operar corretamente na presença de entradas inválidas ou em condições de stress. Devido ao crescente uso, interconexão e complexidade desses serviços, adquirir garantias sobre a sua robustez tornou-se uma parte importante do seu processo de desenvolvimento. Sendo particularmente essencial, quando esses serviços suportam sistemas críticos, onde uma falha pode ter consequências significativas para o negócio ou mesmo para a vida das pessoas.

Enquanto os serviços SOAP, que foram amplamente testados quanto à sua robustez, os serviços REST não passaram pelo mesmo escrutínio. Apesar de seu alargado uso, poucas pesquisas foram feitas sobre o tema. Como resultado, apenas algumas abordagens para testes de caixa-preta de serviços REST surgiram e todas enfrentam o problema de gerar cargas de trabalho de alta qualidade (por exemplo, entradas que permitem uma boa cobertura de código), o que é um desafio aberto e difícil, especialmente do ponto de vista da caixa-preta.

Nesta dissertação, apresentamos um mecanismo evolutivo chamado EvoReFuzz para testes de robustez de serviços REST. Embora diversas abordagens tenham sido estudadas e aplicadas a uma ampla gama de problemas, os serviços REST necessitam de novas práticas na geração inteligente de entradas de qualidade e no aprimoramento do processo exaustivo de verificação e validação. No entanto, o potencial do uso da computação evolutiva para este propósito tem sido em grande parte desconsiderado. Portanto, preenchemos esta lacuna ao propor o EvoReFuzz, uma ferramenta que utiliza um algoritmo evolucionário para gerar automaticamente entradas válidas e inválidas apenas com base na descrição da interface OpenAPI e no comportamento externo observado do serviço.

Usamos o EvoReFuzz para avaliar 12 serviços diferentes, 11 APIs públicas do mundo real e uma privada. Os resultados experimentais nas APIs públicas demonstram a eficácia e eficiência do EvoReFuzz, onde foram descobertos 28 problemas únicos de robustez nos serviços GitLab e Microsoft Bing Maps. Estes resultados exibem que os serviços REST estão a ser publicados online, com bugs presentes no software. Além disso, os serviços privados também estão incluídos, onde encontramos 4 bugs únicos. Assim, a falta de validação dos parâmetros de entrada é um dos erros de implementação mais comuns, como as más práticas ao descrever as interfaces dos serviços. Em adição, fizemos uma comparação de cobertura de código entre o EvoReFuzz e a ferramenta de teste EvoMaster, na qual ambas as abordagens tiveram um desempenho relativamente próximo.

Palavras-Chave

Teste de Software, avaliação de robustez, RESTful APIs, Teste Evolutivo, Algortimos genéticos

Contents

Acronyms

API Application Programming Interface.

bBOXRT black-BOX tool for Robustness Testing of rest services.

EA Evolutionary Algorithm.

ET Evolutionary Testing.

GA Genetic Algorithm.

GAS Genetic Algorithms.

GSA Genetic Simulated Annealing.

HTTP Hypertext Transfer Protocol.

IDL inter-parameter dependency language.

MA Memetic algorithm.

MIO Many Independent Objective.

NE Naive Evolution.

PBT Property-based Testing.

PN Parallel Niching.

PSO Particle Swarm Optimization.

REST REpresentational State Transfer.

RT Random Testing.

RTS Restricted Tournament Selection.

SA Simulated Annealing.

SA/AAN Simulated Annealing with Advanced Adaptive Neighborhood.

SCGA Species Conserving Genetic Algorithm.

SN Sequential Niching.

SUT System under testing.

URI Uniform Resource Identifier.

List of Figures

List of Tables

Chapter 1

Introduction

Most of the services available today depend on software systems. These systems must be robust to avoid failures and thus prevent unexpected conditions. Web services are particularly susceptible to unexpected scenarios due to the fact that they are often exposed to abnormal and malicious inputs which, if left unchecked and unaddressed, can lead to problems such as data loss and disclosure of private information [1].

The majority of web services available today are REpresentational State Transfer (REST) services [2] which follow the REST architecture [3], and large technology companies such as Facebook, Instagram, Twitter, or Google, have their products publicly deployed via REST services. The resources available through REST (i.e., RESTful service, REST API) are identified by Uniform Resource Identifiers (URI) and can be manipulated with actions based on the semantics of the predefined HTTP verbs, such as GET, POST, DELETE, and PUT [4, 5]. Usually, REST rely solely on the Hypertext Transfer Protocol (HTTP) for message transfer communication. Additionally, it has have a relatively loose architectural style where the presence of an interface description document (e.g., a WSDL document) is not mandatory. Consequently, there is no standard way to describe an interface of a RESTful API, but an OpenAPI specification [6] is among the most popular.

Given the lack of formal description and the common public access, many potential inputs can be sent to these services. As a result, if such inputs are not validated, they can activate software faults, which can cause the system to go into an erroneous state, which can lead to a failure [7]. The latter is something that is evident at the system's boundaries and indicates a deviation from the expected behavior. Furthermore, there should be a concern for preventing server crashes and incorrect responses. Even more so if the service in question is a business or mission critical.

Unlike SOAP services, which have been widely tested for robustness, REST services have not been subjected to the same examination [1]. Despite its widespread use, there is little literature on the topic. As a result, a small number of automated tools for software testing of REST services have been developed, and most of them use dictionaries or random inputs for data generation [8–13]. It is argued that software engineering is ideal for the application of metaheuristic search techniques where the search-based technique must outperform the random technique [14]. Furthermore, several observations support the idea that the random technique may not be suitable for industrial applications with huge input spaces [15, 16].

To test REST services, we followed a black-box approach since it only needs a description of the system interface and does not require knowledge of the artifacts or the code itself. In this case, and among other aspects, designing good quality tests (e.g., tests that have high code coverage) is much more challenging, but, at the same time, designing and executing tests may be more straightforward as the whole system is viewed from an external point (many times, the tests represent user operations, which tend to be accessible to model).

In many cases, black-box is actually a mandatory option to use (e.g., closed-source systems, participants in a service mesh provided by external entities, tests performed for specific certification purposes). Additionally, it allows the generalization of testing different REST services that are heterogeneous in an automated way without extensive and challenging incorporation into the system's code. Therefore, any tester may use our approach to test any RESTful API with an interface specification file (e.g., OpenAPI [6]).

Regarding the related work, we started by analyzing different Evolutionary Algorithms and their components, such as the fitness function, variation operators, parent selection, and population. Further, we present the related work for software testing, focusing on state-of-the-art tools for testing RESTful APIs. After this analysis, we use the collected knowledge to highlight the lack of academic support for solutions to evaluate the robustness of RESTful APIs using evolutionary algorithms. We seek to unravel this problem by proposing a novel approach.

In this thesis, we present an evolutionary approach for testing the robustness of REST services that takes advantage of the components that compose an Evolutionary Algorithm to generate valid requests in the form of a workload and to generate invalid ones in the form of faultload (i.e., empty values, boundary values, invalid strings) that will be sent to the service to test its robustness. Our approach, named EvoReFuzz, is capable of producing these requests in an automated way assisted by a Genetic Algorithm, which operates as a client application. The Genetic Algorithm evaluates its individuals by analyzing the returned responses from the service. With such information, our evolutionary approach generates high-quality solutions to optimize the search problems, which is, in our case, producing valid (i.e., requests that originate a response with status code 200) and invalid requests (i.e., requests that originate a response with status code 500). This methodology aims to produce high-quality input to obtain an acceptable code coverage of the system under testing (SUT).

To demonstrate the usefulness and usability of our approach in finding robustness problems, we then performed tests over a set of 12 services, where 11 are real-world services from GitLab [17] and Microsoft Bing Maps [18], and one private service in which we partner with a Masters's student to evaluate the API developed for the framework's dissertation. We conducted over 500,000 tests, in which we discovered 28 unique bugs for the real-world services and four for the private service. This information is essential to the testers and developers. It also helps find bad practices over the conventional RESTful APIs practices and the lack of input validation, which is very common while developing complex systems. Moreover, we performed an experiment to compare the code coverage between our approach, EvoReFuzz, and the state-of-the-art tool EvoMaster [19] over six different systems with different levels of complexity, functions, and code logic.

The main contributions of this dissertation are the following:

- The definition of an Evolutionary approach for testing the robustness of REST services using the components of a Genetic Algorithm.
- A robustness framework, named EvoReFuzz, using our Evolutionary approach available in [20] to be used by testers and developers.
- The practical application of EvoReFuzz to a set of 11 real-world RESTful APIs and one private, which demonstrates the capability of producing robustness problems in these services and finding lousy programming practices in the service itself as well as in the specification file.
- A practical code coverage comparison between EvoReFuzz and the state-of-the-art tool EvoMaster [19, 21], to a set of six REST services.

The outcomes of this dissertation incorporate a paper submission, which is under evaluation at the time of writing. We submit to PRDC 2022 (Pacific Rim International Symposium on Dependable Computing) reporting on "A Framework for Evolutionary Black-box Testing of REST Services". Moreover, we are also currently writing a second paper with the full description of our approach and its experimental results to submit to GECCO 2023 (Genetic and Evolutionary Computation Conference).

The remainder of this document is organized as follows. Chapter 2 provides a context background on concepts addressing the REST architecture, software testing, and Evolutionary algorithms. In chapter 3, we begin by exploring the different Evolutionary Algorithms applied in the literature with a thorough analysis, particularly in Genetic algorithms. Furthermore, a study of relevant work in software testing was undertaken, with particular attention paid to tools for testing REST services as well as various applications that could benefit from the use of evolutionary algorithms. Chapter 4 describes our approach for testing the robustness of REST services using an Evolutionary Algorithm, its architecture, and the correlated components that comprise an EA. Chapter 5 presents the experimental setup conducted in our work, describing each service and the experimental environment. Following up is Chapter 6 showing the results obtained from the conducted experiences. Last but not least, chapter 7 concludes this intermediate report by describing the threads to validity and future work.

Chapter 2

Background

In this chapter, we provide the background on the fundamental concepts related to the work's main subjects, specifically, REST services and software testing. In Section 2.2, we describe the REST architectural and show a simple example of a REST service architecture and its components. Finally, in Section 2.1, we go over the main characteristics of software testing as well as methodologies like white-box and black-box testing.

2.1 Software Testing

As described by Myers et al. [22], software testing is a process, or a series of processes, designed to make sure computer code behaves as designed and that it does not do anything unintended. As so, software should be predictable and consistent, presenting no surprises to users. The purpose of software testing should be considered as a destructive process of attempting to locate program faults (i.e., the root cause of a problem, also known as a bug or a defect). Therefore an appropriate definition for testing application can be the following: "Testing is the process of executing a program with the intent of finding errors." [22]. It is important to keep in mind that software testing is not meant to prove the absence of code errors; rather, it is meant to establish their presence. **Black-box** testing (also known as datadriven or input/output-driven testing) and **white-box** (or logic-driven) testing are two of the most prevalent testing procedures, these two methods are categorized according to visibility [22]. It is important to keep in mind that there's another intermediate level known as **grey-box** testing [23], which is practiced by software testers less frequently.

The degree to which a testing activity accounts for the logic and internal structure of the system or component under test (SUT) is referred to as visibility in software testing. **White-box** testing refers to scenarios in which testers, who design test cases, have full visibility of the entity (i.e., code) under test and reflect that information in the test cases they develop [22] (e.g., by creating test cases that subject specific code paths of the functions under test to such test). In Figure 2.1, we demonstrate an example of white-box testing with a constructed control flow graph, which is used in the control flow testing technique as we further explain.

There are considerable **advantages** of white-box testing [24]. For instance, it reveals

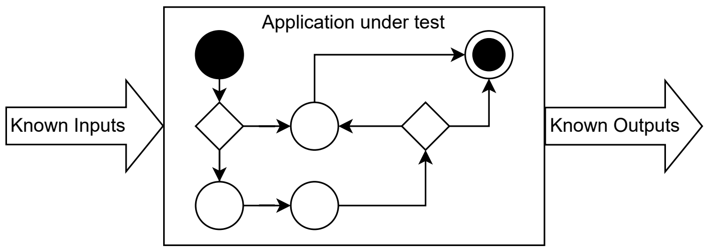


Figure 2.1: Example of white-box testing, Control Flow Testing technique

an error in hidden code by removing extra lines of code, side effects are beneficial, and maximum coverage is attained during test scenario writing. However, there are also **disadvantages**. It is costly as it requires a skilled tester to perform it, and many paths will remain untested as it is challenging to look into every nook and corner to find hidden errors. Moreover, some of the codes omitted in the code could be missed out.

Regarding white box testing techniques, some important types are briefly described below [24]:

- *Control Flow Testing*: It is a structural testing strategy that uses the program control flow as a model control flow and favours more but simpler paths over fewer but complicated path.
- *Branch Testing*: BT has the objective to test every option (true or false) on every control statement which also includes compound decision.
- *Basis Path Testing*: It allows the test case designer to produce a logical complexity measure of procedural design and then uses this measure as an approach for outlining a basic set of execution paths
- *Data Flow Testing*: In this type of testing the control flow graph is annoted with the information about how the program variables are define and used.
- *Loop Testing*: It exclusively focuses on the validity of loop construct.

The lack of visibility about the SUT, on the other hand, is referred to as **black-box** testing, in which the testers are uninformed of the internal structure. Test cases are mostly determined by the availability specifications and interface descriptions, as well as the tester's additional system expertise, about the context. The technique referred as Equivalence Partitioning, splits the value ranges of inputs to the SUT into logical classes (e.g., positive and negative numbers can be classified into two classes), ensuring a good coverage of the existing input value ranges provided in the system interface. Testers can choose at least one value from each of the partitioned input classes rather than selecting completely random inputs (which is not really ideal for large value ranges). The output of the SUT is then compared to the specification to verify that it is correct. To contextualize, in Figure 2.2, we show an example of the Boundary Value Analysis black-box testing technique. It targets the testing at boundaries or where the extreme boundary values are chosen. It includes minimum, maximum, just inside/outside boundaries, error values, and typical values.

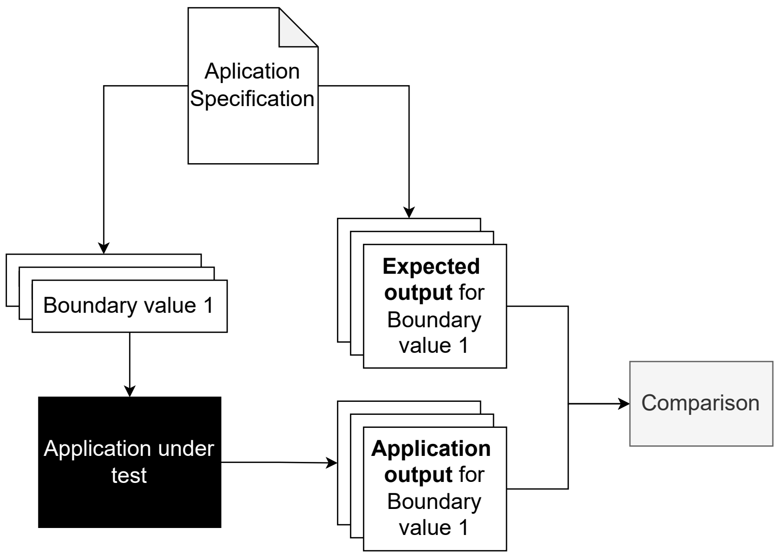

Aplication Specification
Boundary value 1
Expected output for Boundary value 1
Application under test
Application output for Boundary value 1
Comparison

Figure 2.2: Example of black-box testing, Boundary Value Analysis technique

The **advantages** of black-box testing are the following [25]: efficient for large code segments, tester perception is straightforward, user's perspective is separated from the developers perspective (programmer and tester are independent of each other), and quicker test case development. On the other hand, the **disadvantages** are that only a selected number of test scenarios are actually performed. As a result, there is only limited coverage. Moreover, without precise specifications, test cases are difficult to design.

Concerning types of black box testing techniques, the following are essential ones, which we pithily described [25]:

- *Equivalence Partitioning*: It can reduce the number of test cases by dividing the input data of a software unit into a partition of data from which test cases can be derived.
- *Boundary Value Analysis*: It focuses more on testing at boundaries or where the extreme boundary values are chosen. It includes minimum, maximum, just inside/outside boundaries, error values, and typical values.
- *Fuzzing*: Fuzz testing is used for finding implementation bugs, using malformed/semi-malformed data injection in an automated or semi-automated session.
- *Cause-Effect Graph*: It is a testing technique that begins by creating a graph and establishing the relation between the effect and its causes. Identity, negation, logic OR, and logic AND are the four elemental symbols that express the interdependency between cause and effect.
- *Orthogonal Array Testing*: OAT can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- *All Pair Testing*: In all pair testing techniques, test cases are designed to execute all possible discrete combinations of each pair of input parameters. Its main objective is to have a set of test cases that covers all the pairs.

- *State Transition Testing*: This type of testing helps test state machines and navigation of graphical user interfaces.

Grey-box testing [23], also referred to as gray box testing, is a software testing approach used to evaluate a software product or application while only knowing a portion of the application's underlying structure. Grey box testing looks for and uncovers defects caused by improper application usage or code structure. It is a software testing method that combines white-box testing and black-box testing. While the internal structure (i.e., code) is known in white-box testing and unknown in black-box testing, the internal structure is partially acknowledged in grey-box testing.

Grey-box testing has a few **advantages** as it provides combined benefits of white box and black box testing techniques [26]. For instance, the tester relies on interface definition and functional specification rather than source code. The tester can design excellent test scenarios. The test is done from the user's point of view rather than the designer's point of view. Moreover, it creates an intelligent test authoring and unbiased testing. Whereas, for its **disadvantages**, the test coverage is limited as access to source code is unavailable, and it is challenging to associate defect identification in distributed applications. Moreover, many program paths remain untested, and the tests can be redundant if the software designer has already conducted a test case.

To summarize and accordingly with Myers et al. [22], the following are important principles of testing:

- Testing is the process of executing a program with the goal of identifying errors.
- Testing is more effective when not performed by the developer(s).
- A good test case is one that has a high chance of detecting an error that has yet to be discovered.
- A successful test case is one that uncovers a previously unknown error.
- Testing success requires a thorough definition of both desired output and input.
- Testing success includes a thorough examination of test results.

2.1.1 Fuzz testing

The first mention of fuzzing was seen in the research in 1990 when Miller et al. [27] created a tool named "fuzz", in which random input strings were generated to perform their reliability tests. The interest in fuzzing has been growing during the past few years when its potential has been enhanced, and it is nevertheless a relatively new field of research. The initial idea of fuzz testing is to create semi-valid test data either by mutating valid data or generating data with specific rules [28]. Fuzzers that alter existing test cases to create new ones are called mutation-based fuzzers. In contrast, the fuzzers that create the test cases are called generation-based fuzzers. Both generation and mutation-based fuzzers have to fuzz the current test cases, either randomly or with given pre-defined rules [29].

Regarding the black-box **Fuzz testing** [30, 31], it is a software testing technique that consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion. The application is then checked for errors like crashes, failed in-built code assertions, or possible memory leaks. Fuzzers are typically used to evaluate programs that accept structured inputs. This structure separates valid input from invalid input and is provided, for example, in a file format. Effective fuzzers provide semi-valid inputs that are "valid enough" to trigger unexpected behaviors deeper in the program but are "invalid enough" to disclose cases that have not been appropriately addressed.

2.1.2 Robustness testing

The act of producing faulty functioning by triggering design or programming errors within a particular system (i.e., robustness failure) [32] is known as **robustness testing**, which is deeply linked to reliability testing. Limit situations (e.g., out-of-bounds values) or improper inputs are typically sent into a system's interface throughout robustness testing. The system's robustness is then assessed using the ratio between the number of test cases that disclose robustness faults and the total number of test cases performed. A system is said to be resilient if it retains normal operational behaviour as a result of external failures [33].

In other words, when the subject of robustness in software testing raises, it usually indicates that the system, whether it is already in operation or still being developed, is functioning normally. Enhancing dependability and identifying unforeseen circumstances through data that simulates extreme environmental conditions can support establishing whether a system is reliable enough to perform as expected. It is not about those idealized situations where everything works without a problem. To determine what the other tests are missing, we do robustness testing. The goal is to develop test environments for evaluating the robustness of software systems. Furthermore, testing robustness is more focused than dependability benchmarking.

Laranjeiro et al. [1] conducted a systematic review on Software Robustness Assessment in which the authors analyzed 145 papers. They reached the conclusion that the most popular techniques for robustness testing can be classified into two categories: fault injection, which was included in about three-quarters of the works, and model-based techniques in the other category. The authors also discovered fuzzing methods for assessing system properties related to robustness.

Looking at the types of fault used in the works, Laranjeiro et al. [1] identified that invalid inputs overwhelm the distribution, used in more than half of the works, and followed up by random and boundary inputs. In fewer occurrences, there were also bit-level faults, timing faults, MACD operations, and invalid outputs.

2.2 REST architecture

The REpresentational State Transfer (REST) architecture was introduced in 2000, by Thomas Fielding [3]. In simple words, the available resources of the services are exposed accordingly to the REST principles and the REST interfaces rely solely on Uniform

Resource Identifiers (URI) for resource identification and interaction, and usually on the Hypertext Transfer Protocol (HTTP) for message transfer communication. A REST service URI only provides the location and name of the resource, which serves as a unique resource identifier. These resources can then be manipulated with actions based on the semantics of the predefined HTTP verbs [4, 5], such as GET, POST, DELETE, and PUT, these verbs also known as methods are mapped to CRUD operations (Table 2.1) and are used to define the type of operation that should be performed. During the handling of an HTTP request, the API might need to read or write data from or to a database and communicate with other web services. A web service (i.e., REST or RESTful service) using REST should follow some specific guidelines [3, 4]. The architecture should be client-server by separating the user interface concerns from the data storage concerns, and communications between client and server should be stateless (i.e., the server do not store the client state between requests). Also, the responses must be defined as cacheable or non-cacheable (i.e., If the same request is processed multiple times and the response is equal every time, then the server can store it in the cache to fulfill future requests), for scalability purposes. The requests and responses must be self-descriptive, i.e., they should hold enough information to describe how to process them. In addition the server responses must provide links to related and available resources. Lastly, requesting a new resource puts the client in a new state, where server responses must provide links to related and available resources following (Hypermedia As The Engine Of Application State principle).

Table 2.1: CRUD operations to HTTP methods

CRUD operations	HTTP methods
CREATE	PUT/POST
READ	GET
UPDATE	PUT/PATCH
DELETE	DELETE

According to the HTTP methods, a resource can be retrieved using the method GET. PUT and POST methods should, on the other hand, be used to create a resource. There are, however, differences between the two. The PUT method is idempotent, which means that executing the same request several times will result in the same output (response) as if just one was performed, whereas the POST method will result in different outputs (responses)if multiple requests are made, each one equal to the next. Therefore, PUT should be used when the client decides or knows the URI for the new resource. Alternatively, the POST method should be used when the server assigns the URI. The PUT and PATCH methods are used to update a resource. The difference between these two lies in the fact that the PUT updates the entire resource, whereas PATCH only makes partial modifications. In other words, if a resource consists of two parameters with the names param1 and param2, then to update the value of param1 with the PUT method, it will be necessary to specify both values of the two parameters (param1 and param2). While for the method PATCH, it will only be necessary to specify the value of the parameter to be updated, in this case, param1. Finally, the DELETE method removes a resource. Over time, some methods were introduced, such as the PATCH itself, HEAD, CONNECT, and OPTIONS, but usually the main usage is given to the methods PUT, POST, GET, and DELETE.

Each pair of a URI and an HTTP method in a REST service is referred to as an operation.

Optionally, parameters and a payload may be required for an operation. HTTP headers or the URI itself can be used to define operation parameters. Path parameters are included in URI parameters (i.e., `/api/123` is a generic example as a result of `/api/<variable>`), and query string parameters (i.e., defined after the ? and at the end of the URI separated with a &, like so `/api/car?type=suv&color=red`). The operation's payload is sent in the HTTP request body, and most REST services adopt JSON objects as payloads, although alternative formats, such as XML, plain text, and even file formats, are frequently accepted as well.

```
openapi: '3.0.0'
info:
  title: Example of an OpenAPI
  description: API description
  version: 1.0.0
servers:
  - url: http://localhost:8080/Example/API
paths:
  /users:
    post:
      summary: Creates new user
      operationId: create_User
      requestBody:
        description: User information input in json
        content:
          application/json:
            schema:
              required:
                - firstName
                - lastName
                - password
              properties:
                firstName:
                  type: string
                  minLength: 1
                  maxLength: 20
                lastName:
                  type: string
                  minLength: 1
                  maxLength: 20
                password:
                  type: string
                  format: password
                  minLength: 8
                  maxLength: 50
      responses:
        '200':
          description: 'OK'
        '400':
          description: 'Malformed request, incorrect JSON structure'
```

Figure 2.3: An example of OpenAPI specification.

Every HTTP request is followed by a response, which contains a status code as well as the content (if any). The **status code** indicates whether a specific HTTP request has been successfully completed and categorizes the responses in five classes [34]:

1. Informational responses (100–199)
2. Successful responses (200–299)

3. Redirection messages (300–399)
4. Client error responses (400–499)
5. Server error responses (500–599)

The interface of a REST service (i.e., API) must be documented so that developers can design suitable client applications. API specifications are now available in a variety of formats, the most extensively used being the OpenAPI specification (previously known as Swagger) [6]. Figure 2.3 demonstrates a simple example of an OpenAPI specification file formatted in YAML [35]. Here, the paths field describes a list of resources available (i.e., API endpoints, operations, URIs), each corresponding to a set of HTTP methods that can be performed (e.g., the POST method is supported for the resource /users).

For each HTTP method, the format of input and output parameters are described in the parameters (i.e., properties for a request body) and responses fields (e.g., the input-parameter password should take a string value of format password; it is required, and should be sent in the request body). For example, for a 200 OK (i.e., success) response, one may define the HTTP response payload, whereas a different payload could be returned for an error response (e.g., 500 Internal Server Error). As a result, developers can verify that the service's implementation and specification agree on each response. Regarding the example of Figure 2.3, a user can be created with a first name, last name, and password. There are also constraints concerning the size of each one of these parameters (e.g., string max and min length) that will be sent in a JavaScript Object Notation body, also known as JSON.

In Figure 2.4 is represented a general architecture of a REST web service and how the communication between the client-server is made. An HTTP request initiated by a client application must at least include the URI and method for a valid resource in the server. The method is mapped with the operation performed on that resource by the server (e.g., GET for retrieving). Both parameters and payloads are optional and vary depending on the API operation. Generally, *REST frameworks* return an HTTP response with a generic error message when malformed HTTP requests are made (i.e., when they do not follow the HTTP specification [36]).

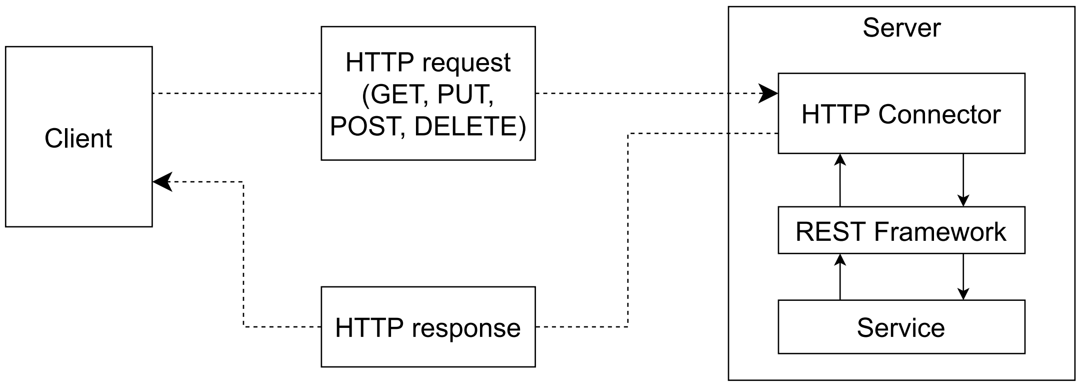


Figure 2.4: General architecture of a REST web service

In any case, if the HTTP request is well-formed, it will be passed to the REST framework component. As a result, the request's body will be parsed so that the server can determine which operation the user is requesting. If, for example, the provided URI does not match the URI of the server resource (e.g., sending a request with PUT method on a resource for which the server only supports PATCH) or there are security requirements that the client has not checked, the REST framework halt processing the request and notifies the client with the suitable error message.

If the targeted API operation requires parameters or a payload, this component will check their existence, extract them, and, in the case of media type-formatted payloads (e.g., a JSON object), cautiously convert the contents of the HTTP request body into the appropriate format. The server will again provide an error response to the client if the payload is faulty.

When the required API action is found, the Service component's corresponding function (i.e., the one which performs the API operation's logic) is invoked, and the HTTP request's inputs are supplied to it. Calls to external systems may be made by the service. The REST Framework returns the result of the called function's execution to the caller component, which may encapsulate it in a JSON object, for example. The required response elements are returned to the HTTP Connector, which encapsulates these in a suitable HTTP response object, serializes it into a sequence of bytes, and sends it to the calling client application over the Internet, completing one effective request-response communication process.

2.3 SOAP vs REST

In this section, we take a brief comparison between SOAP and REST services. We start by describing the main aspects of a SOAP service and then identify the main differences between SOAP and REST web services.

SOAP [37] is a uniform protocol and was first designed to allow communication between programs created in different programming languages and on multiple platforms. It has built-in restrictions that increase its complexity and overhead since it is a protocol, which causes longer page load times. However, these standards have built-in compliances that make them more advantageous in business circumstances. Security, atomicity, consistency, isolation, and durability (ACID), a collection of characteristics that guarantee trustworthy database transactions, are included in the built-in compliance requirements. Figure 2.5 demonstrates a SOAP-based Web services architecture.

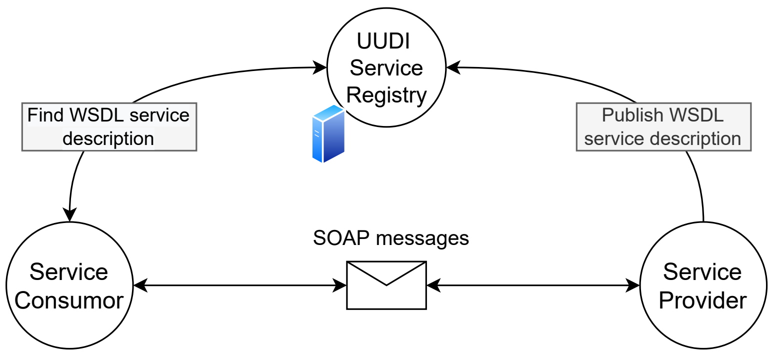


Figure 2.5: SOAP-based Web services architecture

SOAP relies on the WSDL [38], or Web Service Description Language, which is an XML-based definition language. It's used for describing the functionality of a SOAP-based web service. In other words, it describes the service interface (e.g., elements, attributes, data types). A WSDL file may be considered a contract between the provider and the consumer of such service.

Table 2.2: SOAP vs REST [3, 39, 40]

Property	**SOAP**	**REST**
Nature	It is a protocol and was designed with a specification. It includes a WSDL file that has the required information on what the web service does, along with the location of the web service.	It is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being: - Client Server - Stateless - Cacheable - Layered System - Uniform Interface
On API Changes	Client code must be recompiled with new WSDL	Can be backward compatible
Asynchronous	Yes, Asynchronous Messaging	Synchronous
Bandwidth Usage	+++	+
Cacheable	No	Yes
Data Formats	Only XML	XML, Json, Plain Text, etc
Documentation	The SOAP protocol can be quite complex but it is well documented.	Depends on the documentation provided by the service designer. (OpenAPI/Swagger for example)
Error Handling	Built-in	No error handling
Exposed Business Logic	Services Interfaces	URIs/Endpoints
Failure Handling	Retry logic built-in	Expects Client to retry
Goal	Focuses on exposing pieces of application logic (not data) as service.	Focuses on accessing named resources through a single consistent interface.
Invokation	Invokes Services by calling RPC Methods	Simply call services using HTTP Requests.
Java API	JAX-WS	Jax-RS
Javascript Support	Difficult	Easy
Network	Transfer over HTTP, SMTP, FTP etc.	Purely HTTP, however, REST is an architectural style and may use other protocols.
Official Standard	Yes	No
Statefullness	Supports stateless and stateful operations	Emphasizes stateless communication
Payload	XML Soap Envelop, must comply SOAP schema.	Can be any format
Payload Constraints	Must support XML Serialization.	-
Performance	Depends on Message Encryption, Signing etc.	Almost no protocol overhead.
Protocol	XML Based Message Protocol	A free architectural Style Protocol
Reliability	Reliable	Not Reliable. e.g., an HTTP Delete can return OK even if it did not work.
Security	Supports SSL, but also WS-Security (XML Encryption and Signature)	Depends on the documentation provided by the service designer.
Time to Market	Slow	Fast
Tooling	Requires significant middleware tooling support.	Only HTTP support is required.
Who uses it?	Financial, Payment Gateways, Telecommunication.	Social Media, Web, Mobile.

Unlike SOAP, REST services have a relatively loose architectural style where the presence of an interface description document (e.g., a WSDL document) is not mandatory. Consequently, this loosely coupled relationship between client and server does the practices used for robustness testing on SOAP services completely impractical for REST services. Additionally, this loosely coupled relationship between client and server allows for potential weaknesses in the service. Given the lack of formal description, REST services are easily exposed to invalid or malicious inputs that may activate residual faults in the code. Additionally, establishing guarantees about their robustness has become an integral element of the development process due to their increasing use and complexity. Therefore, REST services must be thoroughly tested to accomplish such guarantees.

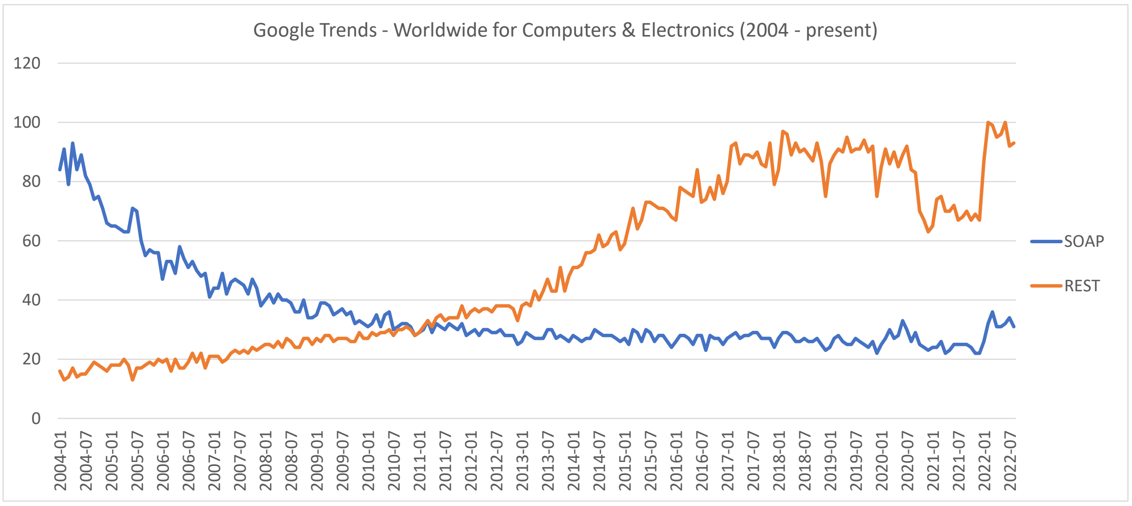


Figure 2.6: SOAP vs REST - Google Trends [41]

Even though we cannot have a straight comparison between the usage of REST and SOAP services, we can, however, look at each search topic across the years in Google trends by the category of Computers and Electronics. Figure 2.6 shows the evolution graph of these keywords searched since 2004. We can analyze from the start of 2011 that the REST services started to be the most searched therm between the two. Additionally, this evolution complies with the conclusion based on the empirical studies in the extensive survey conducted by Laranjeiro et al. [1], in which the amount of research on web services robustness evaluation peaked in the late 2000s where the preponderance focused on SOAP web services. However, the research interest has stopped, and the last work on SOAP services dates back to 2015.

Chapter 3

Related work on Evolutionary Algorithms and Software Testing

In this chapter, we analyze the related work on Evolutionary Algorithms and Software Testing, mainly focusing on RESTful API testing. In section 3.1, we begin by describing Evolutionary Algorithms. Furthermore, we deeply analyze Genetic Algorithms and the associated methodologies applied in their components, such as fitness function, mutation, crossover, and parent selection. Then, in sub-section 3.1.3, we discuss the empirical studies. We conclude this chapter with section 3.2, where we present the related work regarding software testing using evolutionary algorithms and approaches for RESTful API testing. Lastly, we finish with sub-section 3.2.2, where we conduct a discussion about the analyzed works.

3.1 Studies on Evolutionary algorithms

In artificial intelligence, an Evolutionary Algorithm is regarded as a part of evolutionary computation. As a result of solving problems, an evolutionary algorithm uses various evolutionary computational models inspired by the Darwinian principles of the natural process of evolution [42]. Here, the least fit individuals of the population are eliminated during the selection process used by evolutionary algorithms. In contrast, the most suitable individuals survive and evolve until better solutions are found. In other words, evolutionary algorithms are computer programs that simulate biological processes to address challenging problems. Successful individuals develop over time to offer the problem's optimal solution.

This analysis's primary objective is to identify the methodologies and values upheld in the literature so that we may establish a basis from which to start our implementation. These methodologies and values are correlated to the parameters that compose an Evolutionary algorithm, and we must first agree on a list of all its significant components. Accordingly, we assume the following components of an EA:

- Representation of individuals;
- Evaluation function (i.e., fitness function);

- Variation operators (i.e., mutation, crossover, mutation probability, crossover probability);
- Selection operator (i.e., parent selection or mating selection);
- Replacement operator (i.e., survival selection or environmental selection);
- Population (e.g., size, topology).

Table 3.1: Evolutionary Algorithms and techniques

Theme	Sub-theme	Papers
Algorithm	Simulated annealing	Kirkpatrick et al. [43], Metropolis et al. [44], Latiu et al. [45]
Evolutionary Algorithms	Evolutionary strategies	Hans-Georg Beyer and Hans-Paul Schwefel [46], Darrell Whitley [47]
	Particle Swarm Optimization	Kennedy and Eberhart [48], Latiu et al. [45], Eberhart and Yuhui Shi [49]
	Differential Evolution	Storn and Price [50], Thomsen [51]
	Genetic algorithm	Holland et al. [52]
	$1+1$ EA	Droste et al. [53]
	Memic algorithm	Moscato [54]
Genetic algorithm	Variation operators (i.e., mutation and crossover) and Parameters control	De Jong [55], Schaffer et al. [56], Grefenstette et al. [57], Goldberg et al. [58], Tuson and Ross [59], Spears [60], Srinivas and Patnaik [61], Eiben et al. [62], Michalewicz et al. [63], Homaifar et al. [64], Joines and Houck [65], Mühlenbein and Heinz [66], Smith and Fogarty [67], Hesser and Männer [68], Julstrom [69], Bäck [70–73], Lis [74]
	Parent selection	Baker, [75], Brindle [76], Goldberg and Deb [77], Razali and Geraghty [78], Cavicchio [79]
Genetic algorithm variations	messy GA's	Goldberg et al. [80]
	Genetic Simulated Annealing	Koakutsu et al. [81]
	Species conserving genetic algorithm	Li et al. [82]
Observation Studies	Effect of reduction the search space	Harman et al. [83]
	Comparison between GA, SA, GSA, SA/AAN	Xiao et al. [15]

In the following paragraphs, we will describe the components previously mentioned of the EAs, and we will also discuss different techniques to assign values to these parameters.

Furthermore, we deeply analyze genetic algorithms and the associated components, such as fitness function, mutation, crossover, and parent selection. Table 3.1 is a compilation of the analyzed studies. We aggregated them into categories such as the main evolutionary algorithms, different techniques for the genetic algorithm operators, some variations of genetic algorithms, and essential observation studies.

Even though Simulated annealing is a probabilistic technique and is not considered an Evolutionary algorithm, several studies compare the performance of Evolutionary Algorithms and Simulated Annealing. As a result, we chose to address it in the context of the related work. Kirkpatrick et al. [43] proposed the Simulated annealing as the basis of a search mechanism. It was inspired by the annealing procedure of the metal working [44]. SA is a stochastic global search optimization algorithm that is inspired by the slow cooling of metals, which is characterized by a gradual reduction in atomic movements, lowering the density of lattice defects until the lowest-energy state is achieved. Similarly, the simulated annealing algorithm generates a new potential solution (or neighbor) to the problem by modifying the current state according to predetermined criteria at each simulated annealing temperature. The new state is then accepted based on the satisfaction of the Metropolis criterion, and the process is repeated until convergence is achieved. This enables a worse solution to be accepted, allowing the system to escape from a local optimum and converge to the global minimum. Figure 3.1 illustrates a Simulated annealing flowchart.

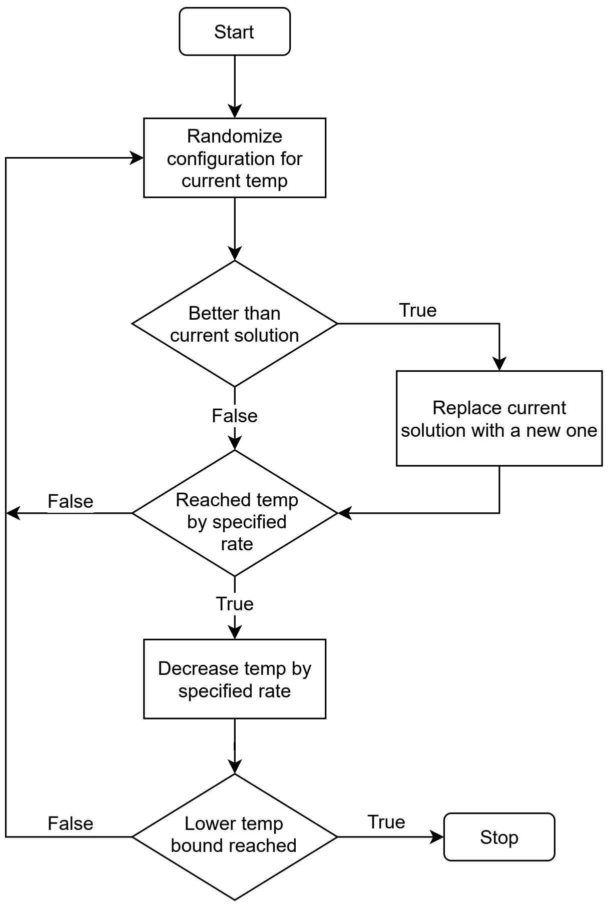

Start
Randomize configuration for current temp
Better than current solution
True
Replace current solution with a new one
False
Reached temp by specified rate
False
True
Decrease temp by specified rate
Lower temp bound reached
False
True
Stop

Figure 3.1: Simulated Annealing flowchart inspired by [45]

Evolutionary Strategies were developed by Ingo Rechenberg and Hans-Paul Schwefel [46, 47], in the late 1960's and 1970's. ESs are generally applied to real-valued optimization problem representations and emphasize mutation over recombination. Moreover, the many methods of manipulating parents and the offspring make up an additional distinct feature. Correspondingly, μ refers to the size of the parents' population and λ to the number of offspring that are produced in a single generation before applying selection. In this note, the following are the commonly used strategies for manipulation of the population [46]:

- $(\mu,\lambda)-$ Parents are replaced by the offspring. Here, the selection occurs among the λ offspring only, whereas their parents are replaced no matter how good or bad their fitness is compared to the new generation. This strategy relies on a birth overbalance (i.e., on $\lambda > \mu$) in a strictly Darwinian sense of natural selection.
- $(\mu+\lambda)-$ Offspring is added to the current population, and then to keep the population size constant, the λ worst out of all $(\mu+\lambda)$ individuals are discarded.
- $(\mu+1)$ - The parents generate a single offspring which only survives if it is a better solution than one of the parents.
- $(1+1)-$ A single parent generates a single offspring through mutation, and the best solution between the parent and the offspring becomes the new parent.

The first applications of ESs in experimental optimization were welcomed as innovative despite lacking proof of convergence toward an indisputable (global) optimum, and people were much more doubtful regarding numerical evolutionary optimization and its potential benefits [46]. The $(\mu+1)$-ES already uses not only of the so far best individual to generate an offspring but also of the second best and even the worst of μ parents. On the other hand, the $(\mu+\lambda)-$ES was welcomed as a further step into a wrong direction, since it does not make immediate use of new information gathered by the next offspring. Instead, it delays the selection decision until all λ descendants are born.

In the sense that even superior intermediate solutions can now be discarded and replaced by worse ones, the transition in the adoption from the plus to the comma version finally appeared to ascend to the top of absurdity. Additionally, it is easier to guarantee the convergence to a $(\mu+\lambda)-$ES since its worst behavior is premature stagnation, for instance, when the mutation strength becomes too low before reaching the optimum. In contrast, a comma version can diverge, especially when the mutation strength is too high.

Particle Swarm Optimization (PSO) is an evolutionary computation technique introduced by Kennedy and Eberhart [48]. Since it starts with a set of randomly initialized individuals (initial population) and uses a fitness value to assess each particle from the population, the PSO technique is identical to the genetic algorithm method. In PSO, a randomized velocity is given to each potential solution. PSO's particle tracking process keeps track of a particle's coordinates (location and velocity), which are associated to the particle's best solution so far. The particle swarm optimization technique alters the velocity of each particle at each step to propel it towards its best performance (pbest) and the overall best value reached by all particles in the population at each step (gbest). A random term w is also used to weight acceleration (weight inertia). For pbest and gbest [49], distinct random values are used to generate acceleration. The main steps of the particle swarm optimization method are illustrated in Figure 3.2.

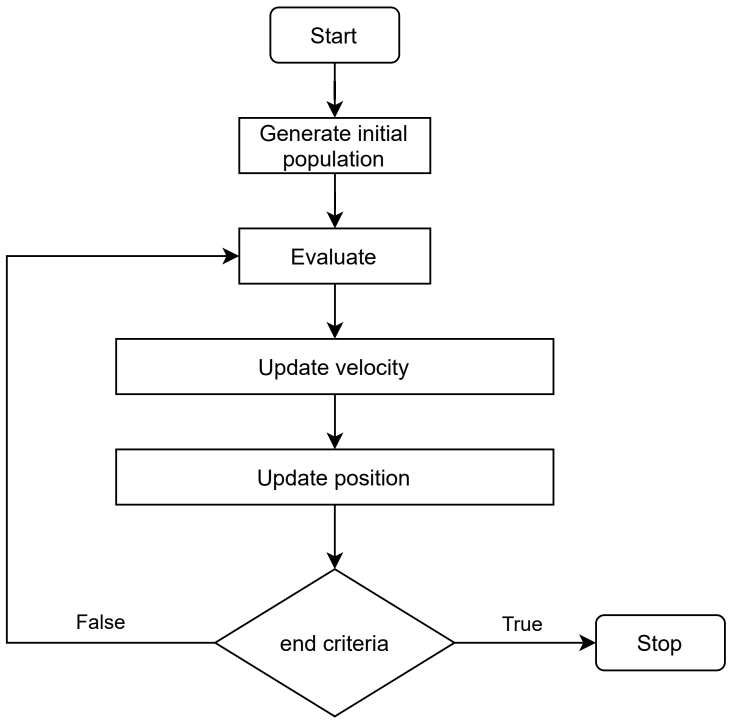


Figure 3.2: Particle Swarm Optimization flowchart inspired by [45]

Storn and Price [50] introduced the **Differential Evolution** (DE) algorithm. In this algorithm, parameter vectors or genomes are the individual trial solutions that make up a population. DE performs the same stages as a conventional EA in terms of computation. Unlike standard EAs, DE searches the fitness function landscape by using the difference of parameter vectors. DE, like other population-based search techniques, generates new points (trial solutions) that are perturbations of existing points; however, unlike Evolution Strategies techniques, these deviations are not samples from a predetermined probability density function. Instead, DE scales the difference of two randomly chosen population vectors to disrupt current generation vectors. DE creates a donor vector equivalent to each population vector by adding the scaled, random vector difference to a third randomly chosen population vector (also known as target vector). The components of the target and donor vectors are then combined to make a trial vector using a crossover operation. The trial (or offspring) vector competes against the population vector of the same index, i.e. the parent vector, in the selection stage. The survivors of all pair wise competitions become parents for the next generation in the evolutionary cycle when the last trial vector has been tested.

Droste et al. [53] proposed $(1+1)$ evolutionary algorithm (EA) which is a very simple algorithm. The so called $(1+1)$ Evolutionary Algorithm ($(1+1)$ EA), is applied to Boolean fitness functions $f : \{0,1\}^n \to \mathbb{R}$ (n being the bit string length) and can be formally described as follows, assuming that maximization of f is the objective:

1. Set $p_m := 1/n$.

2. Choose randomly an initial bit string $x \in \{0,1\}^n$.

3. Repeat the following mutation step: Compute x' by flipping independently each bit x_i with probability p_m. Replace x by x' if $f(x') \geq f(x)$.

An algorithm of this type is sometimes described as a randomized or stochastic hill-climber. It uses only one point in the search space and never accepts a new point with inferior function value, just like a hillclimber. Unlike normal hillclimbers, the $(1+1)$ EA has no clearly defined neighborhood, that is, it can reach any point in the search space in one step, while the probability of reaching a point decreases as the Hamming distance increases. The $(1+1)$ EA may be considered as a degenerate case of Simulated Annealing [43], where the cooling scheme is trivial, since the temperature is constant zero. Once again, however, the probability-based neighborhood is quite unusual.

3.1.1 Genetic algorithms

It is well known that local search techniques can suffer from the problem of becoming trapped in local optima. To overcome this problem many authors have considered global search techniques, most notably genetic algorithms [84–88], giving rise to the so-called evolutionary testing (ET) approach.

Genetic algorithms (GAS) are stochastic search techniques introduced by Holland et al. [52] in 1975. Genetic algorithms are loosely based on ideas from population genetics. First, a population of individuals is created randomly. Each individual can be viewed as a bit string and considered a potential solution to an issue of interest. Some individuals are better qualified to solve a problem than others in the population (e.g., better problem solvers). The selection of a new set of candidate solutions at the next time step, also known as the selection, is influenced by these differences. The selection procedure entails duplicating the most effective individuals while eliminating the less successful ones. The duplicates, however, are not exact. During the copy operation, there is a chance of mutation (random bit flips), crossover (exchange of related sub-strings between two individuals), or other alterations to the bit string. Mutation and crossover operations produce a new set of good individuals by transforming the existing ones into a new set. This new set of individuals have typically a higher chance of also being good than a previously existing set. Throughout this cycle of evaluation, selection, and genetic operations, the overall fitness of the population generally improves, and the individuals in the population represent improved solutions to whatever problem was posed in the fitness function. Figure 3.3 demonstrates an example of a Genetic algorithm flowchart.

Goldberg et al. [80] presented the messy GA's (mGA's) in 1989. This approach is intended for binary representations of fixed length, however it enables for representations to be under or over defined. Each gene has a value (a bit) and a place on the chromosome. The length of the chromosomes varies, and they may include too few or too many bits for the representation. If more than one gene specifies a bit location, the first one found is selected. On the other hand, if the chromosome does not specify bit positions, they are filled in using so-called competitive templates. Messy GAs avoid mutation and instead rely on cut and splice operators to replace crossover. A run of an mGA is in two phases: (i) a primordial phase which enriches the proportion of good building blocks and reduces

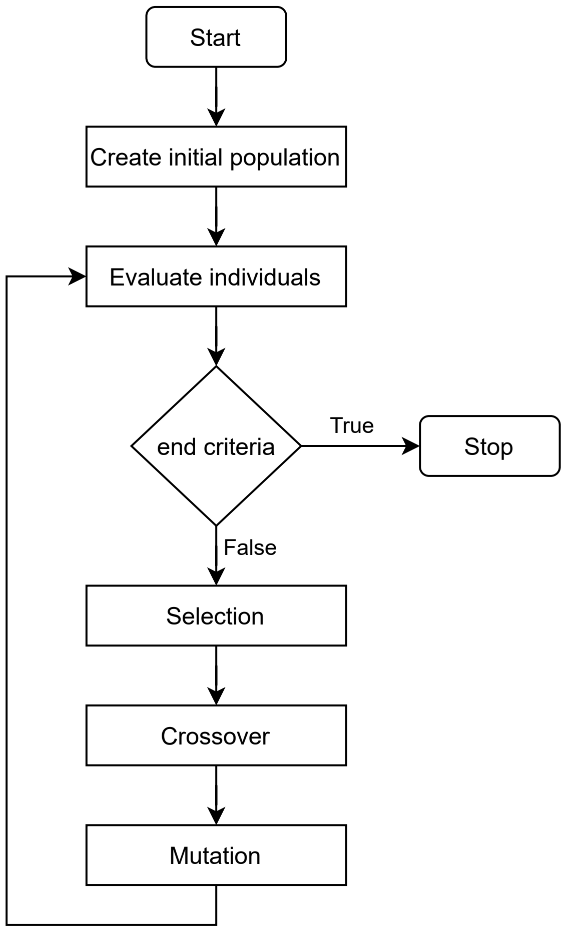


Figure 3.3: Genetic algorithm flowchart

the population size using only selection; (ii) a juxtapositional phase which uses all the reproduction operators. This method is intended for problems with misleading binary bitstrings. The algorithm adjusts its representation to fit a specific instance of the problem at hand. The first use of self-adaptive control was for the dominance mechanism of diploid chromosomes. Each chromosome is duplicated twice here. The extra chromosomes code for different solutions, and dominance determines which one is expressed.

3.1.2 Studies on parameter control of an EA

In this subsection, the main goal is to understand and enlighten the most remarkable works about the parameters that compose an EA. Moreover, which techniques are used to control the parameters of an EA, such as the mutation probability, crossover probability, parent selection, population size, and some approaches that automatically tune these parameters.

Several control parameters influence the performance of each optimization algorithm. Numerous papers have described approaches and methods for determining the appropriate control parameter values for a given algorithm. There is, however, no general formula for determining the values of control parameters. In many circumstances, we must modify the values to fit the problem and methodology. Consequently, we can classify the methods for changing the value of a parameter (i.e., the probability of mutation, the tournament size of selection, or the population size) of an evolutionary algorithm into one of these three categories [62]:

- *Deterministic Parameter Control*: This occurs when a deterministic rule changes the value of a strategy parameter. This rule makes deterministic adjustments to the strategy parameter without relying on search feedback. A time-varying schedule is commonly applied, which means that the rule will be activated when a certain number of generations have elapsed since the last time it was activated.
- *Adaptive Parameter Control*: This occurs when the search provides some kind of feedback that is used to determine the direction and/or magnitude of the change to the strategy parameter. Credit assignment may be involved in the assignment of the value of the strategy parameter, and the EA's action may affect whether the new value remains or propagates throughout the population.
- *Self-Adaptive Parameter Control*: The concept of evolution can be used to implement parameter self-adaptation. The modified parameters are encoded in the chromosomes and are subjected to mutation and recombination here. Better values of these encoded parameters result in better individuals, that are more likely to survive and generate offspring, propagating the better parameter values.

This terminology, introduced by Eiben et al. [62], leads to the taxonomy illustrated in Figure 3.4.

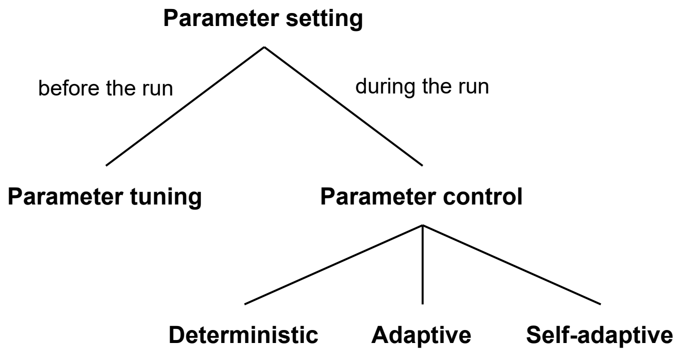


Figure 3.4: Taxonomy of parameter setting in EA's [62]

De Jong [55] concluded in 1975 that increasing population size reduced stochastic effects (of random sampling on a defined population) and improved long-term performance at the cost of a slower initial response. Off-line performance was shown to improve at the expense of on-line performance as the mutation rate was increased. Reducing the crossover rate significantly improved performance, implying that creating totally new individuals was a too high sampling rate. As a result of these experiments, a set of values for these parameters was discovered to produce generally satisfactory behavior for this class of problems, both online and offline. Therefore, these values have become part of the conventional wisdom on the topic. They are the following:

- population size: 50-100
- crossover rate: 0.60
- mutation rate: 0.001

De Jong also established off-line and on-line performance measures, with the assumption that off-line performance is based on monitoring the best solution in each generation, but on-line performance considers all solutions in the population.

Schaffer et al. [56] described an extensive experimental on the effect of changes in control parameters (mutation, crossover and population). The extensive analysis demonstrated that the pattern of the population-crossover-mutation interaction can be seen clearly. At low population size (10) good performance is very sensitive to mutation rate (m) and less so to crossover rate (c). It can be achieved with $m = 0.02$ and $c = 0.85$. As population size is increased, the sensitivity to mutation rate decreases and the best mutation rate to use also decreases (m = 0.002, 0.005 at p = 50). The inverse relationship between population and mutation most likely reflects the fact that increasing either one increases exploration and they can be traded off while keeping exploration at a somewhat constant level. The authors speculated that naive evolution (NE) (a GA using only selection and mutation) does perform a hillclimb-like search and given the range of strategies that can be achieved by varying population size and mutation rates, it is likely to be a powerful search algorithm, even without the assistance of crossover. Moreover, the use of Gray coding, which makes hillclimbing even more effective by eliminating the Hamming cliffs, contributes to this effect. However, they also did note, that NE appears to be a much stronger component in this experiment than suggested by the conventional wisdom. At least in the first generation (if random), a large population size can achieve a large sample of the space (exploration). However, a big population imposes a high cost every generation, and the operators can explore for schemata not present in the initial population. The success of rather high mutation rates and the reduced sensitivity to crossover rates suggests that our suite is still too simplistic to explore the influence of crossover appropriately.

Eiben et al. [62] proposed an approach to change the Mutation Step Size, assuming that Gaussian mutation is used together with arithmetical crossover to produce offspring for the next generation. The mean, which is frequently set to zero, and the standard deviation *sigma*, which can be understood as the mutation step size, are both required parameters for a Gaussian mutation operator. Mutations then are realized by replacing components of the vector $\vec{x}$ by $x'_i = x_i + N(0,\sigma)$ where $N(0,\sigma)$ is a random Gaussian number with mean zero and standard deviation σ. The simplest method to specify the mutation mechanism is to use the same σ for all vectors in the population, for all variables of each vector, and for the whole evolutionary process, for instance, $x'_i = x_i + N(0,1)$. Changing the mutation step size could be advantageous [72, 73, 89] and several options should be discussed one by one. To begin, the authors suggested to change the static argument σ to a dynamic parameter, such as a function $\sigma(t)$. This function can be defined using a heuristic rule that assigns different values based on the number of generations. The mutation step size, for example, could be defined as

$$\sigma(t) = 1 - 0.9 \cdot \frac{t}{T}$$

where t is the current generation number, which ranges from 0 to T, T being the maximum generation number. As the number of generations t approaches T, the mutation step size

$\sigma(t)$ will gradually decrease from one at the start of the run $(t = 0)$ to 0.1. Such decreases may aid the algorithm's fine-tuning capabilities. The value of the provided parameter changes according to a fully deterministic scheme in this approach. As a result, the user has complete control over the parameter, and its value at any given time t is completely predictable.

Grefenstette et al. [57] presented in 1986 a GA as a metaalgorithm to optimize values for the same parameters for both on-line and off-line performance of the algorithm. The most effective set of parameters for optimizing the GA's on-line (off-line are given in parenthesis) performance were:

- population size of 30 (80);
- probability of crossover equal to 0.95 (0.45);
- probability of mutation equal to 0.01 (0.01);

Fitness Function

The fitness function also called the evaluation function, assesses how closely a given solution adheres to the ideal solution to the desired problem. It establishes how fit a solution is to the problem at hand. Different problems require different fitness functions, and each case implements different approaches. Some techniques may optimize the search time of the genetic algorithm. The subsequent studies present a penalty practice to punish infeasible solutions that are known from the start, as invalids to the problem being solved.

Michalewicz et al. [63] in 1996, presented the **general principle used for the method based on penalty function**. They confirmed that the majority of constraint handling approaches are based on the concept of (external) penalty functions that penalize infeasible solutions, for instance, solving an unconstrained issue (on $\mathscr{S}$) using a modified fitness function:

$$\text{eval}\ (\mathbf{x}) = \begin{cases} f(\mathbf{x}), & \text{if} \quad \mathbf{x} \in \mathscr{F} \\ f(\mathbf{x}) + \text{ penalty } (\mathbf{x}), & \text{otherwise} \end{cases}$$

where penalty $(\mathbf{x})$ is zero if no violation occurs and is positive (for minimization problems) otherwise. Typically, the penalty function is based on a solution's distance from the feasible area $\mathscr{F}$ or the effort required to "fix" the solution (i.e., force it into $\mathscr{F}$). The first instance is the most common where several approaches apply a series of functions $f_j(1 \leq j \leq m)$ to calculate the penalty, with the function f_j measuring the violation of the jth constraint as follows:

$$f_j(\mathbf{x}) = \begin{cases} \max\{0, g_j(\mathbf{x})\}, & \text{if } 1 \leq j \leq q \\ |h_j(\mathbf{x})|, & \text{if } q+1 \leq j \leq m \end{cases}$$

In 1994 Homaifar et al. [64] proposed one of the previously referenced methods, the **method of static penalties**. It assumes that for every constraint we establish a family of intervals that determine the appropriate penalty coefficient.

- For each constraint, create several (l) levels of violation.

- For each level of violation and for each constraint, create a penalty coefficient R_{ij} $(i = 1,2,\ldots,l; j = 1,2,\ldots,n)$ where higher levels of violation require larger values of this coefficient.

- Start with a random population of individuals.

- Evolve the population and then evaluate individuals using the following formula:

$$\text{eval}(\mathbf{x}) = f(\mathbf{x}) + \sum_{j=1}^{n} R_{ij} g_j^2(X)$$

The weakness of the method is in the number of parameters. For n constraints the method requires $n(2l+1)$ parameters in total: n parameters to establish number of intervals for each constraint; l parameters for each constraint, defining the boundaries of the intervals (levels of violation); and l parameters for each constraint representing the penalty coefficients R_{ij}. For instance, for $n = 5$ constraints and $l = 4$ levels of violation, it is needed a set of 45 parameters.

The **method of dynamic penalties** was proposed by Joines and Houck [65] in 1994. Contrary to the previous method, the authors assumed dynamic penalties. Individuals are evaluated (at the iteration t) by the following formula:

$$\text{eval}(\mathbf{x}) = f(\mathbf{x}) + (C \times t)^{\alpha} \sum_{j=1}^{n} f_j^{\beta}(\mathbf{x})$$

where C, α, and β are constants. A reasonable choice for these parameters, presented by Joines and Houck, is $C = 0.5$, and $\alpha = \beta = 2$. The method requires a much smaller number (independent of the number of constraints) of parameters than the of static penalties method previously mentioned. Also, instead of defining several levels of violation, the pressure on infeasible solutions is increased due to the $(C \times t)^{\alpha}$ component of the penalty term: Toward the end of the process, for high values of the generation number t, this component assumes large values.

The penalty approach in the fitness function allows the core optimization issue to be changed using the penalty function approach into different formulations, and a series of unrestricted minimization problems are then solved to provide numerical solutions. There are, however, advantages and disadvantages of such methods:

- It is applicable to generally constrained problems with equality and inequality constraints.

- The starting point can be arbitrary.

- The method iterates through the infeasible region where the cost and/or constraint functions may be undefined.

- The final point might not be feasible and thus worthless if the iterative process ends prematurely.

Population

A population is a group of individuals or Chromosomes and each individual is a candidate solution to the problem. Figure 3.5 illustrates an example of the gene, chromosome, and population in the context of Evolutionary algorithms.

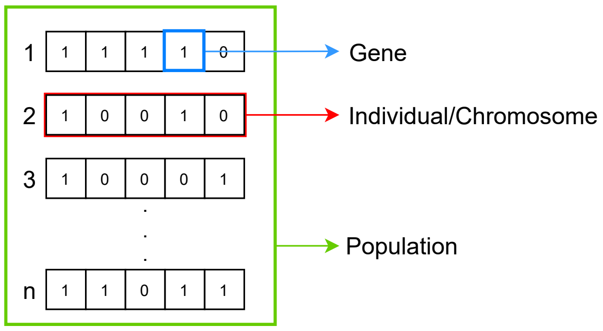


Figure 3.5: An example of the gene, chromosome, and population

Goldberg et al. [58] made effort to determinate which size a population must have for the best results. They stated that urgent use of variance-based population sizing in practical applications of genetic algorithms, as well as deeper foundational investigations, is recommended based on the results. They also proved that population size has a role in defining a distinction between two quite different sorts of simple genetic algorithm behavior. The authors witness the GAs at low population sizes, converging only by the graces of random changes that are lucky enough to survive long enough to be properly evaluated. Finally, they also saw GAs that promote just the best among competing building blocks at significant population sizes, and that when and if these are global, we can predict high probability convergence to global solutions after enough recombination. Understanding these two regimes is beneficial, as is having a quantitative yardstick to discern between high and low population sizes.

As mentioned before, De Jong [55] experimented with population sizes from 50–100, whereas Grefenstette [57] applied a meta-GA to control parameters of another GA (including populations size), the population size range was 30-80. Additional empirical effort was made by Schaffer et al. [56] and the recommended range for population size was 20-30.

Parent Selection

Parent selection is the process of selecting parents to produce the child, or offspring as it is also known, who will be a participant in the upcoming generation. Parent selection is especially crucial to the convergence rate of the GA as suitable parents drive individuals to better and fitter solutions.

Baker in [75] suggested a ranking selection algorithm for GA. The basic principle is that the population is sorted from best to worst, and then a new fitness value is assigned to each individual, which is inversely proportional to their rank. There are two methods, one being the linear ranking and the other exponential ranking. In **linear ranking**, the best individual gets a fitness s, between 1 and 2. Whereas the worst gets a fitness of $2-s$. Intermediate individuals' fitness values are given by interpolation, as described by Hancock [90]:

$$f(i) = s - \frac{(2i(s-1))}{(N-1)}, i = \{1..N\}$$

The worst string has no possibility of reproduction if s is set to 2. In theory, s may be increased beyond 2 to create larger selection pressures, but this would result in negative fitness values for several of the worst strings. In the **exponential ranking**, on the other hand, the best individual receives a fitness of 1. The second-best receives a fitness of s, which is typically around 0.99. The third best receives $s2$, and so on until the last receives s^{N-1}. Because the selection pressure is proportional to $1-s$, $s = 0.994$ yields a convergence rate double that of $s = 0.998$. Exponential ranking provides the worst individuals more opportunities at the expense of those who are above average.

Tournament selection is one of the different approaches to parent selection. One of the first forms of tournament selection was studied by Brindle in [76]. In this technique, several individuals are randomly selected from a population in tournament selection. For genetic processing (i.e. crossover and mutation), the best individual from the group is chosen. This can be done again and again until the mating pool is full. The tournaments are usually held between pairs of individuals (also known as binary tournaments), although a different number, n, can be used. The tournament selection gives a chance to all individuals to be selected and therefore preserves diversity. Note that high diversity may lead to a slower convergence speed. Despite this, the tournament selection has several advantages, namely efficient time complexity, low susceptibility to takeover by dominant individuals and there is no need for fitness scaling or sorting [77, 91].

Another parent selection technique is the **proportional roulette wheel**. Here, the individuals are chosen with a probability that is directly proportional to their fitness values. Those with the best fitness (larger segment sizes) have a higher chance of being selected. Within the roulette wheel, the fittest individual occupies the largest segment, while the least fit occupies a smaller segment. Every segment has a chance, with a probability proportional to the width of the segment. By repeating this process each time an individual must be selected, the fitter individuals will be chosen more frequently than the inferior ones, ensuring that the survival of the fittest requirements is met. Let $f_1, f_2, \ldots, f_\mathrm{n}$ be fitness values of individual 1 , $2, \ldots, n$. Then the selection probability, P_i for individual i is define as,

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

The major advantage of proportional roulette wheel selection is that it does not exclude any individuals from the population and allows all of them to be chosen. As a result, population diversity is preserved. However, there are a few key drawbacks in proportional roulette wheel selection. Outstanding individuals will add bias at the beginning of the search, which could lead to an early convergence and loss of diversity. If an initial population contains one or two very fit but not the best individuals, and the majority of the population is impoverished, these fit individuals will quickly dominate the population, preventing the population from exploring other promising candidates. Such a strong dominance results in a significant loss of genetic variety, which is detrimental to the optimization process. On the other hand, if individuals in a population have very identical fitness values, it will be extremely difficult for the population to improve since selection probabilities for fit and unfit individuals are relatively similar.

Razali and Geraghty in [78] compared three selection techniques for solving the traveling salesman problem. The comparison was made between the tournament selection, roulette wheel selection, and rank-based roulette wheel selection. The authors concluded that the quality of solution improved with rank-based roulette wheel selection. In comparison to the other two strategies, the GA-based tournament selection is more efficient in attaining the smallest total distance with the fewest number of generations and fastest iteration time. This, however, is only applicable to small problems. Tournament selection, as well as the proportional roulette wheel, becomes vulnerable to premature convergence as the size of the problem grows. Rank-based selection, on the other hand, keeps exploring the search area until it finds the shortest distance through the tour. As a result, tournament selection is better suited to small-scale problems, but a rank-based roulette wheel can be utilized to address larger-scale problems.

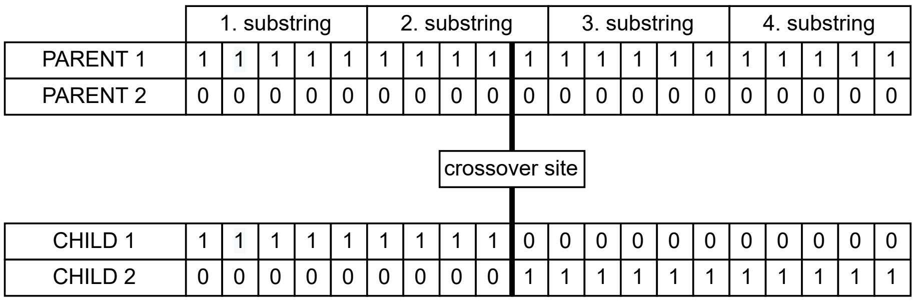
Cavicchio in his Ph.D. thesis in 1970, introduced different methods for genetic algorithms [79]. The **preselection** scheme was proposed in particular to maintain the diversity of the population. The children compete with their parents for survival in this scheme. In the next generation, if a child has a higher fitness (as evaluated by an objective function) than its parent, the parent is replaced by the child.

Goldberg and Deb in [77] stated that ranking and tournament selection are shown to maintain strong growth under normal conditions, while proportionate selection without scaling is shown to be less effective in keeping a steady pressure toward convergence.

Crossover

The crossing operation, also called recombination, is a genetic operator that combines the genetic information (i.e., chromosomes) of two parents to generate new offspring. There is relative success of commonly used crossover techniques in a GA based structural optimization. Below is a quick explanation of these methods.

The most straightforward crossover procedure is a *single-point crossover* (Fig. 3.6), in which paired individuals are each cut at a crossover site that is selected at random, and the sections that remain after the cuts are exchanged to create two new (child) individuals.



	1. substring					2. substring					3. substring					4. substring				
PARENT 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PARENT 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CHILD 1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
CHILD 2	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1

Figure 3.6: Single-point crossover implementation

In the *2-point crossover*, individuals are removed at two randomly chosen crossover locations. Swapping either the outer portions of the interior portions that fall between the locations accomplishes a design exchange (Fig. 3.7). Bare in mind that in both scenarios, the resulting individuals would be the same. Compared to single-point crossover, the 2-point crossover increases an individual's probability of swapping the essential genes on their chromosomal strings [92].

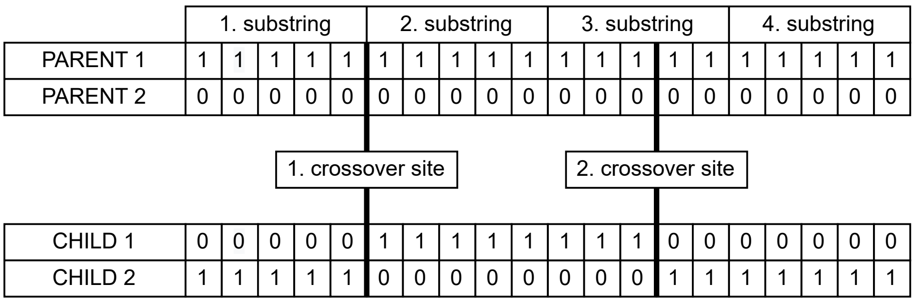


Figure 3.7: 2-point crossover implementation

Another technique is the *Multi-point crossover*, which seeks to provide a more dispersed exchange through an attempt to take a higher number of crossover sites. The implementation for Multi-point crossover, Fig. 3.8, is not so much different from single or two-point crossovers. Again, first a selected number of $(n_c \geqslant 3)$ crossover sites are randomly chosen, and individuals are cut at these sites to be separated into $(n_c + 1)$ portions. The procedure is concluded by completely exchanging one of the two groups of portions. The first group incorporates the set of $\{$1st, 3 rd $,\ldots,(2k-1)$ th, where $k = 1,2,\ldots,\text{int}(n_c/2+1)\}$ portions, and the portions that are not a part of the first group are included in the second group.

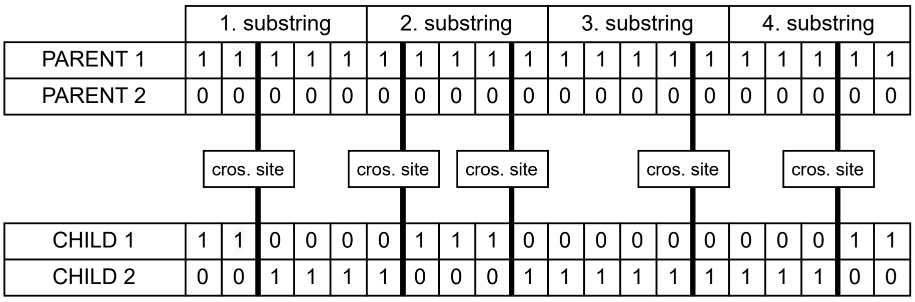
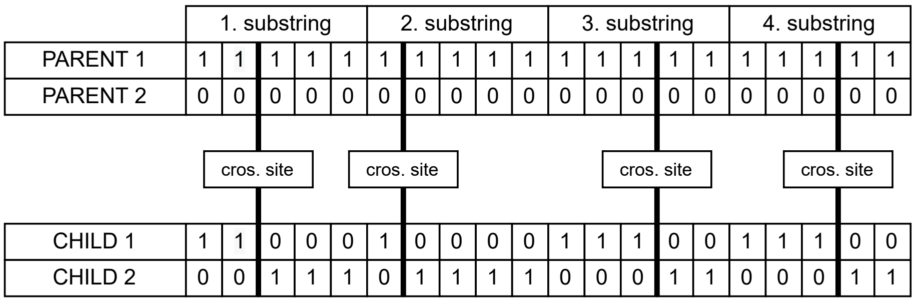


Figure 3.8: Multi-(5)-point crossover implementation

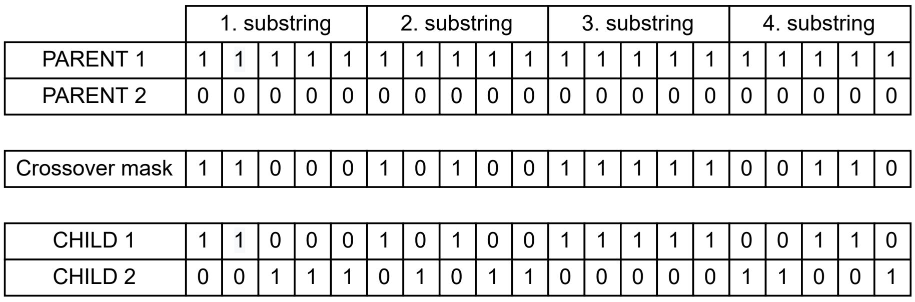
There is also the *Variable to variable crossover* method, where the paired individuals (strings) are first segmented into their substrings. A single-point crossover is performed on each of the substrings independently (Fig. 3.9). Therefore, each individual's design variable is turned on to fulfill the design exchange individually.



	1. substring					2. substring					3. substring					4. substring				
PARENT 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PARENT 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CHILD 1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	0	0
CHILD 2	0	0	1	1	1	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1

Figure 3.9: Variable to variable crossover implementation

A different approach is the *Uniform crossover* [93]. While compared to the methods used when taking crossover sites on individuals, the uniform crossover is radically different. A randomly generated crossover mask is necessary for uniform crossover (Fig. 3.10). According to this mask, a child's genes are inherited from their parents. At positions where the mask has a 1, the new child carries the genes from parent one, otherwise carries the genes from parent two at positions in which the mask holds a 0. We can either use the complementary of the first mask to generate the second child or create a new mask where we repeat the whole procedure. In this case, the numerical tests adhere to the latter strategy.



	1. substring					2. substring					3. substring					4. substring				
PARENT 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PARENT 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Crossover mask	1	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	1	1	0
CHILD 1	1	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	1	1	0
CHILD 2	0	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	1

Figure 3.10: Uniform crossover implementation

The crossover has a rate p_c that acts on a pair of chromosomes, giving the probability that the selected pair undergoes crossover. Some common settings for p_c obtained by tuning traditional GA's are $p_c = 0.6$ [55], $p_c = 0.95$ [57], and $p_c \in [0.75, 0.95]$ [56]. Currently, is well understood that the crossover rate should not be too low, and values of less than 0.6 are rarely adopted.

Tuson and Ross in [59] presented the approach with the name COBRA which explicitly collects information on operator performance (e.g. mutation and crossover) and uses this to adjust the operator probabilities. Normally, this adjustment is done with an ad hoc nature by the developer, as so COBRA is no exception. The main idea of the authors was to provide an operator adaptation mechanism that works in practice. The nature of COBRA is as follows: Given k operators $o_1, \ldots, o_k$, let $b_i(t)$ be the benefit, $c_i(t)$ the cost (the amount of computational effort to evaluate a child), and $p_i(t)$ the probability of a given operator, i at time t. They then apply the following algorithm:

1. The user decides on a set of initial probabilities p_i.
2. When a child is produced, $b_i(t)$ is updated.
3. After G (the gap between operator probability readjustments) evaluations, rank the operators according their values of b_i/c_i, and assign the operators their new probabilities according to their rank (i.e. the highest probability to the operator with the highest value of b_i/c_i).
4. Repeat step 2 every G evaluations.

The measure of benefit in this extensive study was operator productivity. Such operator is defined as the average gain in fitness when a child is born fitter than its parents (i.e., if the child is fitter than the parent, the productivity is zero) over a specific duration. The variables in the adaptation method are derived from two sources: the gap between operator probability readjustments G and the user-supplied initial operator probabilities. The authors observed that while there was no gain in performance when COBRA was applied, the GA was often made less sensitive to the operator probabilities provided when COBRA was used, reducing the effect of bad choices, which may be beneficial in some applications. In that case, obtaining equal performance may be easier than looking through a vast number of typical GA runs. They back up this claim by stating that COBRA's performance may be a technique that prioritizes speed over quality. Finally, COBRA looked promising as a way to eliminate some of the parameter tuning problems that surround GA applications. However, some problems may lead to a poor choice of operator probabilities, and it should be established that the choice of crossover probability is dependent on the problem to be solved.

Spears in [60] proposed a simple adaptive mechanism that allows the GA to choose between uniform and two-point crossover while the problem is being solved. Each individual got an extra bit, which defines whatever form of crossover is used for that individual. As a result, the offspring will inherit their parents' crossover type preference. The mechanism is simple and easy to implement. Also, it will work with almost any conceivable evolutionary algorithm (EA) style. The author states that 1bit adaptation generates good performance results, but much of the performance stems from simply having the two crossover operators at disposal. Consequently, this leads the author to think that it may often be beneficial for an EA to have a more extensive set of search operators that are customarily used.

Srinivas and Patnaik in [61] proposed an adaptive p_c (i.e., probability of crossover) and p_m (i.e., probability of mutation). To do so is essential to be able to identify whether the GA is converging to an optimum, therefore, the authors stated that one possible way of detecting convergence is to observe the average fitness value $\bar{f}$ of the population in relation to the maximum fitness value $f_{\max}$ of the population. $f_{\max} - \bar{f}$ is likely to be less for a population that has converged to an optimum solution than that for a population scattered in the solution space. The values of p_c and p_m are varied depending on the value of $f_{\max} - \bar{f}$. The value p_c should depend on the fitness values of both the parent solutions. The closer f is to $f_{\max}$, the smaller p_m should be, i.e., p_m should vary directly as $f_{\max} - f$. Similarly, p_c should vary directly as $f_{\max} - f'$, where f' is the larger of the fitness values of the solutions to be crossed. Srinivas and Patnaik reinforce that p_c and p_m are zero for the solution with the maximum fitness. Also $p_c = k_1$ for a solution with $f' = \bar{f}$, and $p_m = k_2$

for a solution with $f = \bar{f}$. For solutions with subaverage fitness values i.e., $f < \bar{f}, p_c$ and p_m might assume values larger than 1.0. To prevent the overshooting of p_c and p_m beyond 1.0, the following constraints were stated,

$$p_c = k_3, \quad f' \leq \bar{f}$$

and

$$p_m = k_4, \quad f \leq \bar{f}$$

where $k_3, k_4 \leq 1.0$. The final expressions for p_c and p_m are given as

$$\begin{aligned} &p_c = k_1 \left(f_{\max} - f'\right) / \left(f_{\max} - \bar{f}\right), \quad f' \geq \bar{f}, \\ &p_c = k_3, \quad f' < \bar{f} \end{aligned}$$

and

$$\begin{aligned} &p_m = k_2 \left(f_{\max} - f\right) / \left(f_{\max} - \bar{f}\right), \quad f \geq \bar{f} \\ &p_m = k_4, \quad f < \bar{f} \end{aligned}$$

where $k_1, k_2, k_3, k_4 \leq 1.0$. Finally, the authors have assigned the value 1.0 to k_1, 0.5 to k_2, 1.0 to k_3 and 0.5 to k_4.

The adaptive approach proposed by Srinivas and Patnaik allows low values of p_c, and p_m, to be assigned to high fitness solutions, while low fitness solutions have very high values of p_c, and p_m. As a result, each population's optimal solution is "protected" (i.e., not vulnerable to crossover) and encounters minimal mutation. On the other hand, any solutions that have a fitness value that is lower than the population's mean fitness value have $p_m = 0.5$. Consequently, all below-average solutions will be radically altered, and new ones will be produced. Therefore, the GA is unlikely to become trapped at a local optimum.

Mutation

The mutation can be defined as a slight random modification of the chromosome to obtain a new solution. It is used to maintain and introduce diversity in the genetic population and is generally applied with a low probability (i.e., p_m). Holland has introduced mutation to Genetic Algorithms as a "background operator" [52], which assures the principal possibility to recover from lost alleles, i.e. alleles which are converged within the population.

De Jong [55] recommended a mutation probability of $p_m = 0.001$, the meta-level GA used by Grefenstette [57] indicated $p_m = 0.01$, while Schaffer et al. [56] came up with $p_m \in [0.005, 0.01]$. Mühlenbein and Heinz [66] derived a formula for p_m which depends on the length of the bitstring (L), namely $p_m = 1/L$ should be a generally "optimal" static value for p_m. This rate was compared with several fixed rates by Smith and Fogarty [67] who found that $p_m = 1/L$ outperformed other values for p_m in their comparison. Bäck [70] also found $1/L$ to be a good value for p_m together with Gray coding.

Hesser and Männer [68] proposed a derived theoretically optimal schedules for deterministically changing p_m for the counting-ones function. They suggest

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \times \frac{\exp\left(\frac{-\gamma t}{2}\right)}{\lambda \sqrt{L}}$$

where α, β, γ are constants, λ is the population size, t is the time (i.e., generation counter), and L being the length of the bitstring.

The issue with deterministically changing p_m, however, is that a predetermined deterministic time schedule cannot account for the unique characteristics of various fitness functions because fresh exogenous inputs establish it. As a result, the schedule must be changed according to the fitness function, much like the temperature schedule tuning problem in Simulated Annealing.

On the other hand, with a different perspective, the function to control the decrease of p_m was proposed by Bäck and Schütz [94] and constrains $p_m(t)$ so that $p_m(0) = 0.5$ and $p_m(T) = 1/L$ if a maximum of T evaluations are used

$$p_m(t) = \left(2 + \frac{L-2}{T} \cdot t\right)^{-1}, \quad \text{if } 0 \leq t \leq T.$$

where L is the length of the bitstring and t is the time (i.e., generation counter).

Julstrom [69] presented a mechanism that adaptively adjusts the probabilities with which a steady-state genetic algorithm applies its operators. The ADOPP mechanism distributes operator probabilities proportionally to their recent contributions to chromosomal construction that are better than the population median or 90th percentile, adjusted so that no operator has a probability of zero. To develop an offspring, both operators are applied individually, and the algorithm maintains a tree of their recent contributions to new offspring and rewards them accordingly. With numerous variants of the adaptive mechanism, the algorithm's performance was no better than when operator probabilities were locked at plausible values. This raised fundamental problems about obtaining data from a GA's population and recent performance, as well as using that data during the execution of a GA. As a result, Julstrom stated that for this reason to develop an effective adaptive operator probability mechanism, these problems must be addressed.

Bäck [70, 71] self-adapts the mutation rate of a GA by adding a rate for the p_m, coded in bits, to every individual. This value is the rate, which is used to mutate the p_m itself. Then this new p_m is used to mutate the individuals' object variables. In other words, better p_m rates will produce better offspring, who will pass on their improvements to future generations, whereas bad p_m rates will die out.

Fogarty and Smith [67] used Bäck's idea [70, 71], implemented it on a steady-state GA, and added an implementation of the 1/5 success rule for mutation. There has been empirical evidence showing that there is an optimal "acceptance" ratio of approximately 1:5. In other words, for every five individuals created, one should be integrated into the population. This can be explained by considering that, while repeated mutation of a single individual corresponds to a form of local search, which has been shown to improve the performance of Genetic Algorithms [95], there is a trade-off between local and global search that affects both the convergence velocity and possibly the quality of the final solution. They also observed that gray coding performed significantly better than binary coding in the most complex, uncorrelated landscapes. Gray coding provides a much more uniform landscape for the system to learn mutation rates, and as the landscapes becomes less correlated, mutation becomes more significant in the search process.

Lis [74], in 1995, proposed a method in order to eliminate the necessity of determining the mutation probability in advance. Hence, the mutation probability value is decided along the course of algorithm generation. The developed method is based on the observation that, given the optimal mutation probability, some chromosomes' fitness function values are close to the current highest value of that function, while the remaining chromosomes achieve lower function values while searching for new solutions outside the local minimum. During the GA operation, a criterion represents the proportion ratio between both chromosome groups, and the mutation probability is increased or decreased based on its value. Such a method allows the GA to start with nearly any initial mutation probability and arrive at reasonable values of that probability after a few dozens of steps of algorithm performance. The fitness function values for a GA with many predetermined constant mutation probabilities and a GA with various mutation probabilities were statistically compared. This analysis of sample data indicated that a GA with variable mutation probability produces better outcomes than a GA with any predefined constant mutation probability.

Bäck in [72] concluded that a time-dependent variation in the mutation rate could help a Genetic Algorithm optimize in fewer iterations. The author observed that when the fitness function becomes multimodal, their observations indicate that the search for a mutation rate control different from a constant value $1/l$, where l denotes the bit string length, may be worthwhile to overcome local optima. Finally, for the case of a multimodal fitness function, the results reported may be interpreted as an explanation of the usefulness of a self-adaptation mechanism for mutation rates as described in [71] where a remarkable diversity of mutation rates exists in a population of individuals.

Multimodal optimization

The purpose of multimodal optimization is to find multiple global and local optima (rather than a single solution) for the same function, so that the user can better comprehend the different optimal solutions in the search space and apply them as and when needed, the current solution may be switched to another suitable one while still maintaining the optimal system performance. Since late 1970s, evolutionary optimization methods for locating multiple (global or local) optima have been developed. They are commonly referred to as "niching" methods. As part of a standard EA, niche-based methods can be incorporated to promote and maintain multiple stable subpopulations within a single population, with the aim of finding multiple global optimal or suboptimal solutions simultaneously. In Engelbrecht's book [96], niching algorithms are categorized based on the way the niches are located. Three categories can be identified:

- **Sequential niching** (or temporal niching) develops niches over time. Iteratively, the procedure finds a niche (or optimum) and removes all references to it from the search space. The removal of niche references frequently involves a change in the search space. The process of finding and removing niches continues until a convergence criteria is met, such as when no more niches can be located after a certain number of generations.

- **Parallel niching** locates all niches in parallel. Individuals dynamically self-organize, or speciate, on the locations of optima. Parallel niching algorithms must organize individuals in such a way that they keep their positions around optimal locations throughout time, in addition to locating niches. Such that, once a niche has been discovered, individuals should proceed to cluster around it.

- **Quasi-sequential niching** locates niches sequentially, but does not change the search space to remove the niche. Rather, the search for a new niche continues, while the niches that have already been discovered are improved and preserved in parallel.

Parallel Niching (PN) is based on a parallel hillclimbing technique, which is similar to a binary search technique. The hillclimbing method starts with a large step size and each population element hillclimb until it cannot climb any more (cannot improve). The step size is then divided into half, and the hillclimbing approach is again applied. This is repeated until the predefined smallest possible step size, ε, is used.

Beasley et al. [97] described the Sequential Niching (SN) method. This method is practically an extension of the iterating GAs that maintain the best solution of each run off-line. Every time SN finds a solution, it depresses the search space at all points that fall within a threshold radius, known as the niche radius [98], to avoid converging to the same optimum over and over. It is not too easy to determine the stopping criterion of SN, generally after finding out all desired peaks the iterations are terminated.

Mahfoud [99] concluded, after applying parallel Niching and Sequential Niching on various multimodal problems, that parallel hillclimbing works best for easier problems and reasonably well for problems with intermediate complexity. However, it fails for problems with high complexity. In comparison, SN is weak on easy problems and remains unable to tackle harder ones as well.

De Jong [55] in 1975, introduced the crowding technique to increase the chance of locating multiple optima. The crowding technique compares each child to a randomly selected subpopulation of cf members in the existing parent population (cf stands for crowding factor). Using a distance metric, the parent member most similar to the child is chosen. If the child is fitter than the parent member selected, then the child replaces the parent member. For multimodal optimization [51], Thomsen has also incorporated crowding techniques [55] into differential evolution (CrowdingDE). Thomsen used the crowding factor and Euclidean distance as the dissimilarity measure. The closer the distance, the more similar they are, and vice versa. Even though an intensive computation is required, differential evolution can be effectively transformed into an algorithm specialized for multimodal optimization.

Li et al. [82] introduced the species conserving genetic algorithm (SCGA). It is a technique for evolving parallel subpopulations for multimodal optimization. As a result of this algorithm, a set of species seeds can be bypassed during each generation and be saved into the subsequent generations, after which a population is divided into several species based on the dissimilarity measure. Species seeds are identified by selecting the most fit individuals from the population. After the identification of the species seeds, the population undergoes the usual genetic algorithm operations: selection, crossover, and mutation. In order to maintain the survival of less fit species, the seeds of the saved species are copied back into the population at the end of each generation. Species seeds are determined by sorting a population in decreasing fitness order. Once sorted, the algorithm selects the fittest individual as the first species seed and forms a region around it. If the next fittest individual is not located within a species region, it is selected as a species seed and another species region is created around it. Otherwise, it is not selected. All remaining individuals are checked against all existing species seeds using similar operations. Following the genetic operations, the algorithms need to identify which species each individual belongs to in order to copy the species seeds back to the population. The algorithm replaces the worst (lowest fitness) individual within a species with its seed. If no such individual exists within a species, the algorithm replaces the worst and unreplaced individual across the whole population. By preserving the fittest individuals for each species, the main goal is to preserve the population diversity.

Mahfoud after reviewing De Jong's [55] crowding factor technique indicated its inability to maintain more than two peaks of a multimodal objective function due to replacement errors that result from genetic drift. Additionally, Mahfoud [100–102] suggested *Deterministic crowding* with the objective of maintaining the diverse population, eliminating parameter requirements, reducing replacement error, as well as restoring selection pressure. At first, the algorithm randomly selects two parents from the current population, performs crossover and mutation to create two offspring, and then the offspring replace the nearest parent if they are more fit. In case of a tie, the parents take precedence. Therefore, Deterministic Crowding (DC) results in two sets of tournaments: parent 1 against child 1 and parent 2 against child 2; or parent 1 against child 2 and parent 2 against child 1. A set of tournaments that yields the closest competition is selected. The similarity is calculated by using preferably phenotypic distance.

Mengshoel [103] proposed a probabilistic crowding technique. Under the proposal, a probabilistic replacement rule allows individuals with higher fitness to win against individuals with lower fitness proportionally. As a result, a restorative pressure is permitted, preventing the extinction of niches with lower fitness levels. The algorithm employs a probabilistic replacement operator in addition to deterministic crowding. Evidently, in probabilistic crowding, two comparable individuals X and Y play in a probabilistic tournament, with the probability of X winning determined by: $p(X) = \frac{f(X)}{f(X)+f(Y)}$ where f is the fitness function.

Goldberg in his book [104], originally introduced the Sharing method, being the first attempt to deal directly with the locations and preservation of multiple solutions among all the niching techniques. The idea is to divide the population into separate subgroups based on how similar the individuals are. An individual must share its knowledge with others in the same niche. In heavily populated areas, fitness sharing affects the search space by reducing the payoff. It reduces each individual's fitness by a factor approximately proportional to the number of similar individuals in a population.

Goldberg and Wang in [105] proposed an alternative sharing scheme known as the **coevolutionary sharing**. As a result, it overcomes the limitations of fixed sharing schemes by allowing niche radius and location to be adapted to complex landscapes, as well as allowing for better distribution of solutions in problems with many poorly spaced optima. Coevolutionary sharing relies on the principle of monopolistic competition in economics, which uses two populations - a population of businessmen and a population of customers. In this case, the businessmen's locations represent niche locations and the customers' locations are analogous to solutions. Therefore, individuals in both populations strive to maximize their own individual interests, thereby developing suitably spaced niches comprising of the most fit individuals.

Harik [106] introduced the **Restricted Tournament Selection** (RTS). RTS is a modified tournament selection for multimodal optimization. In RTS, GAs may choose which individuals will be replaced to insert a pair of elements. As in deterministic crowding, RTS selects two parents from a population at random and produces two offspring by applying crossover and mutation operations. For each offspring, the algorithm then picks a random sample of w (window size similar to CF in Crowding) individuals from the population and finds the closest one to the offspring, by applying either an Euclidean similarity or Hamming (for binary coded variables) distance. The nearest member among the w individuals will compete with the offspring to determine who is fitter. Upon winning, the offspring can enter the population by replacing its opponent. This type of tournament prevents elements of the population from competing against those which are too dissimilar to them.

Yin [107] proposed a **clustering-based niching** scheme to help the formation of the niches and avoid the need for estimation of σ_{share} needed in sharing technique. The fitness is calculated based on the distance $d_{i,c}$ between the i th individual and its niche centroid, hence reducing significantly the time complexity. The formation of the niches is based on the adaptive Macqueen's K-means clustering algorithm. The best k individuals are chosen from a set number (k) of seed points in this algorithm. A few clusters are created from the seed points using a minimum allowable distance $d_{\min}$ between niche centroids. The remaining members of the population are then joined to these existing clusters or used to develop new clusters based on $d_{\min}$ and $d_{\max}$. These calculations are carried out in each generation. The final fitness of an individual is calculated using the following relation:

$$F_i = \frac{f_i}{n_c\left(1-\left(d_{i,c}/2d_{\max}\right)^{\alpha}\right)}$$

where n_c is the number of individuals in the niche containing the individual $i, d_{\max}$ is the maximum distance allowed between an individual and its niche centroid, and α is a constant.

Moscato [54] introduced the **Memetic algorithm** (MA) in 1989. MA tries to mimic cultural evolution and, as stated by the author, it is a marriage between a population-based global search and the heuristic local search made by each of the individuals. MA greatly improves the accuracy of EAs in locating the optimal solutions for function optimization problems, the reason being that concentrate on locating a promising area in the search space and then use different local search techniques to strengthen the search within that region. Given a representation of an optimization problem, a certain number of individuals are created. The state of these individuals can be randomly chosen or according to a certain initialization procedure. After that, each individual makes local search. The mechanism to do local search can be to reach a local optima or to improve (regarding the objective cost function) up to a predetermined level. After that, when the individual has reached a certain development, it interacts with the other members of the population. The interaction can be a competitive or a cooperative one. The competition can be similar to the one described by the author in the Competitive and Cooperative method [54] or can be similar to the selection processes of GA. The cooperative behaviour can be understood as the mechanisms of crossover in GA or other types of breeding that result in the creation of a new individual.

Vitela and Castanos [108] proposed the Sequential Niching Memetic Algorithm (SNMA). SNMA combines a gradient-based local search procedure with a derating function, as well as niching and clearing techniques. It penalizes individuals who linger in areas near previously found optima in order to promote the occupation in different niches in the function to be optimized. The SNMA technique requires the usage of a niche radius. However, unlike other algorithms were determining the actual value of this radius or the species distance is complex, the performance results are not significantly sensitive to the values of this parameter. This is a benefit in problems where the quantity and distribution of the optima are unknown.

In SNMA, first, we have to initialize the population of the sequence with randomly generated individuals. Here the total number of optimal solutions, local and global, is given by J_{Total} . A new generation is obtained after applying the genetic operators (i.e., evaluation, selection, reproduction, and mutation) to all members of the current population. Like the GAs, the population size is kept constant from generation to generation. Every individual in the population at each generation moves toward its nearest peak following a hillclimbing gradient-based algorithm. If, at some point, during this process, an individual leaves the pre-specified search space, then the corresponding variable takes the boundary value assigned.

It has been assumed that the population consists of M individuals and expects J_{Total} optimal solutions within the search space. Suppose J optimal solutions have already been located (with $J < J_{\text{Total}}$), the distances d_m from each individual in the population to their nearest optimal solution are determined. These distances, together with the niche radius R, assign a suitable fitness function to each individual in the population. The niche radius R is identified with the width of the inverted Gaussian function. Thus, the fitness value of an individual will be closer to zero as it will approach any of the previously found optima. The individuals in the population are now ordered according to their fitness value in decreasing order from $m = 1$ to M.

Then a roulette wheel selection is used according to a probability of survival in which clearing is introduced. This selection operator assigns larger survival probabilities to individuals with a more significant fitness value. Regardless, the probabilities assigned are not proportional to the fitness value of the individuals. Instead, they decrease linearly (except for clearing) following the order position. Because clearing the SNMA has the essential characteristic that it eliminates individuals lying within a niche radius from any previously located optima promoting the occupation of niches not yet found by the MA. Recombination is implemented through the parent-centric PBX crossover. The mutation is applied to all population members with probability P_m.

The performance of the SNMA is not highly sensitive to the selection of the niche radius R, a feature advantageous, especially when the number and distribution of the optima are unknown. Furthermore, an advantage of SNMA over other algorithms is that it does not need to maintain permanent populations around each optimal found, and it is only necessary to store the location of these peaks.

Observation Studies

The following studies are essential observations made in the literature where we can see the impact of some environmental aspects in the search algorithms and valuable comparisons between evolutionary algorithms.

Harman et al. [83] demonstrated the effect of this domain reduction in the search space, results were presented from the application of local and global search algorithms to real world examples. The main goal of this study was to provide evidence to support the claim that domain reduction has implications for practical search–based test data generation. The conclusions withdrawn were the following:

- There is no relationship between search space reduction and reduction in cost for random search.
- There is a significant improvement in cost reduction for both hill climbing and the genetic algorithm.
- The reduction in cost is more for the genetic algorithm than for hill climbing.
- There is no relationship between search space reduction and search effectiveness in terms of coverage for any of the search algorithms.

The SA and the GA are powerful optimization methods. However, both have limitations. Koakutsu et al. [81] discussed the characteristics of SA and GA. One of the essential features of SA is its stochastic hill climbing. In order to exhaustively search the solution space, SA introduces small random changes in the neighborhood and thus is computationally intensive. Furthermore, GA has crossover operations, which allow it to locate the global optimum in the large search space at a rough and rapid pace. However, it does not have a way to accommodate small changes in the solution space explicitly. In order to combine the good features of these two methods, a new method was proposed by Koakutsu et al. [81], named **Genetic Simulated Annealing** (GSA).

The GSA [81] combines the hill-climbing features of SA and the crossover operation from GA. GSA has three primary operations: SA-based local search, GA-based crossover operation, and population update. SA-based local search slightly changes the local search space while preserving the best-so-far local solution. When the search comes to a large flat area or the system is frozen, the GA-based crossover operation creates a big jump in the search space. GSA updates the population by replacing the worst solution. This replacement can be conducted in two ways: 1) The weakest solution in the population is replaced with the solution produced by the crossover. 2) At the end of the local SA-based search, the weakest solution is replaced with the local best-so-far solution in the local SA-based search.

Xiao et al. [15] reported experimental results of the effectiveness of five different optimization techniques over five different C/C++ programs. The experiments took a white-box approach, and the same fitness function is used for the same program under test by each heuristic test data generator. This means that the "fitness landscape" factor has no influence on the comparison process. Four optimization algorithms were used in the experiments, which were Genetic Algorithm (GA), Simulated Annealing (SA), Genetic Simulated Annealing (GSA) and Simulated Annealing with Advanced Adaptive Neighborhood (SA/AAN). For the purposes of comparison, a random test data generator was used. GA has the best overall performance, according to the results. In fact, the GA method consistently outperforms the competition. With the Time Shuttle, Perfect Number, and Rescue programs, GA provides full condition-decision coverage. However, the GA was unable to achieve complete coverage with the other two SUT, despite this, no other optimization technique was able to perform better. The GA has the ability to keep the beneficial gene transferred down from previous generations and pass it down to successive generations, resulting in high-quality test cases being generated quickly. GA and SA/AAN performed admirably in both input spaces, with average coverage levels of 85 percent and above. To obtain the coverage levels achieved by the GA and SA/AAN methods, the SA and GSA techniques required a lot more effort. With smaller input spaces, the SA and GSA approaches performed significantly better than with larger input spaces. In general, GSA did not perform well in the experiments, and it only slightly outperformed the Random test-data generator. However, the parameters applied in the GSA algorithm used in the experiments, were not optimally tuned, which could explain these results. With a basic program and a modest input space, the Random test data generator performs well. Nevertheless, on programs with a complex structure and a vast input space, it performs poorly and inefficiently. As a result, the SA, GSA, and Random techniques may not be suitable for industrial applications with huge input spaces.

3.1.3 Discussion

In this sub-section, we discuss our observations of the analyzed studies of Evolutionary Algorithms' components. We first focus on the probabilities of crossover and mutation and then highlight a few relevant aspects of the analyzed approaches.

The main emphasis in the presentation of the empirical studies was to understand which probability values and techniques were used for the variation operators (i.e., crossover and mutation) and which methods existed in the literature. Moreover, we also sought studies related to parent selection, fitness function, and population. Subsequently, We had the following questions:

- **RQ-1**: Which interval of values for the probability of crossover and mutation are considered in the literature?
- **RQ-2**: Which interval of values for the population size are considered in the literature?
- **RQ-3**: Should these values be static or dynamic across the number of generations in an Evolutionary Algorithm?
- **RQ-4**: Is there any standardized method to help tune these parameters?

In an attempt to answer **RQ-1**, we observed several studies dating from 1975 through 1990 reporting crossover probabilities between 0.6 and 0.95 [55–57]. Nowadays, it is well comprehended in the literature that the crossover rate should not be too low, and values smaller than 0.6 are seldom adopted. While for crossover probabilities, the values should not be too low, the opposite is the standard for the mutation probability. We identified the interval of values between 0.001 and 0.01. However, since different problems to be solved may behave differently in these static values, some renowned authors in the literature [66, 67, 70] identified that a $p_m = 1/L$, where L is the length of the bitstring, was an excellent mutation rate.

Looking at **RQ-2**, concerning population size, we identified in the papers an interval ranging from 20 up to 100 [55–57]. However, values higher than 60 appears to be too high.

Regarding **RQ-3**, Eiben et al. [62] concluded that a general drawback of the parameter tuning approach, regardless of how the parameters are tuned, is based on the observation that a run of an EA is an intrinsically dynamic, adaptive process. This concept is consequently contradicted by the adoption of inflexible parameters that do not vary their values. Furthermore, it is self-evident that different parameter values may be ideal at different stages of the evolutionary process [72, 73, 89, 109–111].

It may be advantageous to make big mutation steps in the early generations to explore the search space, and modest mutation changes in the subsequent generations to fine-tune the sub-optimal chromosomes. As a result, using static parameters can degrade algorithm performance on its own. Using parameters that can change over time is a reasonable way to overcome this problem, that is, by replacing each parameter by a function $p(t)$ where t represents the generation counter [68, 94]. However, choosing the best static parameters for a given problem can be complex, and the optimal solution can be influenced by a variety of other factors (e.g., such as the applied recombination operator, the selection mechanism). Therefore, creating an optimal function p may be even more challenging. The downside to the $p(t)$ approach is that the parameter values are adjusted deterministically by time t, regardless of how far along the problem is being solved (i.e., without taking into account the current state of the search). However, researchers have enhanced their evolutionary algorithms by applying such simple deterministic criteria (i.e., $p(t)$) to improve the quality of outcomes their algorithms produced while working on specific problems.

For **RQ-4** there are some exciting methodologies where the authors attempted to standardize the tuning of the Evolutionary Algorithm's parameters. The authors tried clever approaches, such as adding the mutation probability codded in the bits of every individual where the mutation rate self-adapts throughout the generations [70, 71]. Others developed sophisticated mechanisms that adaptively adjusts the crossover and mutation probabilities proportionally to their recent contributions to chromosomal construction [59, 69, 74].

To conclude, finding acceptable parameter values for an evolutionary algorithm is, as a result, a poorly structured, ill-defined, and challenging problem. Several studies tackle this problem with innovative approaches. Nevertheless, in most cases, the problem to be solved has particularities where these approaches will not outperform simple deterministic criteria. Regardless, EAs generally outperform other techniques on this type of problem.

3.2 Studies on Software testing

In this section, we examine software testing studies using genetic algorithms that stand out in their techniques, characteristics, and comparisons for software testing, particularly Search-Based Software Engineering. Then, in sub-section 3.2.1, we present the academic research contributions in different techniques for testing REST web services. We conclude this chapter with sub-section 3.2.2, where we discuss our main observations concerning the work presented and emphasize some limitations of the current state-of-the-art practices on RESTful API testing, namely in the evolutionary approaches.

Software testing can be modeled as an optimization problem, where one wants to maximize the code coverage and fault detection of the generated test suites. Then, once a fitness function is defined for a given testing problem, a search algorithm can be employed to explore the space of all possible solutions (test cases).

Buehler and Wegener [112] use evolutionary algorithms to test specification conformance of an early version of an automated vehicle parking system. This system seeks to automate the parking of a vehicle lengthwise into a parking spot by gathering and analyzing data from environmental sensors that detect nearby objects. Individuals in the search are just parking scenarios that specify the parameters of a parking place, including collision zones and the vehicle's initial position. With this information, the parking control unit is invoked, and a simulated parking maneuver is simulated. With a successful test being one that causes a collision, the objective function is simply the value of the smallest distance between the car and the collision area recorded during the simulation. In the experiment undertaken, roughly 900 scenarios were simulated, with more than 25 scenarios found guiding to collisions. After analyzing these scenarios, it was discovered that the controller had difficulties with scenarios where the parking space was some distance away, and the starting position was already near the collision area on one side. A vulnerability in the simulation environment was also revealed, where it was discovered that calculations concerning the position of the car were too inaccurate. Such vulnerability culminated in more simulated collisions with the collision site.

This work by Buehler and Wegener is interesting since we can interconnect the "successful test being one that causes a collision" in their work with ours being the generation of a valid request (i.e., a request that generates a response with status code 200) to the workload or an invalid request in the faultload to the REST service. Furthermore, the authors use "the value of the smallest distance between the car and the collision area recorded during the simulation in the objective function", similar to resembling the distance between two responses from the RESTful API (e.g., the levenshtein distance [113] between the content of two responses).

Harman and Jones in [14] call this new field of software engineering research "Search-Based Software Engineering". They argue that software engineering is ideal for the application of metaheuristic search techniques. They also note that the search-based technique must outperform the random technique in order to be qualified as worthy of even being considered a successful application. The random method, therefore, provides the lowest benchmark. If the metaheuristic method does not outperform the random method, it is likely because it is poorly implemented. They also expect to see dramatic growth in the field of search-based software engineering within the next few years. They list the likely application areas and the developments that the growing research capacity will provide.

Boden and Martino [114] used a GA to generate API tests. They concentrated on the operating system error treatment routines. The chromosome uses order-based [115] encoding to represent an API call or command invocation, followed by ordered parameter codes. Encoding is at the byte level, allowing simple associative-array decoding of call and parameter codes. The genetic operations of order-based crossover and mutation (byte, non-order based) were used. The cycle used to process a single GA population consists of the following steps: 1) On the GA-host system, each chromosome is used to derive an

API test case (the phenotype); 2) each API test is sent to a test system where the API calls or command invocations are executed; 3) The API test results are sent back to the GA-host system; 4) The GA then evaluates the results; 5) Chromosomes with higher fitness are proportionally selected for the next generation. Finnaly, the fitness function was a weighted sum of various factors of a test response with an attempt to assess the sequences of operating system calls.

Tracey et al. [16, 116] used genetic algorithms and simulated annealing to generate input data to test handling runtime error conditions in code. These runtime errors are exceptions in many languages, such as C++ and Java. These languages provide explicit exception-handling constructs so that exception-related code can be separated from the main logic of the program. The authors generate test data for triggering an exception subsequently for the exception handler's structural coverage. Seven basic programs without any more than 200 lines were used in the experiments. Metaheuristic techniques were discovered to generate test data for practically all exception conditions within the code and full branch coverage of exception handlers where they existed. An industrial experiment was also undertaken on an engine controller. Here, test data were generated, raising various exception conditions. However, it was found that these exceptions could not be raised in practice since input situations had been generated, which were not possible during the actual operation of the system.

Mansour and Salame [84] compared Evolutionary Testing, Hill Climbing, and Simulated Annealing for path coverage test data generation, revealing that Hill Climbing discovers test data faster than Evolutionary Testing and Simulated Annealing, while Evolutionary Testing and Simulated Annealing can cover more paths. Simulated Annealing outperforms Genetic Testing, according to the researchers. Hill Climbing, on the other hand, is only applied to programs with integer inputs, and the research is limited to eight functions with fewer than 86 lines of code.

Hunt in [117] used a GA for testing cruise control system software. In his work a GA chromosome represents the input and corresponding expected output. The fitness value is assigned, if the measured output differs from the expected output. The greater the difference, the higher the fitness value. The expected output is derived from the original software specification. Hunt states that software is often developed by a third party, and the tester only has the software, which he treats as a black-box and tests against the corresponding requirement specification. A GA chromosome must be able to represent all input values that the software can process, as well as the values that its single output can have. He claims that the chromosome must be able to represent both the valid and erroneous inputs. In his approach the GA is used as an aid for a human tester. The GA identifies failure scenarios, but it is up to the human tester to identify the faults that led to the failure.

Lin and Yeh [118] have also studied automatic test data generation by a GA for a chosen subpath. Their method uses a so-called "normalized extended Hamming distance" to guide the optimization process and to test the optimality of the candidate solutions. The fitness function, called Similarity, defines how similar the traversed path is to the target path, is used to choose the surviving test cases. Optimality here means that the test case (i.e. a particular input) forces the program to follow the given path of program statements when executed. They claim that a GA is able to significantly reduce the time required for automatic path testing.

Minohara and Tohma [119] developed a GA to estimate the parameters of a so-called "hyper-geometric distribution software reliability growth model" (HGDM), in which the number of errors increases as a function of time. A set of parameter values is represented by its GA chromosome. The fitness value is calculated by comparing observed and estimated test-and-debug data for errors. They tried to minimize the number of errors and their results suggest that the GA approach may be a more reliable method for obtaining the estimations of their problem.

Kasik and George [120] developed a GA to simulate software inputs in an unexpected, but not completely random, way. The GA is applied as a repeatable technique for creating user events that are used to drive standard automated test tools, allowing the system to imitate various forms of naive user behavior. The system seeks to replicate how a new user learns to use a program. The fitness value is determined by how much the chromosome directs the activities to resemble beginner behavior. A specific reward system based on observations has been developed to describe novice behavior.

Bingul et al. [121] apply a GA to test the war simulation software THUNDER with the black box method. They applied multiobjective optimization with the Pareto method, and define three different ways to assign fitness values. The THUNDER software can be viewed more like a two-player game in which blue represents the friendly side and red is the enemy side, and the problem itself as four main objectives for the different scenarios: (i) Minimize the territory that blue side losses; (ii) Minimize the blue side aircraft lost; (iii) Maximize the number of red side strategic targets killed; (iv) Maximize the number of red side armor killed. As previously mention this is a typical multiobjective optimization problem. They try to optimize software behavior, war strategies, and the running time. The authors claimed that the GA was able to provide optimal or near optimal solutions.

Last [122] used the fuzzy based extension of GA (FAexGA) approach for test case generation. Using mutated versions of the original program, the goal is to uncover a minimal number of test cases that are likely to reveal faults. Crossover probability varies according to the age intervals assigned during a lifetime in the FAexGA technique. Young and old individuals have a low crossover probability, whereas other age categories have a high crossover possibility. The crossover probability of very young offspring is low, allowing for exploration. On the other contrary, older offspring have a lower probability of crossover, and dying out would help avoid a local optimum or premature convergence. Middle-aged offspring, on the other hand, are usually used for crossover operations. The fuzzy logic controller (FLC) is used to calculate the probability of crossover, with state variables such as chromosome age and lifetime (parents). FLC's fuzzification interface includes variables that indicate an offspring's age. As a result, FLC assigns the values Young, Middle-age, or Old to each parent, using the concepts of fuzzy set theory [123]. The membership of each rule in the FLC rule base is determined by these values. The centre of gravity (COG) is a defuzzification method that calculates real values for crossover probability using the FLC's linguistic variables. Therefore, and accordingly to this technique, its main goal is on the exploration and exploitation of individuals.

3.2.1 Studies on RESTful API Testing

In this section, we analyze related studies on testing RESTful APIs, where we focus on approaches that rely on multiple techniques for testing. We divided the studies in two categories the evolutionary and non-evolutionary approaches. Also, our main concern was to understand if any of related studies target robustness testing of REST web services. In Table 3.2, we compiled these categories as way of resume the main findings in the literature. We wrap up this section with a few paragraphs highlighting the key trends we observed in the assessed methodologies.

Figure 3.11 demonstrates an example of a tool for testing the robustness of RESTful APIs. It generates valid and invalid requests according to the RESTful API specification. The figure is inspired by the tool presented in [8] and described later in this section.

The main takeaway of Figure 3.11 is the logic behind the tool. It starts by parsing the specification file, and then a Workload is generated with valid requests accordingly with the specification of the REST service. Next, after the valid requests are sent to the service, faults are injected to create invalid requests, and once again, they are sent to the service. Such an approach can be seen as fuzzing testing since it sends random valid requests and then invalid ones. The responses from the generated requests are stored in files for the tester to analyze a posterior.

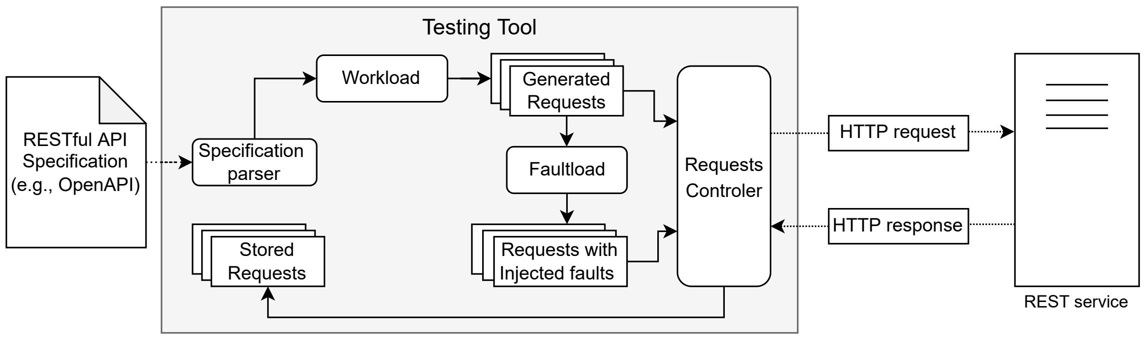


Figure 3.11: Example of a tool for testing RESTful APIs, inspired by [8]

Identification of studies

For the identification of the studies present in this related work, we used three well-known online libraries to search for primary studies, and they are the following:

- ACM Digital Library [124]
- Google scholar [125]
- IEEE Xplore [126]

Our initial choice of data source was Google Scholar since it is renowned for indexing a vast number of works. We conducted the search using the following query string, which was established based on early testing of different queries while using the relevant search engines of the three online libraries:

(((REST AND (services or API)) or ("RESTful API")) AND ((test OR testing) OR Fuzzing))

We had the objective of finding state-of-the-art tools for black-box testing of REST services. To achieve it, we needed to agglomerate *RESTful API*, *REST API*, and *REST services*, as the authors use these similar three keywords, which are the central theme of our targeted search. Also, initial observations indicate the use of fuzzing or testing by the writers with the absence of thermology Robustness, which was found only in one paper [8].

We also did recursive research by dissecting papers that were referenced by other key papers and authors. For instance, any paper of Andrea Arcuri [19, 127–132] had key references for related works. We would also recursively analyze the references of a new identified tool or approach. Furthermore, we took advantage of the survey: *RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions* [133] that helped identify new studies in the related area.

Studies

Table 3.2: Techniques for testing REST services

System	RESTful APIs	Arcuri [128, 129], Zhang et al. [127], Liu and Chen [134], Laranjeiro et al. [8], Viglianisi et al. [9], Atlidakis et al. [10], Martin-Lopez et al. [11], Karlsson et al. [12], Ed-douibi et al. [13], Segura et al. [135], Chakrabarti and Kumar [136], Chakrabarti and Rodriquez [137], Godefroid et al. [138], Fertig and Braun [139], Wu et al. [140]
Type of testing	Robustness	Laranjeiro et al. [8]
	Other	Arcuri [128, 129], Zhang et al. [127], Liu and Chen [134], Viglianisi et al. [9], Atlidakis et al. [10], Martin-Lopez et al. [11], Karlsson et al. [12], Ed-douibi et al. [13], Segura et al. [135], Fertig and Braun [139], Chakrabarti and Kumar [136], Chakrabarti and Rodriquez [137], Godefroid et al. [138], Wu et al. [140]
Evolutionary algorithms	GA	Arcuri [129], Liu and Chen [134]
	$(1+1)$ EA	Arcuri [128], Zhang et al. [127]
	None	Laranjeiro et al. [8], Viglianisi et al. [9], Atlidakis et al. [10], Martin-Lopez et al. [11], Karlsson et al. [12], Ed-douibi et al. [13], Segura et al. [135], Fertig and Braun [139], Chakrabarti and Kumar [136], Chakrabarti and Rodriquez [137], Godefroid et al. [138], Wu et al. [140]

Non-Evolutionary approaches

The bBOXRT tool was presented by Laranjeiro et al. [8]. It is a tool for black-box robustness testing of REST services. Its concept is divided in four steps. In the first step, the tool starts by parsing the basic information of the system under test. An interface description document (OpenAPI [6]) is read and analyzed. Information as the Uniform

Resource Identifier (URI), the available resources and the HTTP methods, input and output datatypes, error codes, and example requests are obtained to generate new requests. For the second step, a valid workload is generated randomly, according to the specification. This workload involves sending requests to the service so that the behavior of the service can be understood in the absence of any errors. The third step involves creating faulty requests by injecting a single fault into each request (e.g., an integer with its maximum value plus 1) present in the workload previously generated. Using the faulty requests, the service is triggered to act in an incorrect manner. Last but not least, in the fourth step, the responses to the services are stored to support the behavior analysis that follows.

Segura et al. [135] proposed an entirely different black-box approach, where the oracle is based on metamorphic relations among multiple requests (inputs) and responses (outputs). It operates by making small changes to the testing environment while keeping the inputs to system calls constant, then evaluating whether the results of these calls, specifically the metamorphic relation output patterns, fulfill specific requirements. For instance, they send two queries to the same REST API, where the second query has stricter conditions than the first one (e.g., by adding constraint). The result of the second query should be a proper subset of entries in the result of the first query. When the result is not a subset, the oracle reveals a defect. However, this approach only works for search-oriented APIs. Furthermore, this technique is only partially automatic since the user is supposed to identify the metamorphic relation to exploit manually and what input parameters to test. The approach was evaluated on the Youtube and Spotify REST APIs, and 11 issues were discovered in the services.

Viglianisi et al. [9] proposed a black-box tool, *RESTTESTGEN*, intended to automatically generate test cases for REST API. The tool uses the API Swagger specification to know which operations can be called and their input/output data format, to send well formed HTTP requests. The authors implemented in the tool an Operation Dependency Graph, mapping the dependencies between operations. Assuming there is a data dependency between two operations (n1 and n2), a common field in the output (response) of n1 and in the input (request) of n2, then the intuitive meaning of this dependency is that the first operation n1 should be tested before n2, because the output of n1 could be used to guess input values to test n2. Two fields (parameters) are assumed to be common when: (i) they are of atomic type (i.e., string or numeric) and they have the same name; (ii) they are of non-atomic type (i.e., structured) and they are associated to the same schema.

Chakrabarti and Kumar [136] propose a test framework, called Test-the-REST (TTR), used to execute test cases based on specific REST requirements. The tester writes a test case represented as an XML file with essential pieces of information, such as the HTTP method, the URI of the resource, and the expected representation, and is used as input to the test case validation module. Once validated, a test case is executed, and when the response is obtained from the target API, it is then used for verifying pass or fail conditions defined in the corresponding test case. This process is repeated for each test case that a tester provides to the tool. The results showed that the framework could detect a considerable amount of faults in the test RESTful service.

Later Chakrabarti and Rodriquez [137] presented a method of testing RESTful web services called connectedness. Such testing is based on every resource in the web service being reachable from the base resource by successive HTTP GET requests. The implemented method takes advantage of a formal web service specification to test its connectedness automatically. The specification is described in WADL++, an enhancement of the Web Application Description Language (WADL), which defines the graphs underlying the hierarchies between the available resources in the service. Consequently, only web service developers can supply such a specification file. Moreover, after creating the graph, a Depth-First Search is carried out by making a GET request on the base URI and extracting all the URIs that appear in the response payload. The process is repeated on all these URIs in a depth-first way and continues until no more unvisited URIs are being visited or a maximum search limit is reached. Then, by comparing the resulting URIs list of both the resource graph and the WADL++ description, a test verdict is conducted, and if there is a difference, then the web service is not fully connected. As an important note, specific requirements such as security (e.g., Basic authorization) are not entirely worked out in this approach, resulting in some resources being blockaded and, therefore, originating misleading results.

Atlidakis et al. [10] presented a stateful REST API fuzzer, with the name RESTler and written in python. The authors stated that RESTler was the first automatic stateful REST API fuzzing tool for test generating with the objective of finding security vulnerabilities. It starts by doing a static analysis of an OpenAPI specification (also know as Swagger) [6], and then generates and executes tests in a stateful manner. It generates requests by inferring dependencies among request types declared in the OpenAPI specification [6], and also by dynamically analysing responses to intelligently build request sequences in order to avoid requests combinations that can lead to future errors in the server. Furthermore, RESTler relies in a user-configurable dictionary to fuzz input values.

Martin-Lopez et al. [11] proposed an automated black-box testing tool for RESTful, with the name RESTest. The main feature of this tool is the automated analysis of interparameter dependencies, which enables for the automated generation of valid test cases using constraint solvers. In fact, certain REST APIs impose constraints that limit not only input values but also how input values can be combined to fill valid requests. The OpenAPI grammar as of now does not allow for formal documentation of these kind of dependencies. In this note, Martin-Lopez et al. [141] introduced a domain-specific language, called inter-parameter dependency language (IDL). RESTest relies in the so called IDL to map the inter-parameter dependencies of the SUT. Finally, RESTest can produce both nominal and faulty test cases using two strategies: random testing (RT) and constraint-based testing, by taking into account the constraints of inter-parameter dependencies.

Karlsson et al. [12] presented the QuickREST to generate input values accordingly to the API's specification. QuickREST follows a black-box approach to generate test cases for REST APIs automatically. The test inputs are generated using a two-fold mechanism: (i) randomly generated values that are agnostic to the specification, as well as (ii) values that are generated at random and comply with the parameter requirements in the OpenAPI document[6]. QuickREST uses a Clojure (functional programming language) variant with the name TestCheck [142], which provides functionality for defining data specifications and validating whether the given data conforms to those specifications. In addition, a key feature of QuickREST is its property-based testing (PBT), which involves generating input data and determining if it holds specific properties when exercised with that input (e.g., testing a *sort* function, the input should be an array sorted).

Wu et al. [140] presented RESTCT, a systematic and fully automatic approach using

Combinatorial Testing (CT) to test RESTful APIs. The approach first generates a constrained sequence covering array to determine the execution orders of available operations and then applies an adaptive strategy to create and refine several constrained covering arrays to concretize input parameters of each operation. The overall process of RESTCT is divided in two phases: (1) Operation Sequence Generation and (2) Input-Parameter Value Rendering.

In the *operation Sequence Generation*, the approach seeks to model the input space of available operations and construct a sequence covering an array as a representative set of operation sequences by identifying dependency relationships between the operations. Regarding the *Input-Parameter Value Rendering*, it models the input space of input-parameters of each operation, and sample representative value assignments via several covering arrays to produce concrete HTTP requests. The authors have four different techniques to generate input values for each input parameter identified in the operation. The first technique, which the authors call Dynamic, uses output values parsed from previous responses with the most similar names to the input parameter, assigning it as its value domain. The second technique uses values (i.e., enums or default values) from examples provided by the developers in the Swagger (i.e., OpenAPI) specification. The third one uses values from previous successful requests (i.e., returning HTTP status code of 200 range). Lastly, for the fourth technique, the values are generated randomly by respecting the domain of the parameter's data type when the other three methods cannot identify/-generate a new value.

With the above approaches to determine input parameters and infer constraints, RestCT will then utilize an adaptive strategy to generate concrete HTTP requests to execute operations in the given operation sequence.

Godefroid et al. [138] presented differential regression testing, which is a technique that finds regressions on REST APIs by comparing the behavior of different system versions against each other using the same inputs [143]. The approach considers regressions (i.e., breaking changes) in the API specification of the RESTful service and the software components of the service. The technique is applied to pairs of different versions to find regression bugs along two dimensions: when the service changes and when new clients are derived from the changed specification. Consequently, to detect a potential regression, the new version must produce an output different from the previous version. Moreover, the authors used RESTler [10], a stateful REST API fuzzer that automatically generates and executes sequences of HTTP requests defined in the API specification. The proposed technique can automatically detect deviations and highlight possible regression bugs based on the HTTP responses obtained during testing. Lastly, the authors assessed the approach across 17 different versions of the Microsoft Azure networking APIs from 2016 to 2019 [144], where five regressions were detected in the official API specifications and nine in the software components of the service.

Fertig and Braun [139] introduced an automated test case generation approach via Model-Driven testing of RESTful APIs. The approach not only automatically generates the source code for the API but also a large batch of functional and security test cases based only on an abstract RESTful API model that consists entirely of resources, states, and transitions. The authors designed the model for RESTful APIs using the Xtext framework and established the test cases on the supplied description. They were enabled to provide templates for the test cases by using Xtend, which is a Java dialect capable of implementing code generators within Xtext. The evaluation of a model-based software development-generated RESTful API was successful since the authors could produce over 20,000 test cases for only an API with four resources.

Ed-douibi et al. [13] proposed an approach to generate specification-based test cases for REST APIs. Their approach has as its goal ensuring that such APIs match the requirements defined in their specification and ensure a high coverage level for both nominal and fault-based test cases. It is divided in four steps. In the first step extracts the model by parsing the OpenAPI specification file [6]. Secondly, extends the previously created method by adding parameter examples which will be used as input data for the test cases. The third step, generates a TestSuite model by deducing the test case definitions for the API operations. Finally, the TestSuite model is converted into executable code in the last phase (e.g., JUnit). During the Transformation from OpenAPI to TestSuite (step 2 to step 3), two rules are defined, one generates nominal test case definitions given correct input data, and the second generates faulty test case definitions given incorrect input data (e.g., for an integer its maximum value plus 1).

Evolutionary approaches

Arcuri [129] proposed a technique to automatically collect white-box information from the running web services, and, then, exploit such information to generate test cases using an evolutionary algorithm. The approach was implemented in a tool called EvoMASTER [19]. The evolutionary algorithm used iteratively improves upon randomly generated test cases that aim to maximize code coverage and the amount of error status code responses from the service under test. A GA was used and each individual represents set of test cases, randomly initialized, with variable size and length. The fitness of a test suite is the aggregated fitness of all of its test cases. The crossover operator will mix test cases from two parent sets when new offspring are generated. The mutation operator will do small modifications on each test case (e.g., like increasing or decreasing a numeric variable by 1). They support all valid types in JSON (e.g., numbers, strings, dates, arrays and objects) and some of them need to be treated specially. For example, for date times, as genotype they consider an array of six bounded numeric values: year, month, day, hour, minute and seconds. They additionally consider valid values (e.g., minutes are from 0 to 59), but also some invalid ones (e.g., -1 minute) to check how the SUT behaves when handling time stamps with invalid format. When such date is used in a JSON variable, the phenotype will be a date string composed from those six integer values. When a test is executed, they check all targets it covers. If it covers a new target, the test will be copied from the test suite and added to an archive, to not lose it during the search (e.g., due to a mutation operation in the next generations). At the end of the search, all tests stored in the archive are collected, the redundant ones are removed, and the minimised suite is written to disk as a test class file.

Later on Arcuri presented the Many Independent Objective (MIO) algorithm [128]. MIO [128] combines the simplicity and effectiveness of $(1+1)$ EA [53] with a dynamic population, dynamic exploration/exploitation tradeoff, and feedback-directed target selection. It holds a test archive, with a different population of tests of size n (e.g., $n = 15$) for each testing target. As a result, given z targets, the archive can keep up to $n \times z$ tests at the same time. The archive will be empty at the start of the search, thus a new test will be generated at random. From the second stage on, MIO will choose whether to randomly sample a new test (probability P_r) or to copy and change (i.e., mutate) an existing test from the archive (probability $1 - P_r$). Finally, when a new test is sampled (i.e., mutated), its fitness is evaluated, and if necessary, it is stored in the archive. Based on the fitness value of a test, a duplicate of it may be saved in 0 or more of the z populations in the archive. Each target will have a heuristic score h in the range $[0,1]$, with 1 indicating that the target is covered and 0 indicating the worst possible heuristic value.

EvoMASTER also provides introductory support for black-box testing of REST services, essentially random generation with no support for data generators or inter-parameter dependencies [131]. However, the black-box configuration in EvoMaster performs significantly worse than the white-box strategy implemented in the tool [130].

After Arcuri presented the MIO algorithm [128], a more recent version was proposed by Zhang et al. [127]. The test case generation and optimization procedure in this version was improved by taking into account the semantics of HTTP methods used in REST services. The main idea was to define a set of templates (e.g., an operation mapped with GET method may be dependent of a POST method from another operation) that list meaningful combinations of actions on one resource based on the semantics of the HTTP methods. Therefore, the authors used these templates to sample new individuals, instead of sampling them completely at random. This approach can be seen as inter-dependability between operations of the API. When compared to the previous version of the MIO algorithm, the results showed an overall improvement in performance throughout all case studies, with increased code coverage and error response finding.

Liu and Chen [134] presented an approach to optimized Test Data Generation for RESTful Web Service. Their approach starts by reading and parsing an extended WADL specification of the API. This document describes the data type of the input parameters, and a restriction element must be added to this XML Schema to describe the constraint of certain input data types accurately (e.g., an integer with a minimum value of 0 and a maximum value of 20). Input data is then generated according to the data types and their constraints by equivalence partitioning with boundary value analysis. After the last step, the generated data is mutated according to each data type through one of a set of seven mutation operators. In order to reduce the cost of testing and select more effective test data, a genetic algorithm supported by a K-means clustering was used to evaluate the bug-detection capability of the mutants (i.e., data that suffered a mutation). The method was tested on a straightforward RESTful online shopping service system. However, the results did not reveal anything compelling.

3.2.2 Discussion

In this section, we present a discussion regarding our observations of the investigated methodologies for software testing with evolutionary algorithms and the testing of REST services.

It is argued that software engineering is the ideal scenario for the application of meta-heuristics, as the search-based approach has to outperform the random approach [14]. According to the extensive surveys [145, 146], GAs and their extensions are the most used search algorithms in search-based testing (SBT) literature (73 %), followed by more limited use of simulated annealing and its extensions (14 %) [146]. Furthermore, most papers (78 %) do not target any specific faults but focus on structural coverage of different test models. These algorithms, with slight adjustments to adapt to the task at hand, were used to handle several problems, including data test generation. Their frequent use also resides in the fact that there exist many publications on the application of GAs to various problems. Moreover, it is a strong indicator that such algorithms can be practical and achieve good results for the types of problems related to search-based testing. Consequently, substantial empirical data is available for the different parameter settings required by the GAs. This data dramatically helps choose appropriate parameters for a specific problem to be solved.

Regarding RESTful API testing, several methodologies were applied to carry out the testing of REST services. For instance, metamorphic relations between input and output in order to evaluate specific requirements, dependency graph to map dependencies between operations available in the REST service. The use of a functional programming language to allow the definition of specific dependencies between the properties of the parameters in a request. We also identified different types of testing, such as Differential Regression testing to compare the behavior of different system versions of a REST API, model driven testing, robustness testing, white-box and black-box testing.

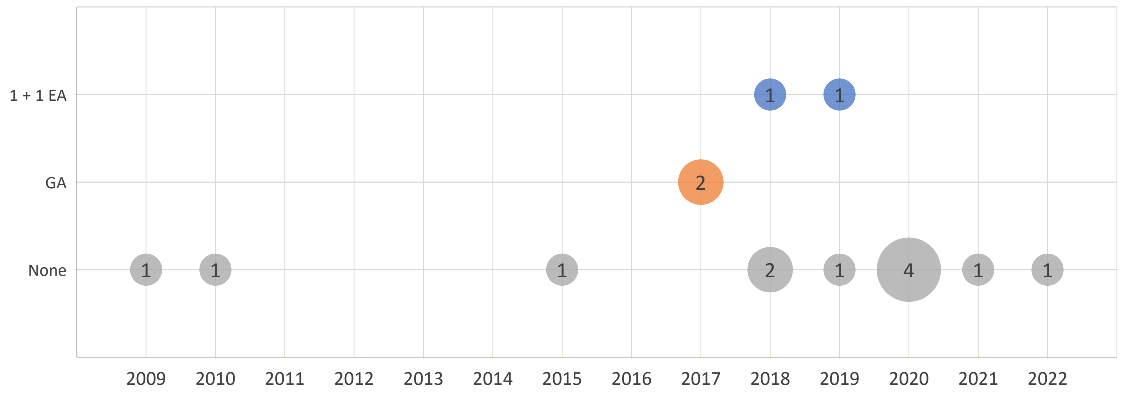


Figure 3.12: Distribution of REST API testing methodologies occurrences based on Evolutionary Algorithms over the years.

Although we can identify different techniques that were carried out in the literature, the same cannot be stated regarding evolutionary approaches for testing RESTful APIs. Looking at Figure 3.12, where we identified the number of occurrences of the studies concerning evolutionary approaches, we quickly recognize that Andrea Arcuri developed the most work with the EvoMaster project [19, 130, 147] (i.e., 3 out of 4 papers), which we identified as the current state-of-the-art tool for the testing of REST services, mostly focusing on white-box testing. However, it also has a black-box mode that takes advantage of its (1+1) EA used for unit testing.

Taking a deep dive into robustness testing, according to Laranjeiro et al. [1] and the conducted systematic review on Software Robustness Assessment in which the authors found that the work encountered on web applications is relatively scarce. The authors concluded at the time of writing the survey (2021) that Web services robustness evaluation has seen a peak of research being carried out in the late 2000s, with some work on Web Applications though with the majority focusing on SOAP web services. However, research interest has stopped, and they could also not identify robustness evaluation research of more recent web service implementations, such as REST services. Moreover, in the set of 145 papers analyzed for the extensive systematic review, only one paper proposed a tool by the name of Chizpurfle [148] for testing proprietary Android services with a fuzzing approach based on genetic algorithms. It also showed that services associated with more complex APIs benefit most from the evolutionary approach.

Even though SOAP services have been extensively tested for robustness, REST services have not been put under the same scrutiny [1] despite their wide range of applications. Consequently, to the best of our knowledge, there are no studies on black-box robustness testing for REST services using an evolutionary approach to generate robustness tests. However, a few automated tools for software testing of REST services have been developed, where most of them either use dictionaries or random inputs for data generation [8–13]. Nevertheless, several observations suggest that the random technique may not be appropriate for industrial applications with large input spaces [15, 16].

Chapter 4

Evolutionary Approach and Testing Tool Architecture

In this chapter, we describe our automated evolutionary approach for robustness testing of REST services. We begin by introducing, in section 4.1, a simple overview and the main concepts that support the approach, and section 4.2 describes the different parts and obligations of our *Evolutionary REST Fuzzer* approach's internal components and how they interact to support each other.

4.1 Approach Overview

Figure 4.1 summarizes the idea underlying our approach. The box delimits the scope of our architecture, having several components that interact with each other and with the external system under test (i.e., RESTful API). It is worth mentioning that the bBOXRT tool [8] is in our scope once we use some of its components to parse the interface's information. As a result, by leveraging information about the system's interface under testing (i.e., OpenAPI [6] description file), our approach, while taking advantage of the evolutionary algorithm, produces valid and invalid requests in an attempt to trigger faults in the REST service. The procedure is divided into the following steps:

- **Interface description analysis** - This first step reads and analyzes the basic information of the description file of the RESTful API (e.g., OpenAPI [6]) by parsing it. This information is then collected and used in the following steps, which include the Uniform Resource Identifier (URI) of available resources and the Hypertext Transfer Protocol (HTTP) methods they implement, input and output data types, error codes, and example requests. This step is performed by the bBOXRT tool, which can parse the RESTful API description file structured in the YAML format [35].

- **Generation of Workload** - In an attempt to generate valid requests (i.e., request that generates a response with status code 200) the Evolutionary Algorithm tries to generate **valid inputs**, sent as parameters in the requests, to analyze the behavior of the REST services. The generated inputs are produced by the EA components (e.g., evaluation function, variation operators, selection operator, population and the representation of individuals).

- **Generation of Faultload** - Similar to generating a Workload, the goal is to generate invalid inputs to analyze the REST service behavior when confronted with faulty inputs that may trigger erroneous behavior (robustness problems). These inputs are also based on the EA helped by a dictionary of pre-defined faults (e.g., replace a parameter by *null*) along with incorrect values for the specific data types (e.g., an integer's maximum value plus 1).

- **Results storage** - The last step takes care of storing every data retrieved regarding the test process to support further behavior analysis of the SUT by the tester. For the workload and faultload phases, the requests and responses are held in *xlsx* files (i.e., Excel). Additionally, the same raw information and the tool messages concerning the testing process are stored in a *txt* file for extensive debugging. Lastly, when code coverage is enabled, the code coverage report is gathered from the SUT and stored for further analysis.

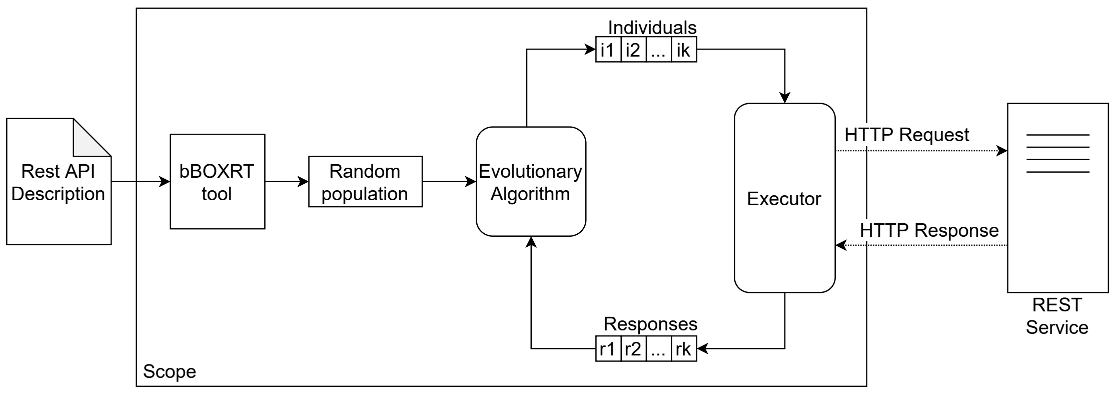


Figure 4.1: Overview of the proposed approach

The following section clarifies these steps in further detail and maps them to the different software elements that comprise our approach, *EvoReFuzz*.

4.2 EvoReFuzz: Evolutionary Approach for REST Testing

The architecture of our approach is depicted in Figure 4.2, where the *EvoReFuzz* components are contained in a rectangle, and the elements of the evolutionary algorithm itself are sub-contained in the dashed rectangle. These interact between each other and with external entities (e.g., REST services).

The *bBOXRT tool* [8] is responsible for parsing an OpenAPI document structured in the YAML format [35], which describes the interface of a given RESTful API service. The

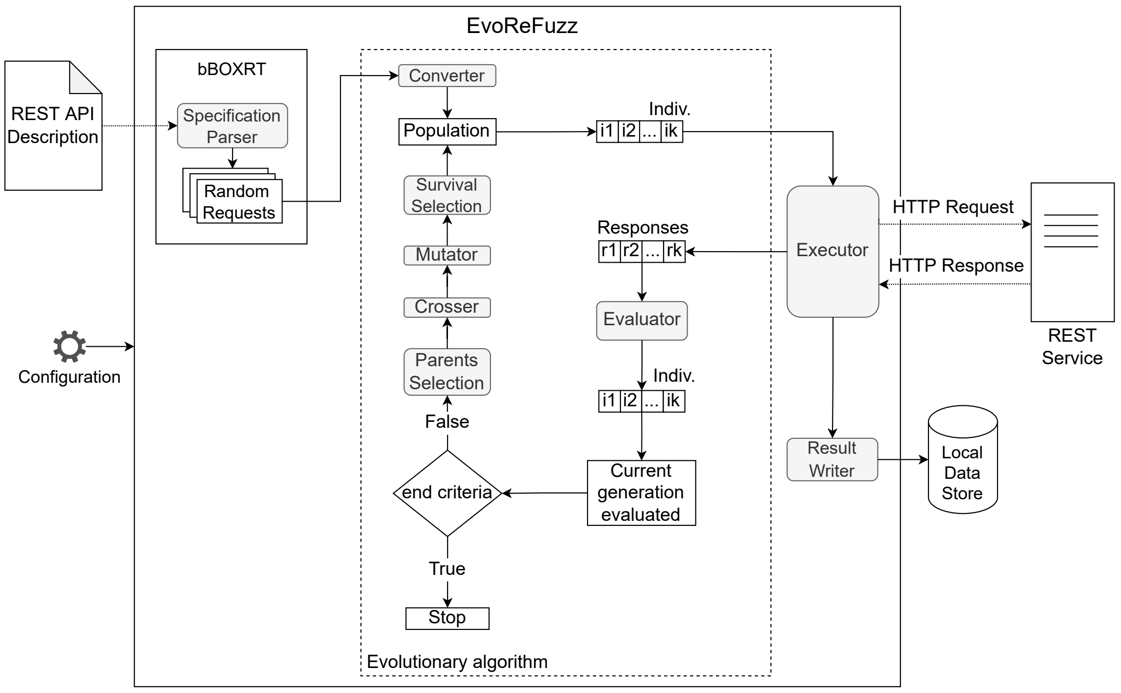


Figure 4.2: Evolutionary approach architecture with a genetic algorithm as the Evolutionary algorithm

basic information parsed is then translated to Java classes and used to generate a set of random requests. We use the *bBOXRT tool*, therefore, to support the **Interface description analysis**.

A RESTful API may have multiple operations (i.e., pair of a URI and an HTTP method) that may also require parameters and a payload (i.e., parameters located in the HTTP request body, such as in a JSON object). Each request is, therefore, composed of a URI, an HTTP method, and parameters. Additionally, there are different data types of parameters and possible locations for the parameters' values. Table 4.1 present some examples.

Table 4.1: Examples of values for the different data types

Type	Example of a value
Array	[1,2], [{"name":"Carlos"}, {"name": "Coimbra"}]
Boolean	true, false
Byte	U3dByb2Nrcw==
Date	2020-07-19, 2019-07-20
DateTime	2019-07-20T17:20:19Z
Double	1.7976931348623157E+308
Float	19.20
Integer	2147483647
Long	9223372036854775807
String	"eSGIAhrNOd", "MHNk5a5Ui0", "deiuc"
Object	{"car":{"color":"black"}}, {"ret": [{"area":20}, {"area":10}] }

The parameters' possible locations can be in the HTTP headers, the endpoint URI itself (i.e., named a *path* parameter), the HTTP query string, or the request payload (i.e., generally a JSON object). For instance, `GET http://localhost:8080/api/123` is a generic example as a result of `/api/<variable>`, where the `variable` is the operation parameter located in the *path* of the URI, and `GET` is the HTTP method.

Moreover, considering the possible parameters' data types and location for an Evolutionary Algorithm in our problem, we need to establish the representation of the subsequent individuals (i.e., chromosomes), their genes, and the population. In Figure 4.3, we demonstrate a simple example of a RESTful API with *n* operations. In this example, Operation 1 returns a list of elements with basic information about a car, where a parameter with the name `listSize` defines the size of the returned list. Accordingly, for each operation, one population is associated with it, representing multiple individuals (i.e., requests).

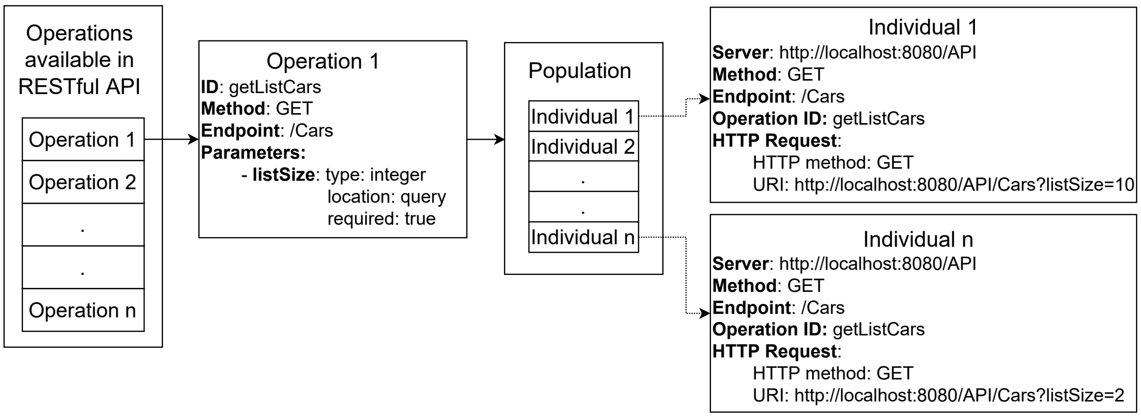


Figure 4.3: Representation of Population and Individuals in our approach

Looking at the population itself in Figure 4.4, our individuals (i.e., chromosomes) are the requests that will be sent to the service. Hence, one individual may have zero or more parameters (i.e., genes), that will be modified by the mutation and crossover that will occur in these genes' values (i.e., alleles), as we will explain further.

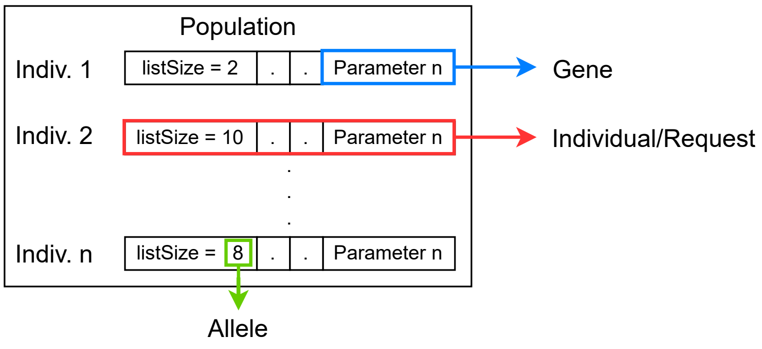


Figure 4.4: Context of Population, Individuals, Genes, and Alleles in our approach

The main focus of our approach is the ***Evolutionary Algorithm*** and its elements. The comprehensive tool has two phases, *Workload* and *Faultload*, which the user may decide to perform both or one at a time. For each mode, we used a ***Genetic Algorithm*** capable of generating requests accordingly to the objective of that specific mode. In the *Workload*, the GA generates valid values to the requests' parameters with the purpose of getting responses with status code 200. The generation of these values is based on the information parsed from the RESTful API description file (i.e., data types, domain of the values), the fitness function, and the variation operators (i.e., crossover and mutation). Whereas in the *Fautload*, invalid requests are generated with injected faults (i.e., replace a parameter with empty value) in an attempt to trigger unexpected behavior from the service (e.g., a response with status code *500 - internal server error*). In both phases, we took advantage of several elements of a genetic algorithm, such as mutation, crossover, parent selection, survival selection, and fitness function. We detail these in the following paragraphs.

The ***Converter*** component converts the raw requests randomly generated from the parsed information gathered in the Interface description analysis into individuals with fitness values. These individuals will be part of the initial population of our genetic algorithm. Additionally, this component can sort and filter requests that may not be appropriate for the current phase (i.e., Workload or Faultload), not affecting the initial population.

The ***Executor*** receives the individuals, processes the raw request associated with each individual, sends it to the service, and waits for a response. The response helps gather information regarding the system's behavior. The lack of one is also a strong indicator that something internally in the service went wrong. Additionally, the gathered information has the status code and the content returned that will be used in the ***Evaluator*** component.

In the ***Evaluator*** the individuals are assigned their fitness values using a function. The responses obtained from the REST API always have a status code and content associated. Depending on the response's status code and content, the *Evaluator* assigns a numeric value to each individual. Our approach is a multimodal optimization problem (i.e., multiple requests can lead to multiple responses, each of which will be a solution to our problem) since we can find multiple global and local optima (rather than a single solution), so the only guidance we have is the feedback retrieved from the REST service itself. In this note, if valid requests are prioritized, then every response with a status code 2xx should be considered a solution (i.e., 2xx represents any status code in the range of 200-299). Whereas, if invalid requests are prioritized, for instance, to trigger robustness problems in the service, then a status code 500 would be fitter than a 200 or even a 404.

The following equation describes the components used to assess the quality of each individual in the ***Evaluator*** component:

$$F(X) = SCWeight * SC(status\,code) + LVTWeight * distance(X_o, X_c) \tag{4.1}$$

where *SC* represents the function that evaluates the individual against the status code returned in the response described by:

Workload	Fautload
$SC(status\,code) = \begin{cases} 1.0, & \text{if status code = 2xx} \\ 0.4, & \text{if status code = 4xx} \\ 0.0, & \text{if status code = 5xx} \\ 0.0, & \text{otherwise} \end{cases}$	$SC(status\,code) = \begin{cases} 0.0, & \text{if status code = 2xx} \\ 0.4, & \text{if status code = 4xx} \\ 1.0, & \text{if status code = 5xx} \\ 0.0, & \text{otherwise} \end{cases}$

The function *distance* calculates the average distance between the responses' content of two individuals X_o and X_c, using the normalized Levenshtein distance [113]:

$$meanDistances(X_o, X_c) = levenshtein\,distance(x_{or}, x_{cr}) \tag{4.2}$$

where x_{or} and x_{cr} represent the string value of the response's content of both individuals. We compare the response content of one of the parents, chosen randomly, to the current individual (i.e., child). Subsequently, the *levenshtein distance* function calculates the distance between the response content of both individuals. The calculated distance is normalized with the maximum possible distance between both strings (e.g., for strings "car" and "house", the maximum possible distance is five since the longest string has a length of five). As a result, we reward individuals with a response content that is more distinct than its father since we consider that different responses' content may result in higher code coverage when the status code is equivalent in both individuals. Moreover, both functions, *SC* and *distance*, have an associated weight in the fitness function, *SCWeight* and *LVTWeight*, respectively.

Regarding the status codes 1xx and 3xx groups in the implemented fitness function, we consider them an imperfect solution due to their meaning in the services. A 1xx status code is an informative status code indicating that the server has received the request and is still processing it. This will mostly never happen since these are purely temporary and are given while the request processing continues until the complete response is retrieved. On the other hand, the 3xx group implies multiple possible responses to the request, and one of these should be chosen by the user-agent or the user himself. Since there is no conventional method for selecting one of the responses, this response status code is rarely used.

We also examined the OpenAPI specification file of 2313 public APIs available in the APIs.guru database [149], where we counted the number of occurrences of each status code per file. If a status code appeared multiple times in the specification file, we only counted it as one occurrence. The idea was to understand the most commonly used status codes in the most popular APIs. This analysis allowed us to verify that **the status codes groups 1xx and 3xx combined represent less than two percent of the total number of occurrences**, followed by the group of 5xx with 12%, 2xx with 40%, and 4xx with 47% due to its high variety specified by the developers. Even though the group of the status code 4xx has the most cumulative occurrences, the status code 200 itself appeared in 2302 of the 2313 OpenAPI files analyzed.

The ***Parent Selection*** is the component of our Genetic Algorithm that selects the parents who will breed and recombine to create offspring for the next generation. This process is essential to the convergence rate of the GA as good parents drive individuals to better and fitter solutions. We used the *Tournament Selection method* which chooses *k* number of individuals (i.e., pool size) randomly from the current generation and then selects the fitter one, or, in other words, the one with the highest fitness value to be a parent. The *k* number of individuals that compose the pool size of the tournament is parameterizable in the command-line of our tool. This way, the user/tester can control the selection pressure, which determines the rate of convergence of the GA, and it is a probabilistic measure of an individual's likelihood of participation in the tournament based on the participant selection pool size. Consequently, weak individuals are less likely to be selected for a larger tournament since a stronger individual will also have a high probability of being in the same tournament and, therefore, be selected.

After selecting the parents, the ***Crosser*** component is in charge of performing the crossover in each pair of parents. Each pair has a crossover probability (p_c), which decides if that pair will undergo crossover. As with many other probabilities, the p_c is also parameterizable in the command-line of our tool by the user. For the crossover operator, we implemented a variation of the *multi-point crossover*, where we defined each parameter (i.e., gene) in the request (i.e., individual) as a crossover point. Here, the technique will cross the value of that specific parameter's value between the two individuals. For each pair under crossover, we defined a probability of half of the number of parameters present in the request (i.e., individual). This means that, half of the genes are switched between the two individuals. In Figure 4.5, we illustrate the application of crossover between two individuals with parameters located in the request payload as a JSON Object, and the implemented crossover without recursiveness, which means that for structures (i.e., objects and arrays, see Table 4.1), the entirety of the value is switched between the individuals. In addition, the blue and green colors are present to facilitate the distinction between the values that underwent crossover.

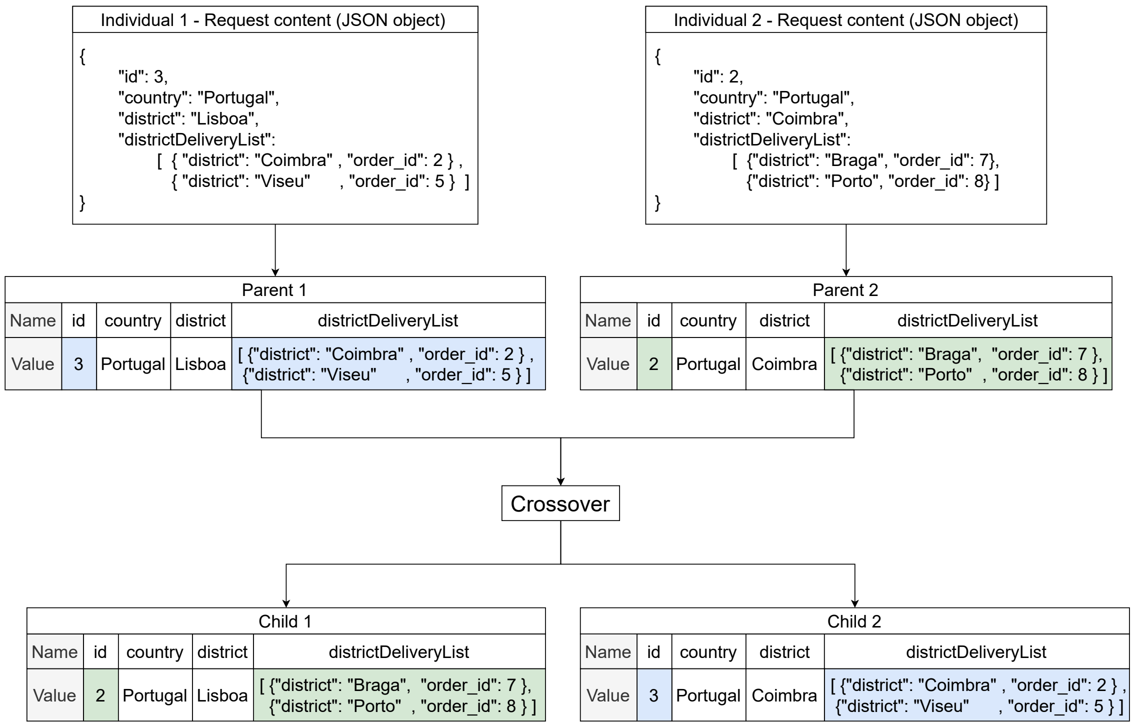
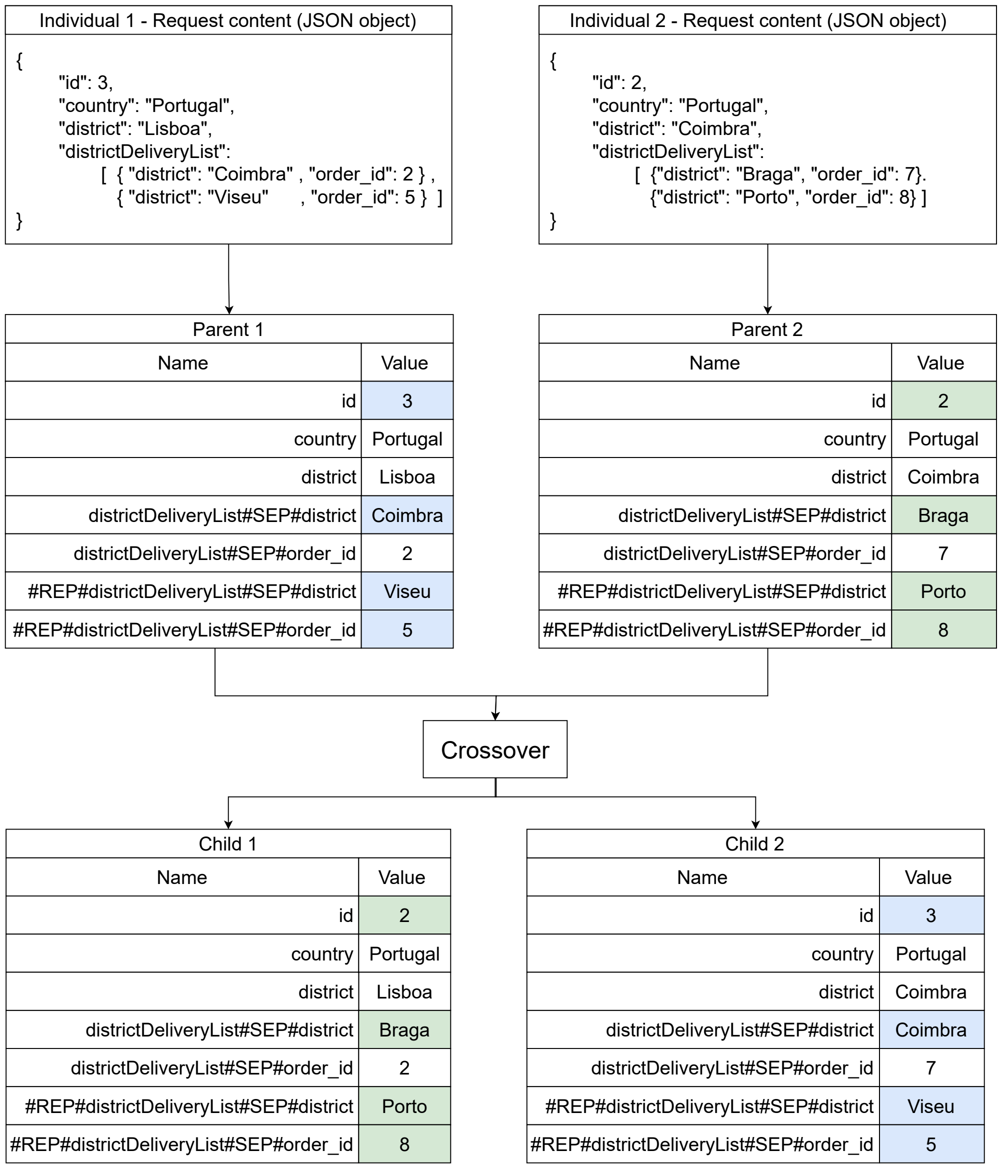

Individual 1 - Request content (JSON object)
{
"id": 3,
"country": "Portugal",
"district": "Lisboa",
"districtDeliveryList":
[{ "district": "Coimbra" , "order_id": 2 } ,
{ "district": "Viseu" , "order_id": 5 }]
}
Individual 2 - Request content (JSON object)
{
"id": 2,
"country": "Portugal",
"district": "Coimbra",
"districtDeliveryList":
[{"district": "Braga", "order_id": 7},
{"district": "Porto", "order_id": 8}]
}
Parent 1
Name | id | country | district | districtDeliveryList
Value | 3 | Portugal | Lisboa | [{"district": "Coimbra" , "order_id": 2 } , {"district": "Viseu" , "order_id": 5 }]
Parent 2
Name | id | country | district | districtDeliveryList
Value | 2 | Portugal | Coimbra | [{"district": "Braga", "order_id": 7 }, {"district": "Porto" , "order_id": 8 }]
Crossover
Child 1
Name | id | country | district | districtDeliveryList
Value | 2 | Portugal | Lisboa | [{"district": "Braga", "order_id": 7 }, {"district": "Porto" , "order_id": 8 }]
Child 2
Name | id | country | district | districtDeliveryList
Value | 3 | Portugal | Coimbra | [{"district": "Coimbra" , "order_id": 2 } , {"district": "Viseu" , "order_id": 5 }]

Figure 4.5: Implemented Crossover **without** recursiveness

As it is illustrated in Figure 4.5, each individual has seven parameters, *id*, *country*, *district*, and *districtDeliveryList* which counts as four. The *districtDeliveryList* parameter is an array (i.e., structured parameter) that may have multiple elements, and each element is an object with the parameters *district* and *order_id*. In our context, these are also considered parameters. For this particular case, the *districtDeliveryList* parameter has two elements, each with two parameters resulting in an individual with seven different genes.

For the Arrays and Objects, we also implemented a **crossover with recursiveness** to extract every possible parameter contained in these structures. Arrays may have elements that are objects, objects may have parameters that are arrays, or objects may even have parameters that are also objects. Subsequently, a tree of parameters inside parameters may occur until a certain point, and to overcome this challenge, we used a recursive function to extrapolate every contained parameter. Figure 4.6 gives an example of such implementation. We used two separators, #REP# and #SEP# , to distinguish between different elements of an array and different parameters within an object, respectively. For instance, if the *districtDeliveryList* array had four elements instead of only two, we would store a key with the name "`#REP##REP##REP#districtDeliveryList#SEP#order_id`" to identify the *order_id* parameter in the fourth element. Additionally, the user/tester can choose between the crossover with or without recursiveness in the command-line of our tool.


Individual 1 - Request content (JSON object)
{
"id": 3,
"country": "Portugal",
"district": "Lisboa",
"districtDeliveryList":
[{ "district": "Coimbra" , "order_id": 2 } ,
{ "district": "Viseu" , "order_id": 5 }]
}
Individual 2 - Request content (JSON object)
{
"id": 2,
"country": "Portugal",
"district": "Coimbra",
"districtDeliveryList":
[{"district": "Braga", "order_id": 7}.
{"district": "Porto", "order_id": 8}]
}
Crossover

Parent 1	
Name	Value
id	3
country	Portugal
district	Lisboa
districtDeliveryList#SEP#district	Coimbra
districtDeliveryList#SEP#order_id	2
#REP#districtDeliveryList#SEP#district	Viseu
#REP#districtDeliveryList#SEP#order_id	5

Parent 2	
Name	Value
id	2
country	Portugal
district	Coimbra
districtDeliveryList#SEP#district	Braga
districtDeliveryList#SEP#order_id	7
#REP#districtDeliveryList#SEP#district	Porto
#REP#districtDeliveryList#SEP#order_id	8

Child 1	
Name	Value
id	2
country	Portugal
district	Lisboa
districtDeliveryList#SEP#district	Braga
districtDeliveryList#SEP#order_id	2
#REP#districtDeliveryList#SEP#district	Porto
#REP#districtDeliveryList#SEP#order_id	8

Child 2	
Name	Value
id	3
country	Portugal
district	Coimbra
districtDeliveryList#SEP#district	Coimbra
districtDeliveryList#SEP#order_id	7
#REP#districtDeliveryList#SEP#district	Viseu
#REP#districtDeliveryList#SEP#order_id	5

Figure 4.6: Implemented Crossover **with** recursiveness

The ***Mutator*** component is responsible for mutating the values of the parameters in the individuals. However, there are several aspects that one needs to take into account. First, every data type (see Table 4.1) needs a different mutation. Secondly, we may want to implement different mutation types for each data type. Finally, mutation for the Workload phase differs from a mutation in the Faultload phase. As such, we considered these aspects when implementing the code, and it is easily expansible with additional mutations for the different data types where the user may choose which one to perform.

Similar to the crossover implementation with recursiveness, we adopted the same approach to mutation. Here, the mutation probability, p_m, is by default 1.0 divided by the total number of parameters present in the individual. For instance, in the example of Figure 4.7, the mutation probability for each parameter would be 0.25 ($1.0/4.0 = 0.25$), since the individual has four parameters. For **non-structured parameters** (i.e., any other data type that is not an array or object), we mutate the value. Whereas, for **structured parameters** (i.e., Arrays and Objects), the mutation is chosen randomly from three types of mutations: Add one element/parameter (Array/Object), Remove one element/parameter (Array/Object), or mutate the value of a parameter contained in the structure itself (i.e., *Mutate value*). If the last is chosen, the described process is repeated for the parameters inside the structure. However, Arrays may have a minimum and a maximum number of elements specified in the RESTful API description. Objects may or may not have additional parameters to be added or removed (i.e., parameters may or may not be required). Therefore, if the mutation type "Add one element/parameter" or "Remove one element/parameter" cannot be performed to the structured data type, we mutate its value. This value, in an Array, is a randomly selected element, and in an Object is a randomly selected parameter. For instance, if the Array element is a structured data type (i.e., an Object), the process will repeat recursively. On the other hand, for the selected parameter in the Object, if it is a non-structured data type, then we mutate the value itself (i.e., mutating an integer with the current value plus two).

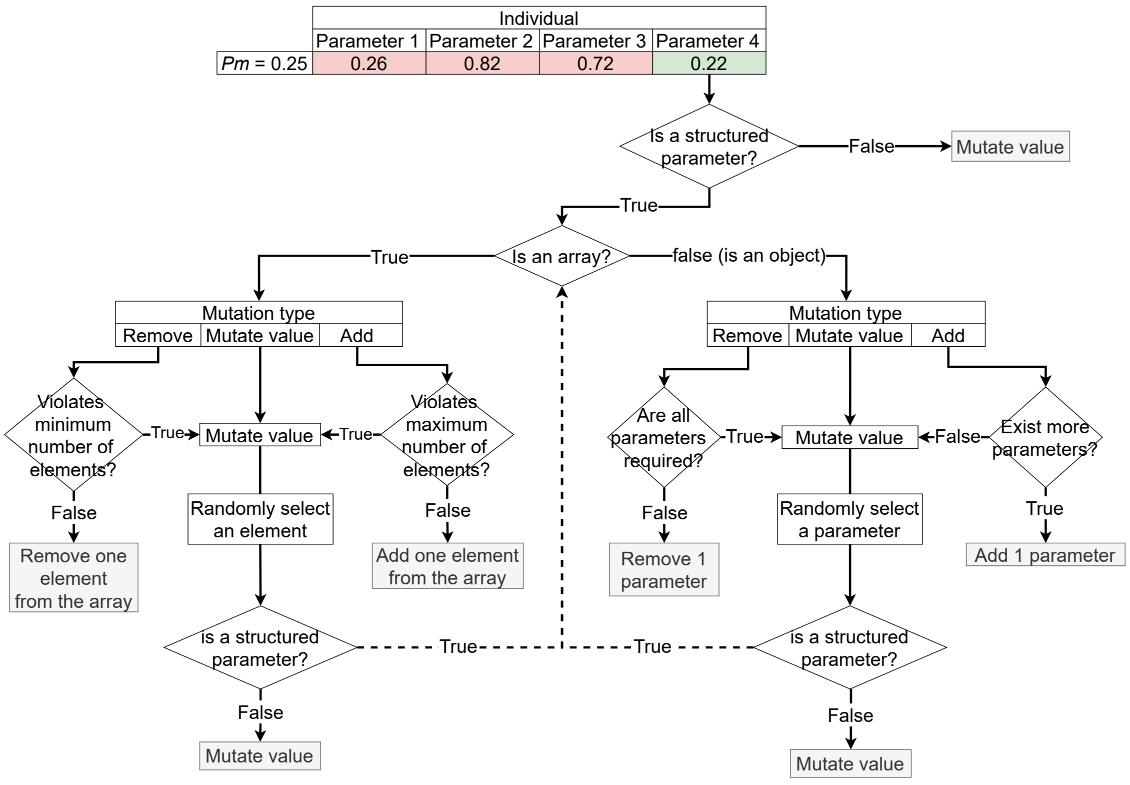


Figure 4.7: Generic example of the implemented Mutation **with** recursiveness

A generic example of the implemented Mutation with recursiveness for the Workload phase is demonstrated in Figure 4.7. Looking at the Flowchart (Figure 4.7), the fourth parameter is selected for Mutation due to the probability being 0.22, which is lower than 0.25 ($1.0/4 = 0.25$). Here, suppose the fourth parameter is a structured data type. In that case, the ***Mutator*** component randomly selects the mutation type to be performed from the set of three (i.e., Add one element/parameter, Remove one element/parameter, or Mutate value). Depending on which structured data type (i.e., array or object), there are conditions from the RESTful API description to be respected. For instance, Arrays may have a minimum and a maximum number of elements specified in the RESTful API description. On the other hand, objects may or may not have additional parameters to be added or removed. Therefore, we mutate its value if either mutation type “Add one element/parameter” or “Remove one element/parameter” is chosen but cannot be performed to the structured data type. In other words, the *Mutator* component selects the mutation type “Mutate value” since the previously selected one could not be executed.

A practical example with actual values of the implemented Mutation with recursiveness for the Workload phase is demonstrated in Figure 4.8. Here, the grey cells represent a hypothetical view, and the green ones represent the example of the mutation type "Mutate value" always being selected randomly. On the other hand, the red cells indicate that the parameter, element, or Mutation type, was not selected. Looking at the Flowchart (Figure 4.8), the parameter *districtDeliveryList* is an array, and it is selected for Mutation due to the probability being 0.22, which is lower than 0.25 $(1.0/4 = 0.25)$. Since it is a structured data type, the *Mutator* component randomly selects the mutation type to be performed from the set of three (i.e., Add one element/parameter, Remove one element/-parameter, or Mutate value). The mutation type "Mutate value" is then selected randomly. However, the *districtDeliveryList* is an array with Objects as elements and has two elements. Of the two elements, one is chosen to be mutated, and in the example, “element 2” is the one selected. Due to “element 2” being an object and, therefore, a structured data type, the process is then repeated where one of the three mutation types is chosen to be executed in “element 2”. In the example, the “Mutate value” type is again selected. Since “element 2” has two parameters, *district*, and *order_id*, one of them must be selected with a probability of 0.5 $(1.0/2 = 0.5)$. The parameter district is selected and undergoes Mutation in its value. The value is mutated from “Viseu” to “Faro”. The result is a new individual with one mutated parameter in the request.

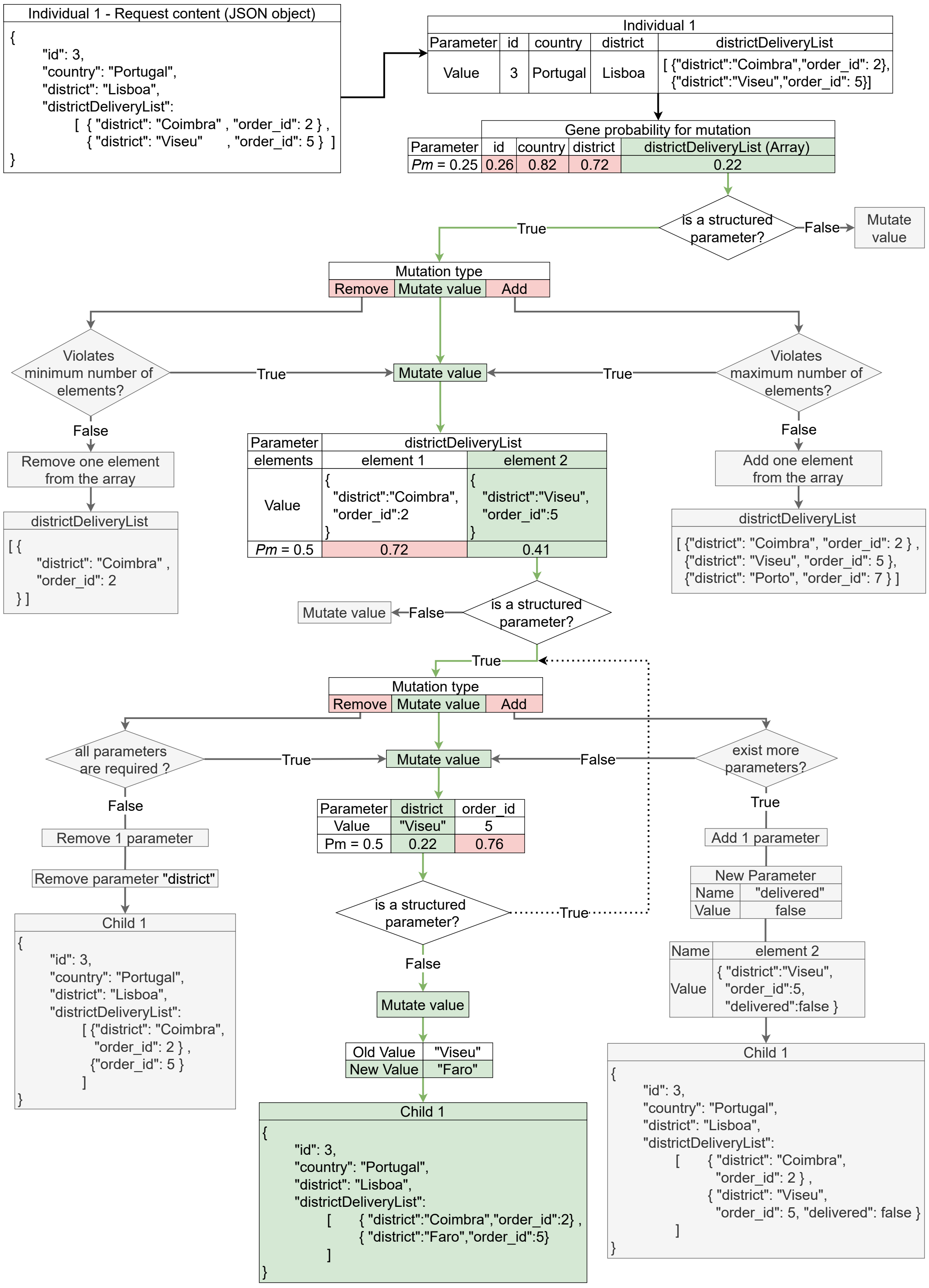

Individual 1 - Request content (JSON object)
{ "id": 3, "country": "Portugal", "district": "Lisboa", "districtDeliveryList": [{ "district": "Coimbra" , "order_id": 2 } , { "district": "Viseu" , "order_id": 5 }] }
Individual 1
Parameter | id | country | district | districtDeliveryList
Value | 3 | Portugal | Lisboa | [{"district":"Coimbra","order_id": 2}, {"district":"Viseu","order_id": 5}]
Gene probability for mutation
Parameter | id | country | district | districtDeliveryList (Array)
Pm = 0.25 | 0.26 | 0.82 | 0.72 | 0.22
is a structured parameter?
True
False
Mutate value
Mutation type
Remove | Mutate value | Add
Violates minimum number of elements?
True
False
Remove one element from the array
districtDeliveryList
[{ "district": "Coimbra" , "order_id": 2 }]
Mutate value
Violates maximum number of elements?
True
False
Add one element from the array
districtDeliveryList
[{"district": "Coimbra", "order_id": 2 } , {"district": "Viseu", "order_id": 5 }, {"district": "Porto", "order_id": 7 }]
Parameter | districtDeliveryList
elements | element 1 | element 2
Value | { "district":"Coimbra", "order_id":2 } | { "district":"Viseu", "order_id":5 }
Pm = 0.5 | 0.72 | 0.41
is a structured parameter?
False
Mutate value
True
Mutation type
Remove | Mutate value | Add
all parameters are required ?
True
False
Remove 1 parameter
Remove parameter "district"
Child 1
{ "id": 3, "country": "Portugal", "district": "Lisboa", "districtDeliveryList": [{"district": "Coimbra", "order_id": 2 } , {"order_id": 5 }] }
Mutate value
exist more parameters?
False
True
Add 1 parameter
New Parameter
Name | "delivered"
Value | false
Name | element 2
Value | { "district":"Viseu", "order_id":5, "delivered":false }
Child 1
{ "id": 3, "country": "Portugal", "district": "Lisboa", "districtDeliveryList": [{ "district": "Coimbra", "order_id": 2 } , { "district": "Viseu", "order_id": 5, "delivered": false }] }
Parameter | district | order_id
Value | "Viseu" | 5
Pm = 0.5 | 0.22 | 0.76
is a structured parameter?
True
False
Mutate value
Old Value | "Viseu"
New Value | "Faro"
Child 1
{ "id": 3, "country": "Portugal", "district": "Lisboa", "districtDeliveryList": [{ "district":"Coimbra","order_id":2} , { "district":"Faro","order_id":5}] }

Figure 4.8: Practical example of the implemented Mutation **with** recursiveness

Regarding **how the values are mutated** from the *Mutator* component for the different data types, we followed the model present in Table 4.2 for the Workload and Table 4.3 for the Faultload. The tables are systematized by data type and format defined in the OpenAPI specification [6]. A data type may have multiple formats (e.g., a number may be an integer of 32-bit, an integer 64-bit, a float, a double). Here, the goal is to generate parameters' values based on the constraints described in the OpenAPI description file of the RESTful API. For instance, numerical values may have described maximum and minimum constraints. However, in the absence of these constraints, we assume the Maximum and Minimum value of the data type (e.g., 32-bit integer maximum value is 2147483647). On the other hand, date and date-time data types follow a significantly restricted pattern, resulting in a straightforward process to generate valid values in compliance with their patterns.

Table 4.2: Workload mutation model

OpeanAPI data types		Parameter mutation description
Array	-	Mutate value from an element inside the array
		Add 1 element in the array, if possible
		Remove last element in the array, if possible
Object	-	Mutate value from a parameter inside the object
		Add 1 parameter in the object, if possible
		Remove 1 parameter in the object, if possible
Boolean	-	Negate boolean value
Number	32-bit integer, 64-bit integer, Single-precision (float), Double-precision (double)	Mutate value under the following Algorithm's 1 logic
String	No format specified, Password, Binary	Generate random printable character string (probiability of 50%) OR Randomly select a string from a set of 466 000 english words (probiability of 50%)
	Byte	Equal parameter mutation as the *No format specified* but base64-encoded
	Date	Generate a new date from the previous value (result is always a valid date)
	Date-time	Generate a new date-time from the previous value (result is always a valid date-time)

Furthermore, Strings that do not have any format and, therefore, without a strict pattern make the process of generating values according to their semantics a challenging problem (e.g., generate a dog breeds name from a description that, in most cases, is not specified in the OpenAPI file). However, we are not trying to generate strings to obey the semantic problem but instead generate valid values according to the constraints in the description of the service. To achieve this, we resorted to the generation of a random string with printable characters and retrieved English words from a data set containing over 466 000 words [150]. The method to produce the string is chosen randomly, where each one has a 50% probability.

In the literature, to contextualize, the standardized mutation step size for numerical values is the following [147]:

- Integer numbers get altered by a $\pm 2^i$ delta, where i is randomly chosen between 0 and a max value that decreases during the search (e.g., from an initial 30 to down to 10).
- For float/double values, the same kind of $\pm 2^i$ delta is applied, but multiplied by a

Gaussian value (mean 0 and standard deviation 1), where $i = 0$ has higher chances (e.g., 33%) to be selected compared to the other values.

Algorithm 1 Variation operator for numerical values

```
procedure MUTATION($value, iteration, totalIterations$)
    do
        $midPoint \leftarrow (Maximum + Minimum)/2.0$
        $scale \leftarrow Maximum - midPoint$
        $scale \leftarrow scale * ((totalIterations - iteration)/totalIterations)$
        $nextValue \leftarrow value + random.nextGaussian * scale$
    while $!(Minimum < nextValue \ \&\& \ Maximum > nextValue)$
    return $nextValue$
end procedure
```

With such in mind, we use algorithm 1 to mutate numerical data types. We followed some ideas from the literature, such as using a mutation step size based on a random Gaussian [62]. As well as the mutation step size changing according to the number of iterations (i.e., current generation index) since this change could be advantageous, as stated by [72, 73, 89].

In this note, the main goal of Algorithm 1 is to, in initial iterations, alter the current value by a delta close to the extremes Maximum and Minimum. Whereas, in later iterations, the delta reaches values close to 1. To contextualize algorithm 1, the *scale* variable is the window size between the *Mid Point* and the *Maximum Point*, as illustrated in Figure 4.9. Here, the *scale* value is multiplied by the normalization of the current iteration to the total iterations possible (i.e., number of generations) in the Genetic Algorithm. The result of this multiplication will be the standard variation of the random Gaussian and the mean the current value being mutated, resulting in $x_i' = N(x_i, scale)$, where x_i' is the new value and x_i is the current value undergoing mutation.

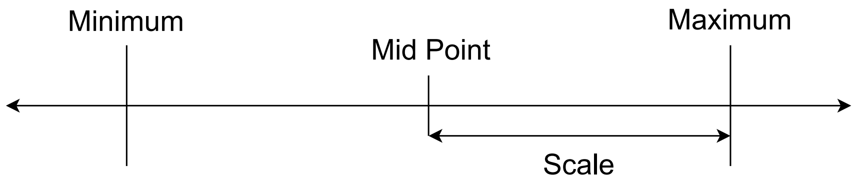


Figure 4.9: Mid point and Scale

Regarding the **fault model** (Table 4.3), it was based on [8], where each of the 32 faults is described as a mutation logic for the request parameters. Moreover, the string fault types, printable and non-printable, refer to the specific parts of the ASCII table [151], and malicious are Structured Query Language (SQL) injection strings from a set of 801 different ones acquired in [152]. Concerning the faults implemented (Table 4.3), **the mutation types applied in the Workload phase** (i.e., "Add one element/parameter" and "Remove one element/parameter") **will not be implemented in the Faultload phase** since the mutation "Replace with null" substitutes the "Remove one element/parameter", and the "Add one element/parameter" can be replaced by a fault mutation that cleans all the injected faults. However, this fault mutation is disabled in our implementation, and we let the evolutionary algorithm conduct the evolution of the injected faults. In other words, if an individual has too many faults injected (i.e., not producing robustness problems), it will be discarded in future generations, and individuals with fewer injected faults will be rewarded. Nonetheless, we implemented it as an extension and, if needed, to allow future users of our tool to take advantage of it.

Table 4.3: Fault model

OpeanAPI data types		Parameter mutation description
All	Applicable to all types	Replace with empty value
		Replace with null
Array	-	Duplicate random elements in the array
		Remove all elements in the array
		Remove random element in the array
		Add Elements Untill Max size + 1
Boolean	-	Negate boolean value
		Overflow string representation of boolean value
Number	32-bit integer, 64-bit integer, Single-precision (float), Double-precision (double)	Replace with 0
		Replace with parameter domain Maximum
		Replace with parameter domain Maximum + 1
		Replace with parameter domain Minimum
		Replace with parameter domain Minimum + 1
		Replace with data type Maximum
		Replace with data type Maximum +1
		Replace with data type Minimum
		Replace with data type Minimum -1
String	No format specified, Password	Append random printable characters to overflow maximum length
		Replace with random printable character string of equal length
		Replace with random non-printable string of equal length
		Append random non-printable characters at the end
		Replace with an extensive random printable character string
		Insert random non-printable characters at random positions
		Append SQL injection attack to original value
	Byte, Binary	Duplicate random elements to overflow maximum length
		Swap a random number of element pairs in the string
	Date, Date-time	Add a considerable number of years
		Replace value with random invalid date
		Subtract a considerable number of years
	Date-time	Add 24 hours
		Replace value with random invalid time
		Subtract 24 hours

These faulty mutations are injected into the requests in the *Mutator* component in the Faultload phase. These mutations' objective is to trigger erroneous behavior in the REST service. Therefore, individuals that generate requests that originate responses, for instance, with the status code *500 - internal server error*, will have a high fitness value in the population.

The ***Survival Selection*** component is responsible for a number of solutions (i., parents) in each generation to be inserted into the next without undergoing any change. This process is also known as elitism, a strategy in evolutionary algorithms. Consequently, in this component, an *X* number of worst solutions from the offspring are replaced by the *X* best solutions of the current generation, where *X* is the number of elites that are going to survive for the next generation. The X value is parameterizable in the tool's command line.

Algorithm 2 Pseudo-code of the EvoReFuzz algorithm

Input: Population size PS, Generation size GS, Mutation probability p_m , Crossover probability p_c, Elitism quantity e_q

Output: Hashmap with all generations for each operation available in the RESTful API

```
 1: $randomRequests \leftarrow generateRandomRequests()$
 2: $initialPopulation \leftarrow Converter.convert(randomRequests)$
 3: $Executor.sendRequests(initialPopulation)$
 4: $Evaluator.evaluate(initialPopulation)$
 5: $allGenerations.add(initialPopulation)$
 6: $i \leftarrow 1$
 7: while $i < GS$ do
 8:     $currentGeneration \leftarrow allGenerations.get(i-1)$
 9:     for each $operation \in currentGeneration$ do
10:         $currentPopulation \leftarrow currentGeneration.get(operation)$
11:         while $offSpring.size() < PS$ do
12:             $parents \leftarrow ParentSelection.tournament(currentPopulation, 2)$
13:             if $Random.double <= p_c$ then
14:                 $childs \leftarrow Crosser.crossover(parents)$
15:             else
16:                 $childs \leftarrow parents$
17:             end if
18:             $childs \leftarrow Mutator.mutate(childs, p_m)$
19:             $offSpring.add(childs)$
20:         end while
21:         $Executor.sendRequests(offSpring)$
22:         $Evaluator.evaluate(offSpring)$
23:         $SurvivalSelection.select(currentPopulation, offSpring, e_q)$
24:         $nextGeneration.put(operation, offSpring)$
25:     end for
26:     $allGenerations.add(nextGeneration)$
27:     $i++$
28: end while
29: return $allGenerations$
```

To conclude this chapter, we give a simple example of EvoReFuzz's Pseudo-code in algorithm 2. It describes the main algorithm, where each component used has a high complexity in the implemented code. We also would like to emphasize the fact that several different mechanisms or techniques could be implemented in our approach. However, our primary focus is to prove the concept of an evolutionary algorithm as the principal approach to generating valid and invalid requests.

Chapter 5

Experimental Setup

In this chapter, we describe the experimental setup that we used to evaluate the effectiveness of our Evolutionary approach for Robustness Testing of rest services, *EvoReFuzz*. In Section 5.1, we detail the process and the used material to execute the experiments. Section 5.2, details what command-line options were used and the overall environment of the experiences.

5.1 Experiments description

We divided the experimental setup into three different parts. The first one is focused on the code coverage analysis of our approach when compared to the state-of-the-art tool *EvoMaster* [19, 130, 147] using black-box testing In-house services. In the second one, we validated the effectiveness of producing robustness problems in public services, namely six GitLab APIs [17] and five Microsoft Bing Maps REST Services [18]. Lastly, we partnered with a Master's student to validate the RESTful backend API of his dissertation project with the goal of showing our approach's usefulness and how easy it is to set up the testing environment. In the subsequent paragraphs, we describe the services in which we conduct the experiments.

5.1.1 In-house Services

For the first part of the experimental setup, we used in-house services (i.e., locally running services) to analyze the code coverage. The main focus was to compare the code coverage reports between the *EvoReFuzz* tool and *EvoMaster* for black-box testing. EvoMaster [19, 21]uses an Evolutionary Algorithm and Dynamic Program Analysis to generate effective test cases. The approach is to evolve test cases from an initial population of random ones, trying to maximize measures like code coverage and fault detection. EvoMaster uses several kinds of heuristics to improve performance even further, building on decades of research in the field of Search-Based Software Testing. Such development resulted in the so-called MIO algorithm [128]. Moreover, the main focus of the tool is white-box testing. However, it has a black-box testing mode, which we will use to conduct these experiments.

The EvoMaster's authors created a set of web/enterprise applications for scientific research in Software Engineering called EvoMaster Benchmark (EMB) [153]. Of the 19 RESTful APIs available in EMB, 13 are written in Java. Moreover, we took advantage of these services and chose the ones that run in a JVM (i.e., Java Virtual Machine) environment and have any documentation on their Github page. We made this choice since we needed to measure the code coverage and analyze the open-source APIs that could run locally. Furthermore, it is easier to gather code coverage reports from the same programming language, or at negligibly not too many different ones, since, for each programming language, we would need to configure its own code-coverage tool to analyze the test results.

Table 5.1 shows the six selected REST services in which we will conduct the experiments. Each of these services runs as a Maven project and, therefore, in a Java Virtual Machine (JVM).

Table 5.1: In-house REST services

SUT	Files	JaCoCo LOCs	EMB	Source code
catwatch	106	1834	✓	-
cwa-verification	48	620	✗	From the official github page (v1.3.0)
features-service	39	457	✓	-
gestaohospital-rest	33	1055	✓	-
languagetool	1384	3568	✗	From the official github page (v5.8)
restcountries	24	543	✓	-
Total	1700	8760	-	-

The **Catwatch API** is a web application that fetches GitHub statistics for GitHub accounts, processes and saves the data in a database, and then makes it available via a REST API. The data reveals the popularity of the open source projects, most active contributors, and other information. It is compose of six operations.

Regarding the **CWA-verification-server** [154], it was developed by the official Corona-Warn-App [155] which is a COVID-19 contact tracing application used for digital contact tracing in Germany based on the exposure notification API from Apple and Google. The applications (for iOS and Android) use Bluetooth technology to exchange anonymous encrypted data with other mobile phones in the vicinity of an application user's phone. The data is stored locally on each user's device, preventing authorities or other parties from accessing or controlling the data. In the Corona Warn App world, the **Verification Server** helps validate whether upload requests from the mobile App are valid.

The **features-service** [156] is a RESTful API for managing products Feature Models, which is, in software development, a compact representation of all the products of the Software Product Line in terms of features. It allows for defining products, their available features, and their activation constraints. Define product configurations, understood as a set of functional features supported by the product that fulfill the constraints of the features. And query the active features for a configuration, so an application in runtime can change its behavior depending on the active features of a configuration.

The **gestaohospital-rest** service [157] was developed with the objective of creating an API to organize a public health system. The Unified Health System (SUS) is one of the world's largest and most complex public health systems, ranging from simple care for blood pressure assessment, through Primary Care, to organ transplantation, ensuring complete, universal access and accessibility for the entire population of the country. In this note, the API allows, for instance, the registration of the hospital, indicating the nearest hospital from the patient's location, performing patient check-in/check-out at the hospital, checking how many beds are available, registering products and their respective quantities, and registering and managing blood bank. Accordingly, it emulates the system management of a SUS hospital.

The **languagetool** service [158] is the most complex API in the In-house tested services, and it operates without a database. Additionally, it is an Open Source of proofreading software for more than 30 languages and, therefore, a CPU-bound application doing complex text analysis to find many errors that a superficial spell checker cannot detect.

Regarding the **restcountries** service [159], it is an API to get information about countries. For instance, it allows filtering by country code, currency, language code, capital city, region, regional bloc, name, and full name. The information retrieved for each country is exceptionally detailed, with several data fields.

Concerning the selection of these services, we chose to select the ones that contained a description on their GitHub official pages and had correct OpenAPI specifications files. However, some changes were needed in these files. For instance, some schemes regarding the HTTP protocol were not detailed since it is not a mandatory field. Regardless, such protocol information must be available for the *EvoReFuzz* tool to work. Another aspect is that several OpenAPI files are provided in JSON formats, and our tool only accepts YAML structured files. Such an aspect is not a problem as we can use the official swagger editor [160] to convert from JSON to YAML.

5.1.2 Public Services

Regarding the testing of public services, we conduct experiences in six APIs from GitLab [161], a popular open-source web service for hosting Git repositories, and five Microsoft Bing Maps REST Services [18]. We selected these services since they contain complex APIs and have been used in previous studies [10, 140]. Moreover, the authors of RestCT [140] also tested these REST services and compared their tool to RESTler [10]. They discovered eight bugs with RestCT, whereas RESTler only found one. Consequently, we wanted to directly compare with the study [140] and understand how EvoReFuzz, which targets robustness testing, would perform for the same services. As a result, we aimed to test and discover robustness problems (i.e., bugs) in these APIs to validate our Evolutionary approach's effectiveness.

We should also note that both Microsoft and Gitlab do not provide official OpenAPI specification files [6] to detail their services (Gitlab does not provide these files for version V4). Therefore, to overcome this challenge, the authors of RestCT manually created the required specifications based on its latest online documentation [17, 18]. They also made it available on their GitHub page [162] to aid others in replicating and extending their experiment. Accordingly, we used these manually created specifications and detailed them slightly better. For instance, there were incomplete Schemas and unprovided formats for specific data types (i.e., integers without the format int32 or int64, numbers without any formats).

In Table 5.2, we identify the tested APIs with the corresponding number of operations and the average number of parameters per operation. Regarding the functional description of each one of these APIs, the following paragraphs describe them in further detail.

Table 5.2: Public REST services

Company	API name	Number of endpoints/operations	Average parameters per operation
Gitlab	Branch	7	2,6
	Commit	13	5,1
	Groups	17	8,9
	Issues	25	7,1
	Project	31	10
	Repository	8	3,9
Microsoft Bing Map	Elevations	4	3,5
	Imagery	10	15,1
	Locations	5	7,6
	Route	14	12,8
	TimeZone	4	4,8

GitLab APIs

The **Branch API** [163] operates on repository branches, which is a version of a project's working tree. It allows the creation of a branch for each set of related changes to make. This keeps each set of changes separate from each other, allowing changes to be made in parallel, without affecting each other. The operations under this API allows the following actions:

- Get a list of repository branches from a project, sorted by name alphabetically.
- Create a new branch in the repository.
- Get a single project repository branch.
- Delete a branch from the repository.
- Protects a single repository branch or several project repository branches using a wildcard protected branch.
- Unprotect the given protected branch or wildcard protected branch, by ID and name.
- Delete all branches that are merged into the project's default branch.

The **Commit API** [164] operates on repository commits. The Git commits are one of the key parts of a Git repository, and more so, the commit message is a life log for the repository. As the project/repository evolves over time (new features getting added, bugs being fixed, architecture being refactored), commit messages are the place where one can see what was changed and how. The API has the following operations:

- Get a list of repository commits in a project
- Create a commit by posting a JSON payload.
- Get a specific commit identified by the commit hash or name of a branch or tag.
- Get all references (from branches or tags) a commit is pushed to.
- Cherry pick a commit to a given branch.
- Reverts a commit in a given branch.
- Get the diff of a commit in a project.
- Get the comments of a commit in a project.
- Get the discussions of a commit in a project.
- List the statuses of a commit in a project.
- Adds or updates a build status of a commit.
- Get a list of Merge Requests related to the specified commit.
- Get the GPG signature from a commit, if it is signed.

The **Groups API** [165] performs operations over the groups. These groups are used to manage permissions for projects. For instance, if someone has access to the group, they get access to all the projects in the group. The user may also view all of the issues, merge requests for the projects in the group, and view analytics that shows the group's activity. The Groups API is constituted of the following operations:

- Get a list of visible groups for the authenticated user.
- Create a new project group. Available only for users who can create groups.
- Get all details of a group. This endpoint can be accessed without authentication if the group is publicly accessible.
- Update the project group. Only available to group owners and administrators.
- Remove group, and queues a background job to delete all projects in the group as well.
- Share group with another group.
- Unshare the group from another group.
- Get a list of visible direct subgroups in this group.
- Get a list of group hooks.
- Adds a hook to a specified group.
- Get a list of group hooks

- Edits a hook for a specified group.
- Removes a hook from a group. This is an idempotent method and can be called multiple times. Either the hook is available or not.
- Get a list of visible descendant groups of this group.
- Get a list of projects in this group.
- Transfer a project to the Group namespace.
- Get a list of projects shared to this group.

The **Issues API** [166] performs operations over the used issues that allow the collaboration on ideas, solving problems, and planning work. Moreover, share and discuss proposals with the team and outside collaborators, track tasks and work status, and elaborate on code implementations. The Issues API allows the following operations under its logic:

- Get all issues the authenticated user has access to.
- Get a single issue, only for administrators.
- Get a list of visible groups for the authenticated user.
- Creates a new project group. Available only for users who can create groups.
- Get a list of a group's issues.
- Get a list of a project's issues.
- Creates a new project issue.
- Get a single project issue.
- Updates an existing project issue. This call is also used to mark an issue as closed.
- Deletes an issue. Only for administrators and project owners.
- Reorders an issue, you can see the results when sorting issues manually.
- Moves an issue to a different project.
- Subscribes the authenticated user to an issue to receive notifications.
- Unsubscribes the authenticated user from the issue to not receive notifications from it.
- Manually creates a to-do item for the current user on an issue.
- Sets an estimated time of work for this issue.
- Resets the estimated time for this issue to 0 seconds.
- Adds spent time for this issue.

- Resets the total spent time for this issue to 0 seconds.
- Get time tracking stats of an issue of a project.
- Get all the merge requests that are related to the issue.
- Get all merge requests that close a particular issue when merged.
- Get Participants on issues.
- Get List of metric image.
- Upload metric image.

The **Projects API** [167] interacts with the projects' resources. Here, projects can be created to host codebases. Moreover, projects can also be used to track issues, plan work, collaborate on code, and continuously build, test, and use built-in CI/CD to deploy the developed app. The operations that form this API are the following:

- Get a list of all visible projects across GitLab for the authenticated user. When accessed without authentication, only public projects with simple fields are returned.
- Creates a new project owned by the authenticated user.
- Creates a new project owned by the specified user. Available only for admins.
- Get a specific project.
- Updates an existing project.
- Deletes a specific project.
- Get the users list of a project.
- Get a list of visible projects owned by the given user.
- Delete an existing forked from relationship.
- List the projects accessible to the calling user that have an established, forked relationship with the specified project.
- Stars a given project.
- Unstars a given project.
- List the users who starred the specified project.
- Get languages used in a project with percentage value.
- Archives the project if the user is either an administrator or the owner of this project.
- Unarchives the project if the user is either an administrator or the owner of this project.
- Restores project marked for deletion.

- Uploads a file to the specified project to be used in an issue or merge request description, or a comment.
- Allow to share project with group.
- Unshare the project from the group.
- Get a list of project hooks.
- Adds a hook to a specified project.
- Get a specific hook for a project.
- Edits a hook for a specified project.
- Removes a hook from a project.
- Start the Housekeeping task for a project.
- Transfer a project to a new namespace.
- Create a forked from/to relation between existing projects.
- Download snapshot of a Git repository
- Get a list of visible projects owned by the given user.
- Get a list of visible projects owned by the given user.

The **Repositories API** [168] is responsible for conducting operations regarding the projects' repositories. In this context, a repository is where the code is stored and changes are made to it. These changes are tracked with version control. The operations under this API allow the subsequent actions:

- Get a list of repository files and directories in a project.
- Allow receiving information about a blob in the repository like size and content.
- Get the raw file contents for a blob by blob SHA.
- Get an archive of the repository.
- Compare branches, tags or commits.
- Get repository contributors list.
- Get the common ancestor for 2 or more refs (commit SHAs, branch names or tags).
- Generate changelog data based on commits in a repository.

Microsoft Bing Map APIs

The Bing Maps REST Services Application Programming Interface provides a Representational State Transfer interface to perform tasks such as creating a static map with pushpins, geocoding an address, retrieving imagery metadata, or creating a route [18].

The first API tested from the set of Microsoft Bing Maps REST services is the **Elevations API** [169], which is used to get elevation information for a set of locations, and polylines, or areas on the Earth. The user may perform the following actions through the available operations, such as:

- Get elevations for latitude and longitude coordinates.
- Get elevations at equally-spaced locations along a polyline path.
- Get elevations at equally-spaced locations within a bounding box.
- Get the offset of the geoid sea level Earth model from the ellipsoid Earth model.

The second one is the **Imagery API** [170], which is used to get static maps and Bing Maps imagery information. The following operations constitute the available actions in this API:

- Get a map that is centered at a specified point.
- Get a map that shows a specified map area.
- Get a map that is centered at the specified point and that displays a route.
- Get a map that shows a specified map area by specifying the image format to use for the static map and the desired camera heading in degrees, clockwise from north.
- Get a map that shows a specified map area by a query string that is used to determine the map location to display.
- Get a map that shows a specified map area from a point on the Earth where the map is centered.
- Get a map that shows a specified map area by querying a string that is used to determine the map location to display.
- Make a Local Search API request based on a string query by specifying a user location by the type of imagery for which the user is requesting metadata.
- Make a Local Search API request based on a string query by specifying a user location by specifying a center point.
- Make a Local Search API request based on a string query by specifying a user location, when using Basic Metadata.

The **Locations API** [171] is used to get location information. It allows, for instance, getting the latitude and longitude coordinates for a location by specifying values such as a locality, postal code, and street address. Alternatively, the location information associated with latitude and longitude coordinates or the latitude and longitude coordinates corresponding to the location information provided as a query string. The Locations API has the following operations available under its logic:

- Get the latitude and longitude coordinates based on a set of address values for any country.
- Get an address for a specified point (latitude and longitude).
- Return latitude and longitude coordinates for a location specified by a query.
- Get an address for a specified point (latitude and longitude) in a search radius.
- Make a Local Search API request based on a string query by specifying a user location.

The **Routes API** [172] is used to create a route that includes two or more locations and to create routes from major roads. The user can create driving or walking routes. Driving routes can include traffic information. The user can also overlay routes on map imagery. This API is one of the most complexes of the set of tested APIs since it has a large number of operations and a high number of average parameters per operation. In this note, the operations available in it are the following:

- Find a driving route. It gets a walking, driving or transit route by specifying a series of waypoints. A waypoint is a specified geographical location defined by longitude and latitude that is used for navigational purposes. The route includes information such as route instructions, travel duration, travel distance or transit information.
- Find a walking, driving or transit route by specifying the mode of travel.
- Get travel routes which take truck attributes such as size, weight and type of cargo. This is important as not all trucks can travel the same routes as other vehicles.
- Find routes from major roads in four directions (West, North, East, South).
- Synchronous POST Optimize Itinerary which returns an itinerary schedule for one or more agents to travel between multiple itinerary items (e.g., between multiple delivery locations).
- Asynchronous POST Optimize Itinerary.
- Synchronous Distance Matrix Request URL retrieves a simple distance matrix for a set of origins and destinations using an HTTP POST request.
- Asynchronous Distance Matrix Request URL creates a job to calculate a distance matrix using an asynchronous POST request. This type of request is only supported when the travel mode is set to driving. A start time must be specified when making asynchronous requests.

- Calculate an Isochrone Synchronous, which provides time-specific, isoline polygons for the distance that is reachable from a given location and supports multiple modes of transportation (i.e., driving, walking, and public transit).
- Calculate an Isochrone Asynchronous.
- Snap Points to Roads Synchronous by taking GPS point data, in the form of latitudes and longitudes, and returns a list of objects that form a route snapped to the roads on a map.
- Snap Points to Roads Asynchronous.
- Get local Insights Synchronous. It returns a list of local entities within the specified maximum driving time or distance traveled from a specified point on Earth.
- Get Local Insights Asynchronous.

The last API is the **Time Zone API** [173], which makes it easy to find a time zone and daylight savings time (DST) for a location by query or latitude and longitude coordinates. It also converts UTC date-time stamps to different time zones with DST information or retrieves a Time Zone by ID and provides a list of time zone information for either the Microsoft Windows or IANA time zone standard. This API is the least complex one from the set of the tested public services, and it is composed of the subsequent operations:

- Get the Time Zone from Location Point. It allows retrieving the time zone information for any point on Earth. Given a pair of coordinates or a place name *query*, the Time Zone API will return the location's local time zone and daylight savings (DST) information.
- Get the Time Zone from Location Name. Given a query for a location (e.g., query = Bellevue, WA) the Time Zone API finds that location and then returns information about the time zone for that location.
- Convert UTC Date-time to a different Time Zone.
- Get the Time Zone and its information from Time Zone ID (Windows or IANA).

5.1.3 ucXception's framework API

To highlight our approach's usefulness in validating robustness problems in RESTful APIs, we partnered with a student doing his Master's thesis in software engineering at the University of Coimbra. The theme of the student's dissertation is a framework to allow the performing of fault injection on various target systems, from a local system to virtualized or even cloud systems. The framework's name is ***ucXception*** [174], which conducts the evaluation of hardware and software fault tolerance mechanisms and the dependability of the systems by combining a suite of fault injection tools capable of emulating hardware or software faults with different fault models. The Backend module consists of two software components: the Manager and a REST API. Regarding the Manager of the framework, it is the software component responsible for executing the fault injection campaign and storing its results, where multiple instances of the Manager process are spawn, one for each campaign being executed. On the other hand, the REST API exposes the functionalities of the Manager to the Frontend module. Moreover, the Backend module also encompasses one database, where the information about the users and their campaigns is kept, as well as multiple CSV files containing each campaign's results.

Our focus was to evaluate the RESTful API's robustness that exposes the Manager's functionalities to the Frontend module. Such API was developed with Flask [175, 176], a small web framework written in Python. We then started by helping document the API with the OpenAPI library available for flask [176]. This way, an OpenAPI file is automatically generated, and we can start performing tests with EvoReFuzz.

5.2 Experimental Environments

In this section, we detailed which command-line options were used in the tested tools and the code coverage report obtained between experiences. Furthermore, we specified the used libraries, the system environment, and the files created to conduct the experiments.

Environment for In-house Services

In order to perform the experiences, we created batch scripts to start each system under test with the JaCoCo Agent [177] so that we may collect the code coverage reports at the end of each tool execution. *EvoReFuzz* and *EvoMaster* were executed for each of the six SUTs (see Table 5.1). This results in 2 × 5 x 6 + 6 = 66 batch scripts. Each script starts a SUT as a background process and then one of the tools, EvoReFuzz or EvoMaster. Each tool runs in a loop for one hour, and we export the code coverage report at the end of each hour. As a result, even if the tool crashes or ends before the period of one hour, we restart it. We repeat this procedure ten times to account for the randomness of the tools. Moreover, the code coverage is computed based on all the HTTP calls done during the fuzzing process. Such was adopted for the case that if a tool crashes, we are still measuring what code coverage it conducts.

Regarding the code coverage analysis, we developed a project called Jacoco-Report-Generator [178], which extrapolates and automatically stores the code coverage reports from the JaCoCo agent [177] for the services using JVM. The JaCoCo Agent allows the creation of a TCP Socket Server that listens for incoming connections from a TCP Socket Client to retrieve (i.e., request a dump) and read the last code coverage report. The execution data (i.e., code coverage report) is written to the socket connection on request. After the request to dump the execution data (i.e., the code coverage report), we can reset the execution data without restarting the JVM. Consequently, multiple reports can be saved and reviewed later. As a result, we were able to extract the code coverage report after each hour by simply executing the runnable jar file of the Jacoco-Report-Generator project.

We also changed the server port for each of the SUTs to perform parallel testing in the same machine, as well as the port of the TCP socket server of the JaCoCo agent running in each SUT. The experiments were carried out locally, and the machine used is composed of an AMD Ryzen 7 1700X with eight cores and sixteen threads, 16GB of RAM at 3200 MHZ, and Windows 10 Pro version 21H2. Every output was stored in an external SSD, SAMSUNG T5 with 256GB. Regarding the java versions utilized, for the SUTs running in java 8, we operated in version 1.8.0_202 and 11.0.15.1 for java 11.

Regarding the scripts, we used the subsequent ones with the names starter.bat, runToolFor1hourInLoop.bat, runEvoReFuzz.bat, runAndTerminateSUT.bat, and retrieveCodeCoverage.bat.

Listing 5.1: Starter script (starter.bat)

```
for /L %%A in (1,1,10) do (
    echo %%A
    start "rtsut" runAndTerminateSUT.bat 3635
    timeout /t 20
    runAndTerminateScriptAfterXtime.bat 3600
    timeout /t 3
    runCodeCoverageReportRetriever.bat %%A
    timeout /t 20
    taskkill /FI "WINDOWTITLE eq rtsut"
)
```

Listing 5.2: Script to run the tool for one hour (runToolFor1hourInLoop.bat)

```
@echo off
if "%1"=="" (
    set duration=10
) else (
    set duration=%1
)
start "myscript" runEvoReFuzz.bat
ping 127.0.0.1 -n %duration% -w 1000 > nul
echo %duration% seconds are over. Terminating!
taskkill /FI "WINDOWTITLE eq myscript*"
```

Listing 5.3: Run EvoReFuzz in loop Script (runEvoReFuzz.bat)

```
:loop
call java -jar "path\to\EvoReFuzz\target\Evolutionary_REST_Fuzzer-1.0.jar" ^
--wl-rep 40 ^
--api-file Config.java ^
--api-yaml-file openapi.yaml ^
--number-runs 1 ^
--generation-size 50 ^
--population-size 20 ^
--out "path\to\outputfolder\output.txt"
goto loop
```

Listing 5.4: Run SUT and terminate after one hour Script (runAndTerminateSUT.bat)

```
@echo off
if "%1"=="" (
    set duration=10
) else (
    set duration=%1
)
start "myscript" \path\to\SUTfolder\runSUT.bat
ping 127.0.0.1 -n %duration% -w 1000 > nul
echo %duration% seconds are over. Terminating!
taskkill /FI "WINDOWTITLE eq myscript*"
```

Listing 5.5: Retrieve the code coverage Script (retrieveCodeCoverage.bat)

```
java -jar path\to\Jacoco-Report-Generator\target\JaCoCo-report-generator.jar ^
--cc-classes "path\to\SUTfolder\target\classes" ^
--cc-source "path\to\SUTfolder\src\main\java" ^
--cc-socket-IP "127.0.0.1" ^
--folder-name "Coverage" ^
--execution-run %1 ^
--cc-socket-port "8502" ^
--out "path\to\output"
```

The starter script 5.1 initiates the `runAndTerminateSUT` script 5.4, which manages the time to kill the process that runs the SUT. Sequentially, after starting the SUT, the starter script will execute the `runToolFor1hourInLoop` script 5.2, which also manages the time to kill the process running the EvoReFuzz tool in a loop (`runEvoReFuzz` script 5.3). Allowing to restart the tool even if it crashes or stops earlier, before the targeted one hour, and still retrieve the code coverage with the `retrieveCodeCoverage` script 5.5. Consequently, the SUT runs a little longer than one hour to allow the testing tool to run for one hour and then retrieve the code coverage report. After these actions, the SUT is restarted, and the process repeats itself ten times or, in other words, ten hours. Therefore, the present "for" in the starter.bat file.

Regarding the **command-line options** chosen **to run EvoReFuzz** in script 5.3, we went for a population size of 40 and a generation size of 50. We defined these values to have a more diverse set of injected faults when executing the Faultload phase. Moreover, these values limit the number of requests per operation to 2000 (i.e., $50 * 40$), which in most complex systems will take a long time. The values can be optimized for each SUT, but we conducted the same ones for every SUT. Looking at the `wl-rep` option, it establishes the size of the pool of individuals that will constitute the initial population. For instance, for a population size of 20 and `wl-rep` of 40, the initial population will have 40 individuals, and the best 20 from the initial population will be selected for the first generation. Such is performed by the Converter component (recall Figure 4.2). We also prioritized a faster execution to finish before the one-hour mark than finishing over the killed process.

To run EvoMaster, the only scrip that needs a change is the runEvoReFuzz.bat, which we named runEvoMaster.bat and has the command-line options of the testing tool. The following script shows the used command-line options for EvoMaster.

Listing 5.6: Run EvoMaster Script (runEvoMaster.bat)

```
:loop
call java -jar "path\to\EvoMaster\core\target\evomaster.jar" ^
--blackBox true ^
--bbSwaggerUrl "file:///path/to/openapi.yaml" ^
--outputFormat JAVA_JUNIT_4 ^
--maxTime 60m ^
--outputFolder "\path\to\outputfolder" ^
--problemType REST ^
--exportCoveredTarget true ^
--writeStatistics true ^
--ratePerMinute 500
goto loop
```

Looking at the command-line options for EvoMaster, two aspects must be taken into account. The first is the option `maxTime` equal to 60m (i.e., 60 minutes), which limits the testing process to one hour. We chose this value since it is stated on the official EvoMaster's GitHub page that a more extended period of time will produce better results. Moreover, since we are limiting the experiences to one hour, we thought adequately to limit the hour mark in the command-line option of EvoMaster. Furthermore, during the experiences, we checked if the process would be killed before the complete execution of EvoMaster, and we witnessed that EvoMaster finishes first, and then the process is killed, not affecting the experiences. The second aspect is that the `ratePerMinute` option is limited to 500. In a pre-mature analysis of the testing environments, we noticed that when the option was not set, the EvoMaster would run out of TCP connections. Therefore, and considering this aspect, we limited it to 500 requests per minute.

We had to use a personalized script for the different SUTs to be able to have multiple ones running and extrapolate the code coverage of each one since they run in different TCP server sockets in the JaCoCo agent. As a result, the following script 5.7 shows a generic example of the batch file used to start a SUT with the JaCoCo agent.

Listing 5.7: Run SUT Script (runSUT.bat)

```
java
-javaagent:path\to\jacoco\jacocoagent.jar=includes=*,output=tcpserver,port=8501,
    address="127.0.0.1"
-jar path\to\SUTfolder\target\sut-runnable.jar
```

Here, the port stands for the port where the TCP Socket Server will listen for upcoming connections. These connections will retrieve (i.e., request a dump) and read the last code coverage report. After the request to dump the execution data (i.e., the code coverage report), we can reset the execution data without restarting the SUT. Consequently, multiple reports can be saved and reviewed later. Additionally, the address is the IP address of the TCP Socket Server to which a TCP Client socket will connect.

Environment for Public Services

For the APIs under GitLab and Microsoft Bing Maps, we have two different environments. The first relies on an existing **docker image to deploy the GitLab's APIs**, and the tests were carried out in a local environment on the same machine as the In-house services (with Ryzen 7 1700X, 16GB of RAM at 3200 MHZ, and Windows 10 Pro version 21H2). The docker image is from the official GitLab repository, in which we used version 13.10.3-ce.0 [179], the same image version as in the paper of the RestCT tool [140]. It emulates the available APIs for GitLab. Since it is running locally, further extensive testing can be carried out due to the lack of a rate-limited number of requests.

The second environment is for the services of Microsoft Bing Maps. We ran EvoReFuzz locally while **testing the public Microsoft Bing Maps APIs servers remotely** (i.e., *http://dev.virtualearth.net/REST/v1*) using a basic private key generated in the Bing Maps Dev Center [180]. Since we are sending requests to external APIs, the Microsoft Bing Maps services have a limit for each category of the generated key. We created our key as a Development/Testing one, and, therefore we **have a limit of 125,000 cumulative billable transactions**. As stated in Bing Maps transactions page [181]: only billable transactions count towards the free-use limits for Basic keys. Non-billable transactions do not incur charges and do not count towards free-use limits. We had to take that into account to not over-extend these restrictions. However, 125,000 requests is a high ceiling for testing such services.

Regarding the command-line option to run EvoReFuzz for testing the GitLab APIs, we used a population size of 40 and a generation size of 50, limiting to 2000 requests per operation. Due to the fact that we are running the docker image of the GitLab APIs, we could extend the time and, therefore, the number of requests for the conducted experiences. On the other hand, for Microsoft Bing Maps, we set up the population size with a size of 20 and a generation size of 50, limiting to 1000 requests per operation since we have a limit of requests as a developer that we can send to the remote Microsoft servers. Here, we had the same idea, as in the options used for the In-house services, of prioritizing a more diverse set of injected faults by setting a higher number of generations than the population. As a last note, we should also emphasize that **the testing of these Public services differs from the testing of the In-house services**. We want to find robustness problems for the public services, whereas, for In-house services, we are targeting a higher code coverage than EvoMaster. Furthermore, we did not run the In-house services experiences simultaneously as the public services, nor did we perform the tests in the GitLab APIs in parallel with the ones of Microsoft Bing Maps.

Environment for testing ucXception

To test the ucXception RESTful API, we needed to conduct the experience remotely. Figure 5.1 shows the overall context of both our and the student's environments. To be able to send requests remotely from our router to the student's router, we needed to port-forward the port where the REST API was deployed (i.e., port 5000). Furthermore, we also got the student's static public IP assigned by the Internet Service Provider 2 (meo), which can be done in the router's admin page at *http://192.168.1.254* (for MEO routers). After gathering such information and having the OpenAPI specification file, we were ready to carry out the experiences.

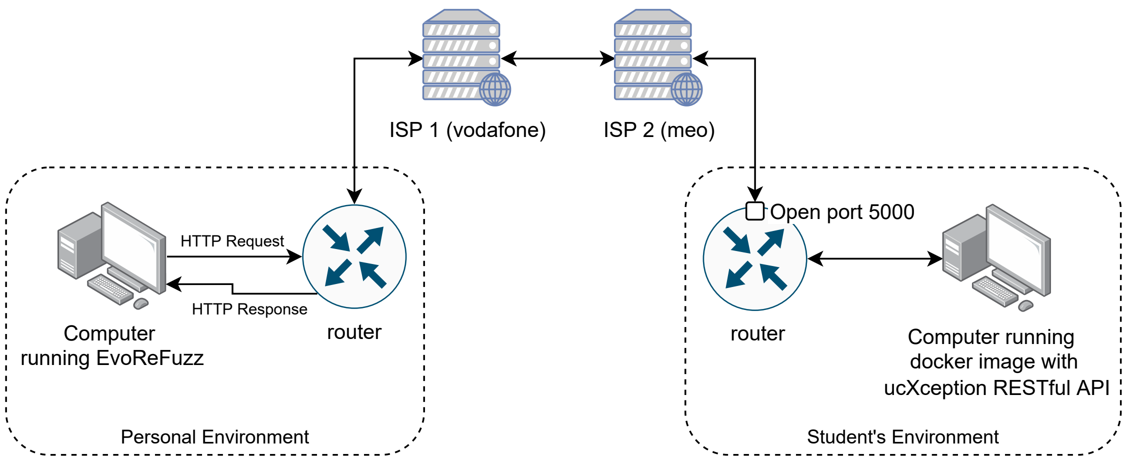


Figure 5.1: Experimental environment overview

Regarding the ucXception RESTful API, it was developed in Flask 2.0.3 and running in a docker image through the docker engine with version 20.10.16 in Windows 10 Pro version 21H2. The student's system is composed of an AMD Ryzen 5 3600 6-Core Processor, 16GB of RAM at 2666 MHZ, and an SSD with 3000 MB read and 1410 MB of writing speeds. On the other hand, EvoReFuzz was running in a JVM through a runnable jar (Java 8) in a Macbook Pro 14-inches 10cores CPU (M1 Pro) and 16 GB of RAM. Moreover, we defined a limit of 1200 requests per operation available in the API. In other words, a population size of 30 and a generation size of 40. We prioritize a higher generation size since it will result in a more diverse set of injected faults.

Chapter 6

Experimental Results

In this chapter, we present and analyze the results obtained taking into account the experimental setup of the previous chapter. We start by discussing at the In-house services, then the Public services, and the observations of the problems found in the ucXception API (i.e., partnership with a Masters's student). We conclude by reporting the main findings of all of the experiences.

6.1 Experimental Results

In-house services Results

Testing the set of In-house services to extract the code coverage took a total of 120hours of computational time. The results show that EvoMaster had a higher average line coverage in four of the six SUTs. Whereas EvoReFuzz performed better in two of them. Nonetheless, the overall average was close between both tools, with EvoMaster having an overall line coverage average of 47.01% throughout the six SUTs, and EvoReFuzz 45.14%. The experimental results of the two black-box fuzzers on the six RESTful APIs are depicted in Table 6.1, where for each tool, we report the average line coverage, as well as its [min, max] values out of the ten runs. We attained some essential conclusions in both the tested SUTs and the experimental results, which we will discuss in the subsequent paragraphs.

Table 6.1: In-house Services **line coverage** results

SUT	EvoReFuzz	EvoMaster
catwatch	**29.01** [29.01, **29.01**]	24.42 [24.42, 24.48]
cwa-verification-server	49.55 [44.52, 50.65]	**50.45** [50.32, **50.97**]
features-service	42.76 [40.92, 43.98]	**58.42** [58.42, **58.42**]
gestaohospital	48.28 [42.75, **55.64**]	**52.45** [49.48, 54.12]
languagetool	**26.49** [26.46, 26.54]	21.22 [20.21, **27.55**]
restcountries	74.73 [71.82, 75.51]	**75.1** [74.4, **75.69**]
Average	45.14	**47.01**

In Table 6.2, we compare EvoReFuzz's performance with EvoMaster, one at a time on each SUT, and report the p-values of the Mann-Whitney-Wilcoxon U-Test one-sided, in which the distribution underlying the number of covered lines of EvoReFuzz is stochastically greater than the EvoMaster. Moreover, the stated **hypotheses** are the following:

- Null Hypothesis H_0: There is **no** difference between the ranks of EvoReFuzz and EvoMaster.
- Alternative Hypothesis H_1: There **is** a difference between the ranks, and the distribution underlying the **EvoReFuzz is stochastically greater** than the EvoMaster.

Table 6.2: Mann-Whitney-Wilcoxon U-Test one-sided of **EvoReFuzz**'s results compared to EvoMaster. Values lower than the $\alpha = 0.05$ threshold are reported in bold.

SUT	P-value	Effect size
catwatch	**0.000**	1.335 (large)
cwa-verification-server	0.982	-
features-service	1.000	-
gestaohospital	0.979	-
languagetool	**0.001**	0.969 (large)
restcountries	0.732	-

From the results of Table 6.2, we can reject the Null Hypothesis with a significance level of 0.05, and, therefore, the underlying distribution of EvoReFuzz is stochastically greater than EvoMaster for the *catwatch* and *languagetool* APIs with a large effect size. Whereas, from Table 6.3, we can reject the Null Hypothesis with a significance level of 0.05, meaning that the underlying distribution of EvoMaster is stochastically greater than EvoReFuzz for the *cwa-verification-server*, *features-service*, and *gestaohospital* APIs. Consequently, for the *restcountries* SUT, we cannot prove that either EvoReFuzz or EvoMaster has a stochastically greater distribution.

Table 6.3: Mann-Whitney-Wilcoxon U-Test one-sided of **EvoMaster**'s results compared to EvoReFuzz. Values lower than the $\alpha = 0.05$ threshold are reported in bold.

SUT	P-value	Effect size
catwatch	1.0	-
cwa-verification-server	**0.022**	0.636 (large)
features-service	**0.000**	1.289 (large)
gestaohospital	**0.025**	0.622 (large)
languagetool	0.999	-
restcountries	0.294	-

Looking at the experimental results for the **catwach** API, EvoReFuzz outperformed EvoMaster in line coverage. However, we identified a significant problem while analyzing the logic behind the SUT. When a request is made, the SUT makes a call to an outside service, specifically the GitHub APIs, to obtain project information. In the process, the call appears to be stuck for a while (and may time out), which prevents the code responsible for interpreting the answers from running. Since external services can change or go down at any time and deliver different data with each call, calling them is a difficult task for the current fuzzers.

Regarding the **cwa-verification-server**, EvoMaster had a higher average percentage but only by 0.9%, which translates to 307 missed lines out of 620 lines that composed the SUT compared to 313 from EvoReFuzz. An average difference of 6 missed lines between the two fuzzers. Nonetheless, six lines may be the difference in finding a bug in the code. Moreover, there is an intriguing observation to discuss, which was made by Zhang et al. [132]. The input in the endpoint handler InternalTanController can be an HTTP header with the value TELE_TAN_TYPE_HEADER="X-CWA-TELETAN-TYPE". However, the OpenAPI schema does not contain this information, resulting in the object teleTan-Type always being null. This is an illustration of underspecified schema, or in other words, a lack of specification elements in the swagger/OpenAPI file.

In the **features-service**, we can see a significant variation of code coverage between Evo-Master and EvoReFuzz. EvoMaster had close to 16% higher line coverage than EvoRe-Fuzz. Futhermore, we could not look for the output data since running the tools in a loop made it impossible to control the writing of the output information of each tool in different folders (at least in our time window). Nonetheless, there are, for instance, getters and setters that will never be callout and some functions that are only used by the manually written unit tests. Additionally, the API has an operation dependency where a POST should create specific data prior to a GET operation fetching it. This particularity is a prominent factor for the MIO algorithm from EvoMaster to outperform our evolutionary approach. Zhang et al. [127] implemented a set of effective templates on top of MIO to structure test actions based on the semantics of HTTP methods used to manipulate the web services' resources. This technique allows the calling of POST-based operations before GET-based ones and then uses such previous data to generate valid requests. Consequently, since the features-service API is based primarily on the management of available resources in the API, this approach allied with a 1 + 1 EA (e.g., MIO algorithm) will efficiently outperform our approach. This conclusion is a fundamental observation to note as future work to be developed in our Evolutionary Algorithm.

In the experimental results of the **gestaohospital** API, even though EvoMaster had a higher average of line coverage, we must point out that the maximum code coverage achieved was conducted by EvoReFuzz with a 55.64% against 54.12% of EvoMaster. Moreover, this API heavily depends on interactions between the database, and not much coverage is achieved if such accessed data is not present in the database.

Languagetool is a CPU-bound application (e.g., no database) executing complex text computations. This API has only two endpoints: /v2/languages, which is a simple GET with no parameters; therefore, it is trivially covered by just calling it onetime, and /v2/check, which is a POST with 11 input parameters. From the set of state-of-the-art tools, such as bBOXRT, RESTct, RESTler, RESTest, and RESTTESTGEN, only EvoMaster can generate requests with payloads of nature application/x-www-form-urlencoded [132]. This is a significant advantage since the more endpoints tested, the easier it should be to achieve a higher code coverage. We notice that, even though the Languagetool service accepts application/x-www-form-urlencoded, it also can receive parameters in the query location for its main operation (i.e., /v2/check). In this note, we changed the OpenAPI only for the /v2/check operation in order for EvoReFuzz to be capable of generating requests.

Moreover, interestingly we achieved a higher average line coverage than EvoMaster. We should emphasize that EvoMaster does not have any type of parameter value configuration besides the headers to filter the value for authentication parameters (i.e., RESTful

API Authentication methods). For these, EvoMaster lets set a value through the command line options. EvoReFuzz, on the other hand, is capable of setting values in all locations possible where parameters reside (i.e., query, path, header, and body). We took advantage of it and we filtered the value of the language parameter (i.e., en, pt, fr, en-US). In a way, it is unfair, however, from the point of view of being so straightforward to set values that otherwise, the lack of such would result in lower code coverage, made us test it in this environment. Additionally, every automated black-box tool should have at least some type of parameter value customization. Therefore, a framework capable of producing these features would be ideal for future work. Regarding the code coverage results for languagetool, we should note that without altering the OpenAPI, EvoReFuzz would not have achieved values remotely close to the ones in Table 6.1 since it is incapable of building requests with payloads of nature application/x-www-form-urlencoded. Furthermore, similar to the case observed in the gestaohospital API where the maximum value of line coverage was achieved by EvoReFuzz even though it did not have the highest average line coverage, it occurred in languagetool where EvoMaster has the maximum line coverage with 27.55%.

Concerning the **restcountries** SUT, high coverage is achieved with an average of 74.73% for EvoReFuzz and 75.1% for EvoMaster. Therefore, EvoMaster missed, on average, 135 lines out of 543 that composed the restcountries API, whereas EvoReFuzz missed 137. Hence, the difference between the two fuzzers is two lines of coverage. Additionally, like the other SUTs, there is a significant amount of dead code as a result of getters and setters that are never called and catch blocks for potential non-throwable exceptions. On the other hand, the API has an operation that returns a list of countries based on different filtering criteria, such as country codes. However, the response with NOT_FOUND is never returned. The problem is that, even if the HTTP requests provide invalid inputs (e.g., a country code that does not exist), the countries list is incorrectly populated with null values, so the list will never be empty. This is an intriguing API bug, which results in a crash (i.e., a returned 500 status code). Consequently, these NOT_FOUND statements are essentially dead code that tests cannot reach until this API bug is fixed.

Public services Experimental Results

In the public 11 real-world RESTful APIs tested, we found major robustness problems in both GitLab and Microsoft Bing Maps services. More than 360,000 requests were sent to the GitLab APIs and over 100,000 to the Microsoft Bing Maps services. In the final, EvoReFuzz detects 28 new bugs (i.e., robustness problems) in the subject APIs, where only eight of them can be triggered by RESTCT, and just one by RESTler. Table 6.4, demonstrate the results obtained in our experiences.

In Table 6.4, the cost is the time, in **minutes**, each tool spent testing each service individually. Even though EvoReFuzz takes more time to perform, it uses both workload and faultload phases. Moreover, most of the found bugs were triggered in the fautload phase, which means that we could produce the same results with half of the requests and most likely half of the time. Regarding robustness problems, we consider a request a bug if it results in a response with status code 500. Additionally, a unique bug is every bug found in one operation. Therefore, multiple requests producing responses with status code 500 in one operation will only count as one unique bug.

Table 6.4: Public services results - RESTler and RESTct values were based of [140]

		RESTler		RESTCT		EvoReFuzz	
Company	APIs	Bug	Cost	Bug	Cost	Bug	Cost
GitLab	Branch	0	23.3	0	2.3	0	12
	Commit	0	23.4	0	4.4	0	8
	Groups	1	0.5	1	5.2	**1**	16
	Issues	0	60.6	0	27	0	20
	Project	0	24.3	0	25.5	0	27
	Repository	0	23.1	0	2.3	0	8
Microsoft Bing Maps	Elevations	0	0.2	0	1.3	**4**	10
	Imagery	0	60.2	4	55.2	**10**	33
	Locations	0	61.4	0	3.8	**2**	12
	Route	0	60.8	3	63.2	**9**	292
	TimeZone	0	1	0	28.6	**2**	27
	Average	0.1	30.8	0.7	19.9	**2.5**	42.3

From the six APIs under **GitLab**, we could only find a robustness problem in Groups API. Overall, these services are well implemented, either responding with status code 200 for valid requests, 400 for invalid ones, or 404 for not found resources. Even in the presence of multiple injected faults these APIs stayed robust without triggering any major bug. Except the Groups API, where the operation GET /groups had multiple ***500 - Internal server error***. In order to contextualize, this operation gets a list of visible groups for the authenticated user, and only public groups are returned when accessed without authentication. In this particular case, numerous faults can trigger a bug. For instance, replacing the parameter page (i.e., the page number of the returned list) with the Maximum value plus 1 (i.e., 9223372036854775808) of the numeric data type, which in this case is an integer, produced a bug. There are also two types of faults that can easily trigger robustness problems in this operation, Replace with null and Replace with an empty value, which may indicate that parameter verification is not being conducted in the code. Consequently, in the faultload phase, we had a total of 1274 different requests that originated bugs over the /groups endpoint (i.e., operation). We found it peculiar since the other endpoints and the subsequent APIs from GitLab stayed consistent and robust thought out the experiences.

Regarding the **Microsoft Bing Maps APIs**, the Bing's developer center register over 34,000 requests while testing these public APIs. Figure 6.1 shows a pie chart representing the overall testing usage divided into the primary operations available in the services. The data labels are composed of the operation's name, how many requests were sent for that particular operation, and the respective deducted percentage.

In every API, we found at least two unique bugs, wherein two of them were discovered a cumulative of 19 robustness problems. Even though these services are complex and perform complex tasks, by finding a total of 27 unique robustness problems, we may affirm that these services lack some type of validation in their parameters. We cannot show what did trigger these problems since the responses disclaim such data as protection against possible security breaches. Nonetheless, we may affirm that further testing should be conducted by the Microsoft testers, where server logs may be checked to locate the

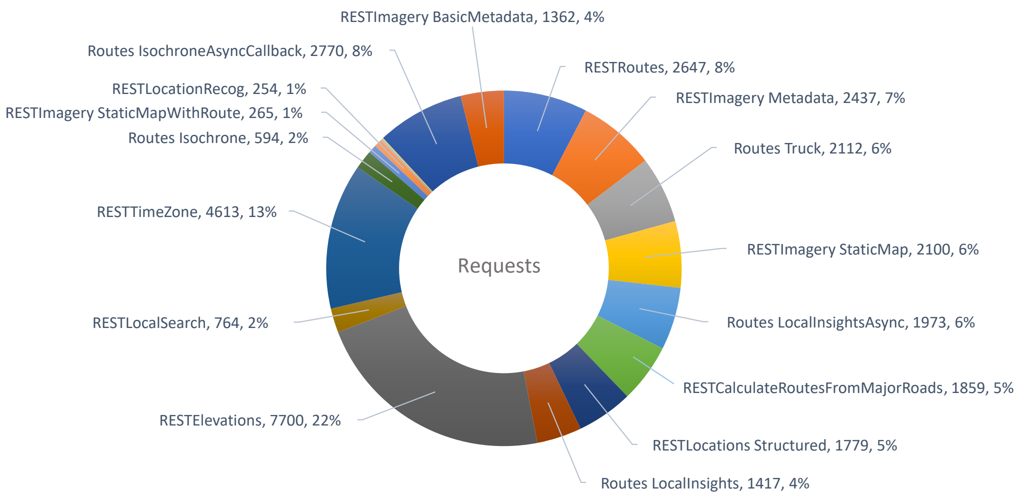


Figure 6.1: Microsoft Bing Maps usage

weakness in the present code's logic.

Table 6.5 shows the overall distribution by endpoints of the robustness problems found in the Microsoft Bing Maps APIs. From the 17293 requests that originate a response with status code 500, we triggered 27 unique bugs. However, in this set of 17293 requests, we could have also triggered different types of robustness problems that cannot be pinpointed since we do not have access to the server logs. Therefore, the identified number of unique robustness problems are correlated to the endpoints in which at least one request with status code 500 is present.

Table 6.5: Public Services results with unique bugs

API	Endpoint		HTTP method	Total Number of requests with status code 500	Unique bugs	Total requests sent to the API
Elevations	/Elevation	/SeaLevel /List /Bounds /Polyline	GET	4256	4	8240

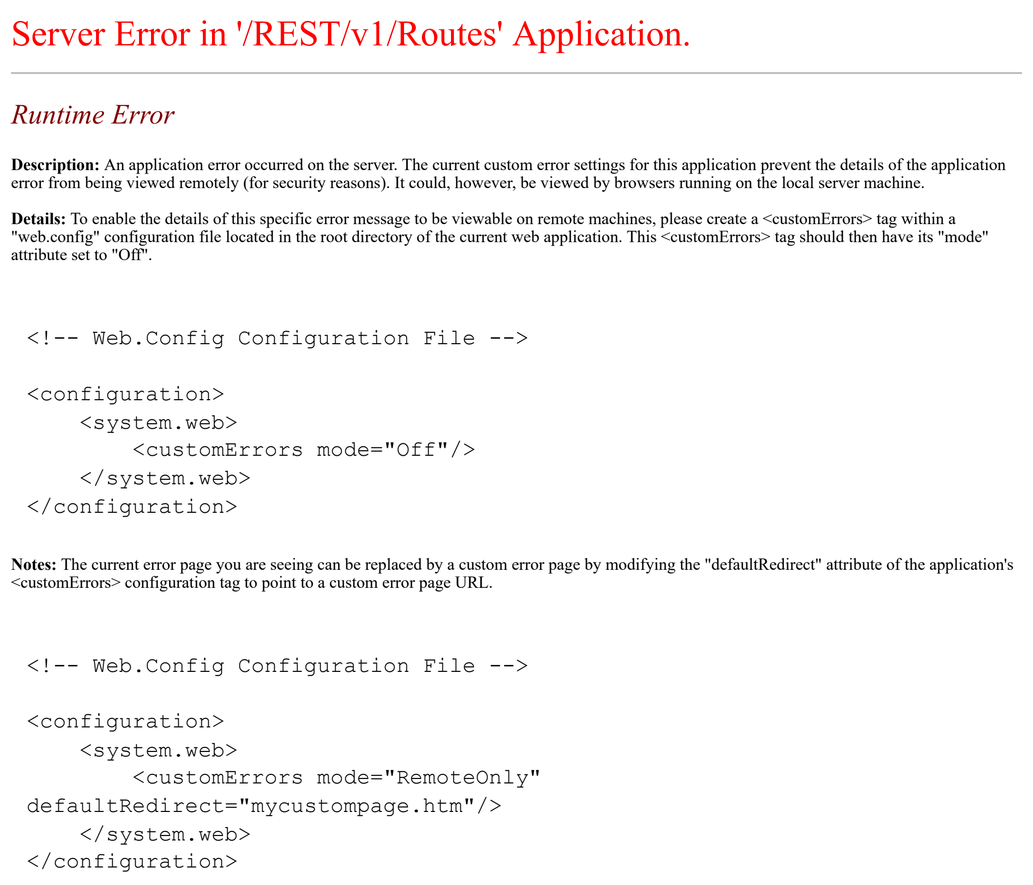
Imagery	/Imagery/Map	/{imagerySet}
		/{imagerySet}/{query}
		/{imagerySet}/Routes/{travelMode}
		/Streetside/{address}
		/Streetside/{centerPoint}/{zoomLevel}
		/{imagerySet}/{centerPoint}/{zoomLevel}
		/{imagerySet}/{centerPoint}/{zoomLevel}/Routes/{travelMode}
			GET	4831	10	20600

	/Imagery/BasicMetadata	/{imagerySet}/{centerPoint}				

	/Imagery/Metadata	/{imagerySet}
		/{imagerySet}/{centerPoint}				

Locations	/LocalSearch	/	GET	1127	2	6180
	/LocationRecog	/{point}				
Routes	/Routes	/ /{travelMode} /Truck /Isochrones /IsochronesAsyncCallback /LocalInsights /LocalInsightsAsync /FromMajorRoads	GET	5150	9	58240
		/DistanceMatrixAsync /OptimizeItinerary	POST			
TimeZone	/TimeZone	/ /Convert /{point}	GET	1929	2	8240
			Total:	17293	27	101500

Concerning the error messages for the internal server error, a vast majority of responses



Server Error in '/REST/v1/Routes' Application.

Runtime Error

Description: An application error occurred on the server. The current custom error settings for this application prevent the details of the application error from being viewed remotely (for security reasons). It could, however, be viewed by browsers running on the local server machine.

Details: To enable the details of this specific error message to be viewable on remote machines, please create a <customErrors> tag within a "web.config" configuration file located in the root directory of the current web application. This <customErrors> tag should then have its "mode" attribute set to "Off".

```
<!-- Web.Config Configuration File -->

<configuration>
    <system.web>
        <customErrors mode="Off"/>
    </system.web>
</configuration>
```

Notes: The current error page you are seeing can be replaced by a custom error page by modifying the "defaultRedirect" attribute of the application's <customErrors> configuration tag to point to a custom error page URL.

```
<!-- Web.Config Configuration File -->

<configuration>
    <system.web>
        <customErrors mode="RemoteOnly"
defaultRedirect="mycustompage.htm"/>
    </system.web>
</configuration>
```

Figure 6.2: Run time error message

with status code 500 had the error message of **RunTime Error** which is proof of a bug in the implemented code. Figure 6.2 shows the RunTime error message returned by the service that appears consistently across all APIs and is triggered several times. Moreover, we also find another type of message for the internal server error. Listing 6.1 shows the generic message returned by the API where it is also current in every API, and the traceId is different for each one of the occurrences. When we look up both messages, the Runtime error message (Figure 6.2) occurs considerably more often than the generic message in a JSON body (Listing 6.1).

Listing 6.1: Generic Error message

```
{
  "authenticationResultCode": "ValidCredentials",
  "brandLogoUri": "http://dev.virtualearth.net/Branding/logo_powered_by.png",
  "copyright": "Copyright 2022 Microsoft and its suppliers. All rights reserved.
      This API cannot be accessed and the content and any results may not be used
      , reproduced or transmitted in any manner without express written
      permission from Microsoft Corporation.",
  "errorDetails": [
    "Your request could not be completed because there was a problem with the
       service."
  ],
  "resourceSets": [],
  "statusCode": 500,
  "statusDescription": "Internal Server Error",
  "traceId": "4609027139db4a1f9f4af01211355833|DU00002742|0.0.0.0|DU000005E8,
    DU00001FE3|Ref A: 21A23F743F6E4A3AA5D34D9CDBC96E10 Ref B: DB3EDGE2120 Ref C
    : 2022-08-13T01:46:44Z"
}
```

It is undoubtedly intriguing that the APIs have a generic message (Listing 6.1) for internal server errors but also return an HTML message for the RunTime error. Even though the generic message is related to an internal error, it appears to be a more controlled manner of yielding an answer to the client. Whereas, for the RunTime error, the server unexpectedly stopped its operation. Nonetheless, having two types of messages different from each other for an internal error, let us deduct that we are reaching different parts of the code and, in a way, having a higher code coverage.

An exciting conclusion we come up with, looking at the results, is which requests Microsoft count as billable and non-billable, or in other words, which requests are registered in the Microsoft developer center [180]. We deduced that requests producing responses with status codes 500 would not be registered in the developer center. With this finding, we notice a discrepancy between the values demonstrated in Figure 6.1 showing over 34,000 registered requests, and Table 6.5, which demonstrates a total of 101,500 requests sent to the server. Of the 101,500 requests, only 17,293 requests were not counted since they resulted in responses with status code 500. Thus, only 84,207 are valid for registration in the developer center, although just over 34,000 were indeed reported. Accordingly, we can conclude that we duplicated many individuals throughout the generations and used a low mutation rate. Furthermore, due to the fact that these services have a large number of input parameters, a mutation probability $1.0/number\,of\,parameters$ is, in fact, a low mutation rate.

ucXception's framework API Experimental Results

To test the ucXception API, we did pair testing but remotely. The student started the API, and we conducted the testing with EvoReFuzz. Both parties analyzed the behavior of the API, and we helped with the correct conventional implementation that should be accomplished in RESTful APIs.

Regarding the results, we found some interesting yet common robustness problems in the ucXception's API and common implementation errors. Table 6.6 demonstrates the errors found.

We manually analyzed the output from the test and essentially looked for operations that ended with a status code of 500. We found robustness problems in four different operations:

- GET /campaign/{campaign_id}/download
- GET /components/{component_type}
- GET /components/{component_type}/{component_choice}
- GET /components/{component_type}/{component_choice}/{campaign_id}

Regarding these operations, they are in charge of validating a campaign id and retrieving a CSV file, getting the names of all available components in ucXeption (i.e., logger

Table 6.6: List of encountered problems

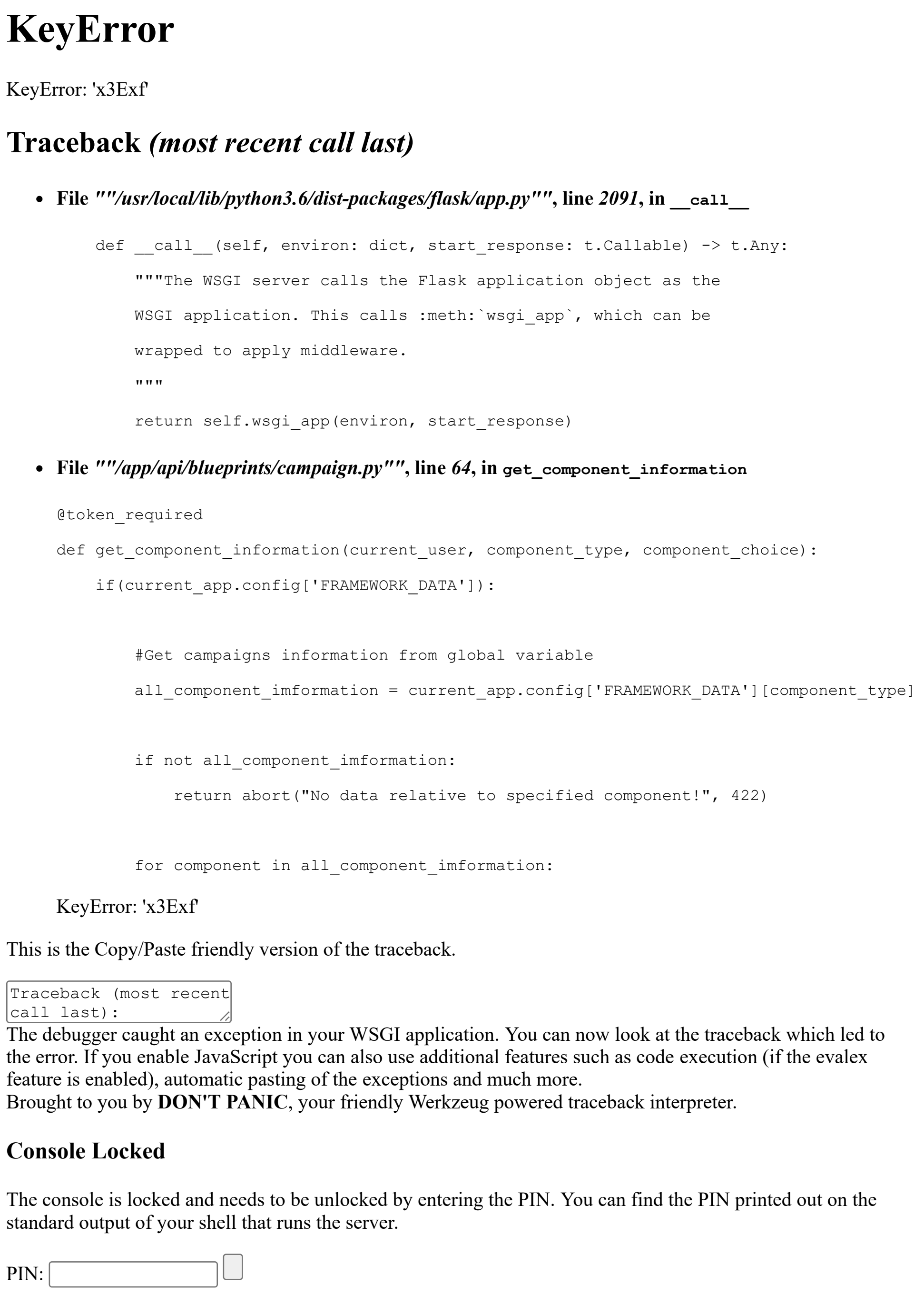
Problem	**Type**
Missing verification if the campaign with the given id existed in the database.	Common missing verification found, possible future problems
Missing verification if the campaign with the given id belonged to the user.	Common implementation error found, possible future problems
Missing validation (syntax) of the campaign id, validation if it is an integer.	Common missing validation found, possible wrong feedback in the frontend module
Lack of validation for positive number.	Common missing validation found
Always returned status code 200 regardless of whether the element existed in the database.	Bad practice, possible wrong feedback in the frontend module
Use of body in delete method is not good practice.	Bad practice
Missing validation of the user email.	Common implementation error found
Missing validation and verification of the path parameters.	**Resulted in robustness problems**

analyzer - Probe logs), extrapolating data from a single component, and validating path parameters to obtain only the allowed components already created. In this note, the bugs discovered in these operations were critical to bulletproofing the frontend module of possible flaws. Consequently, the robustness problems were triggered whenever the API tried to access a parameter that did not exist and had no protection. Furthermore, the operations that produced the status code for the 2xx group (i.e., 200 to 299) were also examined to determine whether a validation or verification was missing. Here, it was possible to detect some errors that could lead to potential future problems in the ucXeption framework. For instance, looking at Table 6.6, the most typical problems encountered were the missing validation of path parameters and body parameters and the missing verification of the existing elements in the database.

In Figure 6.3, a generic Traceback message from the server is sent when a major robustness problem is triggered without any type of handling. The results showed a total of four different Traceback messages that occurred multiple times. They are correlated with accessing a not-existent dictionary key, resulting in a **key error in Python** as demonstrated in the Figure 6.3. Therefore, we determined four distinct locations where such an error existed in the code.

While building the OpenAPI specification file and testing the REST service, some bad practices were in place. For example, some operations returning only the status code 200 regardless of the code's logic, potentially leading the frontend to present wrong messages to the user. Consequently, it can be considered a bug in the frontend module. Moreover, a delete operation accepted a JSON body. Even though such can be implemented, it is however bad practice in the conventional world of RESTful APIs. We pinpointed that to the student, and changes were made to accomplish the conventional methods.

We can conclude that since the frontend module is significantly dependent on the responses gotten from the API, this type of validation is essential to extrapolate some significant errors that otherwise would not be discovered and could result in several cumulative bugs that would originate in a poor experience to the user of the ucXeption framework. In the end, all the detected errors were corrected not to compromise the API's function and the subsequent frontend module. This partnership was also advantageous for both parties since, on the one hand, we can show the usefulness of our tool for developers to validate their systems and the student that did a well-structured validation of the implemented API in the dissertation.



KeyError

KeyError: 'x3Exf'

Traceback *(most recent call last)*

- **File *""/usr/local/lib/python3.6/dist-packages/flask/app.py""*, line *2091*, in `__call__`**

```
    def __call__(self, environ: dict, start_response: t.Callable) -> t.Any:
        """The WSGI server calls the Flask application object as the
        WSGI application. This calls :meth:`wsgi_app`, which can be
        wrapped to apply middleware.
        """
        return self.wsgi_app(environ, start_response)
```

- **File *""/app/api/blueprints/campaign.py""*, line *64*, in `get_component_information`**

```
@token_required
def get_component_information(current_user, component_type, component_choice):
    if(current_app.config['FRAMEWORK_DATA']):

        #Get campaigns information from global variable
        all_component_imformation = current_app.config['FRAMEWORK_DATA'][component_type]

        if not all_component_imformation:
            return abort("No data relative to specified component!", 422)

        for component in all_component_imformation:
```

KeyError: 'x3Exf'

This is the Copy/Paste friendly version of the traceback.

```
Traceback (most recent
call last):
```

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error. If you enable JavaScript you can also use additional features such as code execution (if the evalex feature is enabled), automatic pasting of the exceptions and much more.
Brought to you by **DON'T PANIC**, your friendly Werkzeug powered traceback interpreter.

Console Locked

The console is locked and needs to be unlocked by entering the PIN. You can find the PIN printed out on the standard output of your shell that runs the server.

PIN:

Figure 6.3: Generic Traceback message for robustness problems

6.2 Main findings

In this section, we summarize the main findings of our experimental evaluation by going through crucial characteristics we noticed during this endeavor. We enumerate them in the following list:

- In most systems, it is unrealistic to target a 100% code coverage since it is almost impossible to achieve it in complex systems (or even 90%) due to unreachable code or getters/setters that will not be utilized.
- Schemas in the OpenAPI file, like software, might include faults and/or omissions (e.g., constraints on some inputs might be missing).
- Underspecified schemas are correlated to a lower code coverage when testing RESTful APIs. For instance, in the logic underlying the service, there might be parameters that are not specified in the OpenAPI/swagger file.
- A tool capable of dealing with inter-parameters and operations dependencies will achieve a higher code coverage for APIs heavily dependent on databases. Since, in the context of these services, the main logic is writing and reading to/from the database, dealing with possible dependencies in the input data will make it possible to cover all possible scenarios.
- A tool must be capable of generating requests to any type of possible payloads (i.e., application/json, form-data, x-www-form-urlencoded, XML) to achieve a higher code coverage since higher operation coverage will translate to a higher code coverage of the SUT.
- Parameters value customization is a must for every black-box fuzzer. When testing a RESTful API, there are times in which the developer/tester knows possible values to specific parameters where these are crucial to producing, for instance, a response with status code 200. As a result, the capability of allowing the customization of such values is a fundamental feature that should be implemented in the fuzzer. For example, in most RESTful APIs, authorization parameters should be set with a basic key or token to allow the user (i.e., tool) to send authorized requests.
- Real-world services are being deployed holding software bugs, or in other words, robustness problems. Consequently, their overall availability is affected when confronted with faulty scenarios, or in other words, unexpected inputs.
- Lack of parameter validation is one of the most common implementation errors (e.g., not verifying if the input parameter for an *ID* in the database is a positive integer).
- Bad practices while specifying the OpenAPI files are widespread (e.g., a delete operation with a payload).
- Messages returned by the system under test when robustness problems occur may lead to vulnerability breaches. The responses' content originated from a robustness problem may hint to the attacker to possible weaknesses in the service.

- When an API is used as middleware for a frontend framework, validating such a system will bulletproof the frontend component from potential misleading bugs. For instance, if the RESTful API only returns status code 200, then the frontend component will wrongly give feedback to possible operations that were not executed correctly (i.e., feedback that an entry was created in the database, but no data was added).
- A request that originated a robustness problem appears to stay consistent over time. In other words, the exact requests will consecutively originate the same robustness problem.
- Every OpenAPI/swagger file analyzed had a poor expected response specification, especially for the status code 500, which describes the possible message for the originated internal server error.
- With a trivial configuration and one hour of testing, the testers, developers, and practitioners will have a good test bench to validate their systems.

Regarding our evolutionary approach, EvoReFuzz, we found that it achieved close code coverage compared to the state-of-the-art tool EvoMaster. EvoReFuzz also efficiently was capable of triggering 28 unique bugs in a total of 11 public APIs deployed by Microsoft and GitLab, whereas RESTct found eight and RESTler only one. Such robustness problems are correlated to the lack of verification and validation before deploying these services, which affects their general availability. Moreover, with an easy testing setup and a short test time, we found, together with the Masters's student, that EvoReFuzz is a valuable tool for discovering bugs and potential future ones that would affect the overall robustness of a system's architecture.

We also noticed that duplicate requests sent to the service might affect the results in the code coverage of the system and in the process of finding robustness problems. On this note, running a small test bench and then looking for the number of duplicates is helpful to decide if we change, for instance, the mutation rate or the tournament selection pool, reducing the convergence speed of the GA in our approach. Consequently, future techniques to avoid duplicated requests are practical applications for future work.

Chapter 7

Conclusion

Nowadays, the Internet is present in every aspect of our lives, from simple websites to complex software systems available in the palm of our hands, such as Google, home banking, and entertainment, to mention a few. In these software systems, constant availability is an essential non-functional requirement. As such, these services must operate correctly by behaving robustly in the presence of invalid input or stressful conditions that may not be favorable and compromise the service itself. In addition, RESTful web services are commonly used in such software systems and are based on a relatively loose architectural style, allowing for potential weaknesses in the service that may result in robustness issues.

In this work, we present a state-of-the-art analysis of Evolutionary Algorithms and software testing with a particular focus on the current implemented tools for testing REST services in the literature. In addition, we propose an Evolutionary approach to test the robustness of RESTful APIs as a solution to the existing gap in the literature regarding evolutionary techniques to evaluate these services.

Concerning the current techniques in the literature, we designed a proof of concept based on a Genetic Algorithm, with the name EvoReFuzz, where the requests are individuals composed of a fitness function that will guide the genetic algorithm towards better solutions according to the problem being solved, such as, the generation of valid and invalid requests. We implemented different strategies for applying the operator crossover and mutation to complex requests. Moreover, we subsequently show EvoReFuzz's capabilities in the form of a practical experiment, running thousands of tests over 18 REST services divided into three sets (i.e., real-world, In-house, and private). The outcomes showed the approach's capability to evaluate a diverse gamma of services while revealing robustness problems as well as poor implementation techniques in the tested services.

Regarding the **threats to the validity** of the approach implementation and evaluation, for the experiments conducted between EvoReFuzz and EvoMaster, the command-line options chosen to test the six APIs might have affected the results of both approaches. However, we did choose these options with the best possible outcome in our mind, at least to the best of our knowledge.

For the extensive testing of the real-world services, our approach's parameterizable options could have led to fewer robustness problems in these services, especially in GitLab's APIs. We did not consider a different mutation rate, which could have been advantageous in this case. Nonetheless, we did find a large number of unique bugs considering the total number of available operations.

While testing the ucXception's API (private service), we ran two phases of testing, in which more than 50,000 requests were sent in total. In the last run, of the 34,000 requests, we found four unique bugs and multiple occurrences of the status code 500. Consequently, some unique bug might have escaped in the manual process of analyzing the output of the large test bench. Regardless, the number of disclosed issues, such as bad practices in the OpenAPI specification file and robustness problems in the service, provide the developers with helpful information for robustness and validation appraisal. Moreover, we manually analyzed and reproduced the four bugs found to confirm their existence.

Regarding the software bugs of the implemented approach, it might have affected some results and, therefore, the subsequent observation related to those same results. We tried, however, to detect any possible bug the fastest we could. Additionally, we tested a diverse set of operations, where each API from the set of real-world services was exceedingly complex and worked as a debug method to assess our approach's code robustness.

In **future work**, we intend to implement machine learning capable of categorizing robustness problems from non-robustness problems based on the response's content and status code. In this note, after a neural network has been trained, it is readily incorporated into the code responsible for assessing each individual's fitness function. With such in mind, we developed the fitness function code to be easy for future extensibility. Furthermore, implementing a technique capable of resolving operations dependencies that follows practical templates based on the semantics of HTTP methods to operate over the web services' resources. For instance, a POST operation is executed, and the generated parameters' values are used for the subsequent GET operation. As a result, such logic allows the generation of valid requests by reusing the parameters' values and, therefore, achieving a higher code coverage. Moreover, another aspect that should be mentioned is a framework for a graphical interface to facilitate the customization of the parameter values and the overall approach's usability.

References

[1] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. A systematic review on software robustness assessment. *ACM Comput. Surv.*, 54(4), may 2021.

[2] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 14(4):957–970, 2021.

[3] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[4] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[5] Leonard Richardson, Mike Amundsen, Michael Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[6] Smartbear software. openapi specification version 3.0.3. `https://swagger.io/specification/`, Accessed: 2021-12-29.

[7] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[8] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. A black box tool for robustness testing of rest services. *IEEE Access*, 9:24738–24754, 2021.

[9] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: Automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, 2020.

[10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019.

[11] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: Black-box constraint-based testing of restful web apis. pages 459–475, 12 2020.

[12] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, 2020.

[13] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for rest apis: A specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–190, 2018.

[14] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[15] Man Xiao, Mohamed El-Attar, Marek Reformat, and James Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

[16] Nigel James Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, Citeseer, 2000.

[17] Gitlab apis documentation. Available: `https://docs.gitlab.com/ee/api/`, Accessed: August 2022 [Online].

[18] Microsoft. Microsoft bing maps rest services documentation. Available: `https://docs.microsoft.com/en-us/bingmaps/rest-services/`, Accessed: August 2022 [Online].

[19] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397, 2018.

[20] Carlos Santos, Nuno Laranjeiro, and Nuno Lourenço. Evolutionary rest fuzzer. `https://git.dei.uc.pt/cnl/bBOXRT/tree/master`, 2022.

[21] EMResearch. Evomaster github. `https://docs.microsoft.com/en-us/bingmaps/rest-services/imagery/`, Accessed: November 2021.

[22] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[23] Shivani Acharya and Vidhi Pandya. Bridge between black box and white box–gray box testing technique. *International Journal of Electronics and Computer Science Engineering*, 2(1):175–185, 2012.

[24] Mohd Ehmer Khan et al. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–14, 2011.

[25] Mohd Khan et al. Different approaches to black box testing technique for finding errors. *International Journal of Software Engineering & Applications (IJSEA)*, 2(4), 2011.

[26] Syed Roohullah Jan, Syed Tauhid Ullah Shah, Zia Ullah Johar, Yasin Shah, and Fazlullah Khan. An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN*, 23951990, 2016.

[27] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[28] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[29] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[30] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. 2012.

[31] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[32] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. Robustness testing techniques and tools. In *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer, 2012.

[33] Algirdas Avizienis, J.-C. Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[34] Http response status codes. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`, Accessed: 2021-01-19.

[35] Yaml. `https://yaml.org/`, Accessed: 2022.

[36] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[37] M Gudgin, M Hadley, N Mendelsohn, Y Lafon, JJ Moreau, A Karmarkar, and HF Nielsen. Soap version 1.2 part 1: Messaging framework, 2nd edn. w3c recommendation, w3c, retrieved september 24, 2012, 2007.

[38] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[39] Jérôme Loisel. Soap vs rest. `https://octoperf.com/blog/2018/03/26/soap-vs-rest/#comparison-table`, Accessed: July 2022.

[40] Snehal Mumbaikar, Puja Padiya, et al. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5):1–4, 2013.

[41] Google. Google trends - rest vs soap. `https://trends.google.com/trends/explore?cat=5&date=all&q=REST,SOAP`, Accessed: 21-08-2022.

[42] Charles Darwin's. On the origin of species. *published on*, 24:1, 1859.

[43] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[44] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[45] Gentiana Ioana Latiu, Octavian Augustin Cret, and Lucia Vacariu. Automatic test data generation for software path testing using evolutionary algorithms. In *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, pages 1–8. IEEE, 2012.

[46] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[47] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, 43(14):817–831, 2001.

[48] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[49] Eberhart and Yuhui Shi. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 1, pages 81–86 vol. 1, 2001.

[50] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, Dec 1997.

[51] Rene Thomsen. Multimodal optimization using crowding-based differential evolution. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, volume 2, pages 1382–1389 Vol.2, 2004.

[52] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[53] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the optimization of unimodal functions with the (1+1) evolutionary algorithm. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN V*, pages 13–22, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[54] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[55] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* University of Michigan, 1975.

[56] James David Schaffer, Rich Caruana, Larry J. Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, page 51–60, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[57] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[58] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Accounting for noise in the sizing of populations* *portions of this paper are excerpted from a paper by the authors entitled "genetic algorithms, noise, and the sizing of populations" (goldberg, deb, & clark, 1991). In L. DARRELL WHITLEY, editor, *Foundations of Genetic Algorithms*, volume 2 of *Foundations of Genetic Algorithms*, pages 127–140. Elsevier, 1993.

[59] Andrew Tuson and Peter Ross. Cost based operator rate adaptation: An investigation. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN IV*, pages 461–469, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[60] William M Spears et al. Adapting crossover in evolutionary algorithms. In *Evolutionary programming*, pages 367–384, 1995.

[61] Mandavilli Srinivas and Lalit M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[62] Ágoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[63] Zbigniew Michalewicz and Marc Schoenauer. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation*, 4(1):1–32, 03 1996.

[64] Abdollah Homaifar, Charlene X. Qi, and Steven H. Lai. Constrained optimization via genetic algorithms. *SIMULATION*, 62(4):242–253, 1994.

[65] Jeffrey A. Joines and Christopher R. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with ga's. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 579–584 vol.2, 1994.

[66] Heinz Muhlenbein. How genetic algorithms really work: I. mutation and hill-climbing. In *Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, 1992*. Elsevier, 1992.

[67] Jim Smith and Terence C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 318–323, 1996.

[68] Jürgen Hesser and Reinhard Männer. Towards an optimal mutation probability for genetic algorithms. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, pages 23–32, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[69] Bryant A. Julstrom. Adaptive operator probabilities in a genetic algorithm that applies three operators. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, SAC '97, page 233–238, New York, NY, USA, 1997. Association for Computing Machinery.

[70] David B Fogel. Evolutionary algorithms in theory and practice, 1997.

[71] Thomas Bäck. Self-adaptation in genetic algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press, 1992.

[72] Thomas Bäck. Optimal mutation rates in genetic search. In *Proceedings of the fifth international conference on genetic algorithms*. Citeseer, 1993.

[73] Thomas Back. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In *Proc. 2nd Conference of Parallel Problem Solving from Nature, 1992*. Elsevier Science Publishers, 1992.

[74] Joanna Lis. Genetic algorithm with the dynamic probability of mutation in the classification problem. *Pattern Recognition Letters*, 16(12):1311–1320, 1995.

[75] James Edward Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, volume 1. Hillsdale, New Jersey, 1985.

[76] Anne Brindle. Genetic algorithms for function optimization. 1980.

[77] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69–93. Elsevier, 1991.

[78] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the world congress on engineering*, volume 2, pages 1–6. International Association of Engineers Hong Kong, 2011.

[79] Daniel Joseph Cavicchio. Adaptive search using simulated evolution. Technical report, 1970.

[80] David E Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex systems*, 3(5):493–530, 1989.

[81] Seiichi Koakutsu, Maggie Kang, and Wayne Wei-Ming Dai. *Genetic simulated annealing and application to non-slicing floorplan design*. Citeseer, 1995.

[82] Jian-Ping Li, Marton E. Balazs, Geoffrey T. Parks, and P. John Clarkson. A Species Conserving Genetic Algorithm for Multimodal Function Optimization. *Evolutionary Computation*, 10(3):207–234, 09 2002.

[83] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 155–164, New York, NY, USA, 2007. Association for Computing Machinery.

[84] Nashat Mansour and Miran Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–136, 2004.

[85] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[86] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 13–24, New York, NY, USA, 2006. Association for Computing Machinery.

[87] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software testing, verification and reliability*, 9(4):263–282, 1999.

[88] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and software technology*, 43(14):841–854, 2001.

[89] Jürgen Hesser and Reinhard Männer. Towards an optimal mutation probability for genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 23–32. Springer, 1990.

[90] Peter JB Hancock. An empirical comparison of selection methods in evolutionary algorithms. In *AISB workshop on evolutionary computing*, pages 80–94. Springer, 1994.

[91] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

[92] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[93] Gilbert Syswerda et al. Uniform crossover in genetic algorithms. In *ICGA*, volume 3, pages 2–9, 1989.

[94] Thomas Bäck and Martin Schütz. Intelligent mutation rate control in canonical genetic algorithms. In Zbigniew W. Raś and Maciek Michalewicz, editors, *Foundations of Intelligent Systems*, pages 158–167, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[95] Lawrence Bull and Terence C. Fogarty. An evolution strategy and genetic algorithm hybrid: An initial implementation and first results. In Terence C. Fogarty, editor, *Evolutionary Computing*, pages 95–102, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[96] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[97] David Beasley, David R Bull, and Ralph R Martin. A sequential niche technique for multimodal function optimization. *Evolutionary computation*, 1(2):101–125, 1993.

[98] David E Goldberg, Jon Richardson, et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[99] Samir W Mahfoud et al. A comparison of parallel and sequential niching methods. In *Conference on genetic algorithms*, volume 136, page 143. Citeseer, 1995.

[100] Samir W Mahfoud. *Niching methods for genetic algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[101] Samir W Mahfoud et al. Crowding and preselection revisited. In *PPSN*, volume 2, pages 27–36. Citeseer, 1992.

[102] Samir W Mahfoud. Simple analytical models of genetic algorithms for multimodal function optimization. In *ICGA*, page 643. Citeseer, 1993.

[103] Ole J Mengshoel and David E Goldberg. Probabilistic crowding: Deterministic crowding with probabilistic replacement. 1999.

[104] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[105] David E Goldberg, Liwei Wang, et al. Adaptive niching via coevolutionary sharing. *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, 97007:21–38, 1997.

[106] Georges Harik. Finding multiple solutions in problems of bounded difficulty. *Illi-Gal report*, 94002, 1994.

[107] Xiaodong Yin and Noël. Germay. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. In Rudolf F. Albrecht, Colin R. Reeves, and Nigel C. Steele, editors, *Artificial Neural Nets and Genetic Algorithms*, pages 450–457, Vienna, 1993. Springer Vienna.

[108] Javier E. Vitela and O. Castanos. A real-coded niching memetic algorithm for continuous multimodal function optimization. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 2170–2177, 2008.

[109] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 61–69, 1989.

[110] Terence Soule, James A Foster, et al. Code size and depth flows in genetic programming. *Genetic programming*, pages 313–320, 1997.

[111] Gilbert Syswerda. Scheduling optimization using genetic algorithms. *Handbook of genetic algorithms*, 1991.

[112] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of an automated parking system. In *Proceedings of the International Conference on Computer, Communication and Control Technologies (CCCT'03) and the 9th. International Conference on Information Systems Analysis and Synthesis (ISAS'03), Florida, USA*, 2003.

[113] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[114] Edward B. Boden and Gilford F. Martino. Testing software using order-based genetic algorithms. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 461–466, Cambridge, MA, USA, 1996. MIT Press.

[115] Lawrence Davis. Handbook of genetic algorithms. 1991.

[116] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software: Practice and Experience*, 30(1):61–79, 2000.

[117] John Hunt. Testing control software using a genetic algorithm. *Engineering Applications of Artificial Intelligence*, 8(6):671–680, 1995.

[118] Jin-Cherng Lin and Pu-Lin Yeh. Automatic test data generation for path testing using gas. *Information Sciences*, 131(1):47–64, 2001.

[119] Takashi Minohara and Yoshihiro Tohma. Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 324–329, 1995.

[120] David J Kasik and Harry G George. Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 244–251, 1996.

[121] Zafer Bingul, Ali Sekmen, S. Palaniappan, and Saleh Zein-Sabatto. Genetic algorithms applied to real time multiobjective optimization problems. In *Proceedings of the IEEE SoutheastCon 2000. 'Preparing for The New Millennium' (Cat. No.00CH37105)*, pages 95–103, 2000.

[122] Mark Last and Shay Eyal. A fuzzy-based lifetime extension of genetic algorithms. *Fuzzy Sets and Systems*, 149(1):131–147, 2005. Fuzzy Sets in Knowledge Discovery.

[123] George Klir and Bo Yuan. *Fuzzy sets and fuzzy logic*, volume 4. Prentice hall New Jersey, 1995.

[124] Acm digital library. `https://dl.acm.org/`, Accessed: 2022.

[125] Google. Google scholar. `https://scholar.google.com/`, Accessed: 2022.

[126] Ieee xplore. `https://ieeexplore.ieee.org/Xplore/home.jsp`, Accessed: 2022.

[127] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. Resource-based test case generation for restful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1426–1434, New York, NY, USA, 2019. Association for Computing Machinery.

[128] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018.

[129] Andrea Arcuri. Restful api automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20, 2017.

[130] Andrea Arcuri. Automated black- and white-box testing of restful apis with evomaster. *IEEE Software*, 38(3):72–78, 2021.

[131] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. Black-box and white-box test case generation for restful apis: Enemies or allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 231–241, 2021.

[132] Man Zhang and Andrea Arcuri. Open problems in fuzzing restful apis: A comparison of tools, 2022.

[133] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, 12(9), 2022.

[134] Jing Liu and Wenjie Chen. Optimized test data generation for restful web service. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 683–688, 2017.

[135] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2018.

[136] Sujit Kumar Chakrabarti and Prashant Kumar. Test-the-rest: An approach to testing restful web-services. In *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 302–308, 2009.

[137] Sujit Kumar Chakrabarti and Reswin Rodriquez. Connectedness testing of restful web-services. ISEC '10, page 143–152, New York, NY, USA, 2010. Association for Computing Machinery.

[138] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential regression testing for rest apis. ISSTA 2020, page 312–323, New York, NY, USA, 2020. Association for Computing Machinery.

[139] Tobias Fertig and Peter Braun. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 1497–1502, New York, NY, USA, 2015. Association for Computing Machinery.

[140] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of restful apis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.

[141] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. A catalogue of inter-parameter dependencies in restful web apis. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 399–414, Cham, 2019. Springer International Publishing.

[142] Clojure test check. `https://github.com/clojure/test.check`, Accessed: 2021-01-07.

[143] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[144] Microsoft. Azure rest api specifications. `https://github.com/Azure/azure-rest-api-specs`, 2019.

[145] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. 2009.

[146] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[147] Andrea Arcuri. Restful api automated test case generation with evomaster. 28(1), jan 2019.

[148] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11, 2017.

[149] Mike Ralphson. Apis.guru. Available: `https://apis.guru/`, Accessed: 2022-02-17 [Online].

[150] List of english words. Available: `https://github.com/dwyl/english-words`, Accessed: March 2022 [Online].

[151] Ascii table. Available: `https://www.asciitable.com/`, Accessed: January 2022 [Online].

[152] İsmail Taşdelen. Sql injection payload list. Available: `https://github.com/payloadbox/sql-injection-payload-list`, Accessed: January 2022 [Online].

[153] Evomaster benchmark. Available: `https://github.com/EMResearch/EMB`, Accessed: April 2022 [Online].

[154] Germany Federal Government. Cwa-verification-server. Available: `https://github.com/corona-warn-app/cwa-verification-server`, Accessed: August 2022 [Online].

[155] Germany Federal Government. Corona-warn-app. Available: https://www.coronawarn.app/en/, Accessed: August 2022 [Online].

[156] Javier Ferrara. Features-service. Available: https://github.com/JavierMF/features-service, Accessed: August 2022 [Online].

[157] Valéria Vargas, Leonardo Wlach, and Daniel Batista. Gestaohospital api. Available: https://github.com/ValchanOficial/GestaoHospital, Accessed: August 2022 [Online].

[158] LanguageTool Org. Languagetool github. Available: https://github.com/languagetool-org/languagetool, Accessed: August 2022 [Online].

[159] Restcountries api github. Available: https://github.com/apilayer/restcountries, Accessed: August 2022 [Online].

[160] Smartbear software. openapi specification editor. https://editor.swagger.io/, Accessed: August 2022.

[161] GitLab. Gitlab apis docs. Available: https://docs.gitlab.com/ee/api/, Accessed: August 2022 [Online].

[162] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Swagger descriptions for gitlab and microsoft apis. Available: https://github.com/GIST-NJU/RestCT/tree/main/exp/swagger, Accessed: August 2022 [Online].

[163] GitLab. Branches api. https://docs.gitlab.com/ee/api/branches.html, Accessed: August 2022.

[164] GitLab. Commit api. https://docs.gitlab.com/ee/api/commits.html, Accessed: August 2022.

[165] GitLab. Groups api. https://docs.gitlab.com/ee/api/groups.html, Accessed: August 2022.

[166] GitLab. Issues api. https://docs.gitlab.com/ee/api/issues.html, Accessed: August 2022.

[167] GitLab. Projects api. https://docs.gitlab.com/ee/api/projects.html, Accessed: August 2022.

[168] GitLab. Repositories api. https://docs.gitlab.com/ee/api/repositories.html, Accessed: August 2022.

[169] Microsoft. Elevations api. https://docs.microsoft.com/en-us/bingmaps/rest-services/elevations/, Accessed: August 2022.

[170] Microsoft. Imagery api. https://docs.microsoft.com/en-us/bingmaps/rest-services/imagery/, Accessed: August 2022.

[171] Microsoft. Locations api. https://docs.microsoft.com/en-us/bingmaps/rest-services/locations/, Accessed: August 2022.

[172] Microsoft. Routes api. https://docs.microsoft.com/en-us/bingmaps/rest-services/routes/, Accessed: August 2022.

[173] Microsoft. Time zone api. https://docs.microsoft.com/en-us/bingmaps/rest-services/timezone/, Accessed: August 2022.

[174] Frederico Cerveira. ucxception fault injection framework. http://estagios.dei.uc.pt/cursos/mei/ano-lectivo-2021-2022/propostas-atribuidas-ano-lectivo-2021-2022/?idestagio=4388, Accessed: August 2022.

[175] Flask framework. https://flask.palletsprojects.com/en/2.2.x/, Accessed: August 2022.

[176] Apiflask. https://apiflask.com/openapi/, Accessed: August 2022.

[177] EclEmma. Jacoco agent. Available: https://www.eclemma.org/jacoco/trunk/doc/agent.html, Accessed: 2022-04-10 [Online].

[178] Carlos Santos. Jacoco-report-generator. Available: https://github.com/TheDemeaN/Jacoco-Report-Generator, Accessed: August 2022 [Online].

[179] GitLab. Gitlab docker image. https://hub.docker.com/r/gitlab/gitlab-ce/tags?page=1&name=13.10.3-ce.0, Accessed: August 2022.

[180] Microsoft. Bing maps dev center. https://www.bingmapsportal.com/, Accessed: August 2022.

[181] Microsoft. Bing maps transactions. https://docs.microsoft.com/en-us/bingmaps/getting-started/bing-maps-dev-center-help/understanding-bing-maps-transactions, Accessed: August 2022.