Academic Year
2023-2024

Enhancing White-Box Search-Based Testing of RESTful APIs Through Seeding with Existing Tests

Mohamed Darkaoui

Master's thesis
Master of Science in computer science: software engineering

Supervisor
Prof. Serge Demeyer, AnSyMo, UAntwerpen
Supervising assistant
Dr. Mutlu Beyazit, AnSyMo, UAntwerpen

University of Antwerp
Faculty of Science

Contents

List of Figures

List of Tables

Chapter 0

Preamble

Abstract

Search-Based Software Testing (SBST) is a promising approach for automating test generation for RESTful APIs. However, the quality of automatically generated tests often falls short compared to manually written tests. Prior research has shown that seeding the search algorithm with existing tests can improve the quality of generated tests in the context of unit testing. In this thesis, we investigate the impact of seeding an evolutionary search algorithm in the context of system-level REST API testing using EvoMaster. We extract tests from existing test suites, transform them into a format compatible with EvoMaster, and use them as seeds for the search algorithm. Our results show that seeding can improve the effectiveness of test generation for RESTful APIs.

Acknowledgments

TOFIX

Chapter 1

Introduction

Recently, there has been a noticeable shift towards cloud computing across various sectors, primarily due to the advantages of service-oriented architectures. In these architectures, tasks are divided into independently deployable services that communicate with each other using the HTTP protocol, which ensures loosely coupled interactions. This leads to benefits such as improved scalability and technology independence between the services. Services worldwide can work together, using the internet, to form a more capable service or an application. The most widely used type of architecture for these services is the REST architectural style.

Software testing, in general, is already challenging by itself. The dynamic and stateless nature of RESTful APIs, coupled with their reliance on network-based communications, introduces additional challenges for testing. Manual testing efforts often fall short in addressing these challenges, achieving comprehensive coverage, and can be complex and time-consuming.

This has led to a growing interest in automated testing methods as a potential solution to these challenges. One promising approach is Search-Based Software Testing (SBST), which employs meta-heuristic optimization methods to generate test cases automatically. While SBST has been widely researched across various contexts, the researchers often report lower quality tests compared to manually written tests [2], particularly in terms of coverage when applied to REST API testing.

Prior research has explored the possibility of combining the knowledge embedded within existing handwritten tests with SBST by seeding search algorithms with the existing tests to enhance the quality of the produced solutions. This approach has shown positive results in the context of unit testing of classes and methods. However, its applicability and effectiveness in system-level REST API testing are still unexplored.

In this thesis, we investigate the impact of seeding an evolutionary search algorithm in the context of system-level REST API testing using the state-of-the-art research tool, EvoMaster. We investigate how seeding influences different coverage metrics, fault detection capability, and the number of generated tests by comparing seeded and unseeded execution processes and results.

This thesis is organized as follows: Chapter 2 discusses relevant background information needed to better understand the rest of the thesis. Chapter 3 discusses the findings and relevant contributions of prior work. The method, where we discuss the research questions and how we tackled them, is discussed in chapter 4. The results, along with the answers to the research questions, are presented in chapter 5. Chapter 6 addresses the threats to validity, and chapter 7 provides the conclusion to this thesis.

Chapter 2

Background

In this chapter, we introduce the fundamental concepts and technologies that form the basis of this research.

2.1 Search-Based Software Testing

Search-based software testing (SBSE) is a subset of search-based software engineering techniques [3] that apply meta-heuristic optimization methods. SBST is mainly used for the automatic generation of test cases and test data, such as values for input parameters of a function. The goal is to generate exhaustive test suites that cover the system under test (SUT) as much as possible. This approach can use many different principles, including evolutionary algorithms, ant colony optimization, and hill climbing. Random search techniques can also be used instead of meta-heuristic ones. These are the simplest forms of SBST and are often used as a baseline to assess other techniques.

SBST is a widely recognized approach for tackling challenges in the software testing domain. Traditional software testing methods fall short in ensuring wide code coverage and fault detection, as the software under test grows in size and complexity. Furthermore, these conventional methods typically require a lot of manual effort. SBST's ability to automatically generate high-quality test cases or test data makes it worth studying as we are looking for ways to supplement or even replace manual testing efforts. According to multiple sources [4, 5, 6, 7], the earliest form of SBST was first introduced in the 1970s by Webb Miller and David Spooner [8]. It was initially an experimental approach, and has now evolved significantly over the decades. Today, it is recognized as an effective enough method for practical use, and it has been adopted in the software industry [7].

In SBST, random techniques, which generate test cases or input data randomly without any strategy, have been proven to be effective in some contexts. However, they generally do not provide a reliable testing strategy across diverse scenarios [9]. This is because random techniques do not have any guidance to explore large and complex problem spaces effectively. On the other hand, meta-heuristic approaches use sophisticated algorithms and fitness functions to guide the search process. Meta-heuristic approaches have been shown to be very promising in various contexts [10], such as unit testing in monolithic architectures [11], in API testing [2], and in Android testing [12].

Meta-heuristic search algorithms are used to find optimal or near-optimal solutions to problems with a large search space. Software test case generation can be reformulated into such a problem [13]: Many possible test cases can be generated for a specific system under test (SUT), we need to systematically select, at a reasonable cost, those that maximize the effectiveness of the generated test suite. The effectiveness is usually measured in terms of code coverage and

fault detection capability [10].

In meta-heuristic SBST, fitness functions play an important role in guiding the algorithm. They quantitatively evaluate how close the generated solution candidate is to the desired outcome. For example, a fitness function might measure the code coverage achieved by a candidate test case or the number of faults it detects. A fitness function measures the quality of a candidate solution based on some criteria and assigns a numeric value to it. The optimization algorithm uses this value to decide which candidates to keep, discard, modify, or combine for further exploration. However, fitness functions can not be applied across contexts. They must be designed to accurately reflect the goals of the scenario at hand [14]. There are three different methods for crafting a fitness function: manual definition, automated generation, and hybrid approaches [15].

A SBST tool iteratively optimizes a test suite or generates content such as test cases or test input data, until it reaches a stopping condition. In the context of test generation with evolutionary algorithms, the process typically starts with an initial population of randomly generated test cases, which serves as a starting point for further exploration. Each test case is evaluated using the fitness function. Then, the tool selects the ones with the highest fitness score from the current population to keep in an archive. In every iteration, the tool selects some test cases from the archive to undergo mutation and crossover with other test cases, generating new candidate test cases. After each iteration, the population of test cases gets more effective. This evolutionary process continues until a stopping condition is met. Typically, reaching a certain time budget or a certain number of generations. After finishing the evolution process, the tool outputs the generated test suites.

2.2 Seeding in SBST

In SBST, seeding refers to using existing information to improve the test generation process and its results in terms of code coverage, fault detection, and other context-specific metrics. This approach makes use of existing SUT data, such as values extracted from source code, runtime values, or previous testing solutions. This information is included in the algorithm's populations to reduce randomness by providing more informed choices and directing the algorithm to more promising areas of the search space. The literature describes various seeding strategies.

2.2.1 Literal Values and Type Seeding

The static constants seeding strategy exploits the constant values found in the SUT's source code or bytecode. The boundary values of a branch can be derived from the branch conditions, and string variables are often compared with strings or matched against string patterns. For example, if a branch condition in the SUT checks if a variable equals a specific string like `"string1"`, this string can be extracted and used inside one or more of the initial test cases. Whenever a new constant needs to be generated for our initial population, the technique samples it with a certain probability from the values found in the source code instead of generating a new random value [16].

The dynamic constants seeding strategy captures values that are observed during runtime, such as results of database queries, user inputs, or computed results and uses them as seeding values. This dynamic data cannot be derived from the source code [17].

Collecting type information from the code, such as types of parameters expected by methods, helps the algorithm select the appropriate object instances. In statically typed languages, such as Java, this can be done using static type analysis, cast operations, and checking with the `'instanceof'` function [17]. For dynamically typed languages, like Python, type information

can be collected at runtime through dynamic profiling, which records the types of variables and method parameters while the code is being executed [18, 19].

2.2.2 Heuristic Optimization of the Initial Population

Instead of starting with a completely random initial population, we can use a strategy that maximizes class method coverage. When selecting a method call for the initial population, a ring buffer can be used. This ring buffer contains a list of method calls, and each time a method is requested, it returns the next method in the buffer. This ensures that all methods are called by some test case in the initial population.

Even when domain knowledge is lacking, it is possible to improve the initial population by creating a much larger pool of randomly generated test cases and selecting the best ones based on their fitness value to form the initial population [16].

Knowledge of the typical test suite size can also help optimize the initial population. Domain experts can estimate the most effective size, and all solutions inside the initial population will have a size close to that. When domain knowledge is lacking, we can infer a balanced test suite size at runtime. The idea is that by selecting the desired size for the generated test suites, with balanced cost and efficiency, we aim to ensure a balanced evolution of the test suites. The search begins with test suites with moderate size and can evolve in both directions, from large and costly solutions with high coverage to smaller, more efficient solutions with lower coverage [20].

2.2.3 Incorporating Previous Solutions

In many cases, a set of existing test cases is available. These tests can originate from earlier executions of the same technique, other testing techniques [21], or written manually [22]. In any case, in order to be able to use this data during the search process, the existing test cases need to be translated into the representation used by the `SBST` tool at hand. A convenient method is first to execute the tests and then reconstruct them from the execution traces [17].

An important decision to make is determining how much the algorithm should exploit existing data. A high rate of exploitation might lead to the algorithm getting stuck in a sub-optimal area of the search space, while a low rate of exploitation might not fully leverage the valuable information available in the existing data.

Another important factor is the quality of the existing set of test cases. If the quality is too low, these test cases will likely have little impact on the search process and may even hinder it. If the quality is optimal, further improvements are unnecessary. Relying too much on low-quality test cases would probably affect the algorithm's performance in a negative way [17, 16].

2.3 Overview of Web Services

A web service is a software system designed to support interoperable machine-to-machine interaction over a network [23]. These services are loosely coupled and can exist independently from one another. They interact with each other and with client applications through a well-defined protocol, typically the Hypertext Transfer Protocol (`HTTP`), either via a private network or, more commonly, via the Internet. This means that services from around the world can work together and with other applications to form a robust service-oriented architecture, even if each is developed in a different environment and a different programming language. Each web service is designed to perform a specific set of tasks, ranging from simple requests for information to complex calculations.

2.3.1 SOAP vs. REST

According to the World Wide Web Consortium, a web service is defined as "A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [24].

Although outdated, this definition holds historical significance. It shows the early importance of strict communication contracts between systems, facilitated by WSDL (Web Services Description Language) and required by SOAP (Simple Object Access Protocol). SOAP is a protocol designed for communication between services and applications. It heavily relies on XML messages as the data format, which can sometimes become very complex. It can work with any transport protocol, not just HTTP. By default, SOAP web services are stateless; however, stateful operations are also supported. To maintain the state, the server stores previous messages with a client, which makes it not scale well [25, 26].

In more recent years, the Representational State Transfer (REST) [27] architectural style has gained a lot of popularity. The shift towards this style is primarily due to its flexibility in handling communications, simplicity, and scalability. Unlike SOAP, REST is not a protocol. It is an architectural style that defines guidelines and constraints for designing APIs that must be satisfied if we want to refer to this APIs as "RESTful". Section 2.4 describes this style in more detail.

2.3.2 What is an API?

An application programming interface (API) in the context of web services is the interface to a reusable software entity that defines the rules and specifications that clients must follow to interact with the service. An interface can be considered a contract between two applications, which defines how to use the requests and responses to interact with one another. These interfaces allow multiple systems to communicate with each other seamlessly [28, 29, 30, 31, 32].

2.4 Representational State Transfer

REST is an architectural style, not a protocol or standard, which means that it does not enforce strict rules on how the API should be implemented. This allows developers to implement a RESTful API in many different ways. In fact, anyone can claim that their API is RESTful. Like any other architecture, REST provides guidelines and constraints that are designed to promote simplicity, scalability, efficiency, and statelessness. These principles were introduced by the famous dissertation of Fielding [27] in the year 2000. They can be summarized as follows:

Client-Server: This principle enforces the separation of the user interface (client) concerns from the data storage (server) concerns. This improves user interface portability across multiple platforms and server scalability by simplifying the components.

Stateless: The communication between client and server must be stateless at all times. Each message from client to server must contain all the necessary information to handle the request. Storing context on the server is not allowed. Not having to store messages between the server and the client promotes scalability.

Cache: The response must indicate whether the data within it can be cashed. If it can, then the client is given the right to reuse that response for future equivalent requests. This constraint

improves efficiency and scalability by reducing the average latency of the interactions between client and server.

Uniform Interface: The way the clients interact with the server must follow a standardized set of rules. This allows different clients to interact with the server in the same way, regardless of their underlying implementations. However, this constraint decreases efficiency since the information is transferred in a standardized way instead of an application-specific way. Four sub-constraints help us achieve interface uniformity: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.

Layered System: Dividing a system into hierarchical layers allows us to constrain the behavior of the components so that each component is unaware of the other layers. This restriction of system knowledge to a single layer places a bound on the system's overall complexity. The trade-off is that these layers add overhead and latency to the processing of data.

Code on Demand (Optional): `REST` enables the downloading and execution of code in the form of applets or scripts to expand the client's capability. This makes things easier for clients by lowering the amount of features that must be pre-implemented. Enabling post-deployment feature downloads enhances system extensibility. But it also reduces visibility. Therefore, it's only an optional constraint in `REST`.

In `REST`, all interactions are based on resources. A resource refers to any piece of data that is important enough to be referred to independently. This not only holds for individual data but extends to collections of items, which are themselves considered as resources. For example, a resource can be a single product in an E-commerce application, a user profile in a social media platform, a lecture video in an educational platform, or a collection of all transactions in a banking service. Each request from the client to the server must identify the resources of interest. This is done through a uniform resource identifier (`URI`).

A resource representation captures the current or the intended state of that resource. It contains resource data and metadata that provide more information about the resource. The representation can be in any format, such as `XML`, `JSON`, or `HTML`. Among these formats, `JSON` is the most widely used [33]. Each resource representation contains enough information to describe how it should be processed. Clients also receive information on additional actions and other related resources. Responses that are returned from a `RESTful API` must tell its clients what the `API` can do, i.e., what resources it has. This concept is known as Hypermedia as the Engine of Application State (`HATEOAS`) [34].

To retrieve, modify, or delete a resource, the client sends its intentions through `HTTP` methods. Each method, `GET`, `POST`, `PUT`, `DELETE`, and other less common ones, specifies which action the client wants to perform on the recourse. For example, `GET` is used to retrieve resources, `PUT` to modify existing resources, and `DELETE` to remove them. Clients also specify how they want to interact with the servers through the `HTTP` headers. In the next section, we discuss `HTTP` in more detail.

An `API` endpoint is a URL that connects a client and server. Clients must submit requests to `API` endpoints to access the features and data of an `API`. A `RESTful API` typically has multiple endpoints that correspond to its available resources or collections of resources [35]. For example, in an e-commerce application, the endpoint `/products` represents all products available, while `/products/{id}` represents a specific product that is uniquely identified by

{id}. Terms between curly brackets are path parameters that can be considered variables. For example, {id} could hold an integer value of 3.

2.5 The Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) was created by the World Wide Web (WWW) architecture and has evolved to support scalability. It is a stateless, application-level protocol designed for distributed, collaborative, hypermedia information systems. It is a request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. This means that HTTP can support a wide range of functionalities, and each HTTP message contains enough information to describe how it should be processed. The uniform interface of the protocol allows client and server interactions to be independent and scalable by hiding the implementation details of services.

An HTTP client is a program that connects to a server and sends one or more HTTP requests. An HTTP server is a program that accepts connections and serves HTTP requests by providing HTTP responses. The same program could act as a client on some connections and as a server on others at the same time. This protocol uses the Uniform Resource Identifier (URI) standard to identify the target resource and the relationship between the resources.

2.5.1 HTTP Message

An HTTP message can either be a request from client to server or a response from server to client. It consists of a start line, headers, and an optional message body. The syntax of the request and response messages differ only on the start line, which is either a request line for requests or a status line for responses [36].

Request line

The request line contains a method token, which is also known as the verb. This indicates the operation that the client wishes to perform on the target resource. The possible request methods are the following [37, 38]:

- **GET:** Requests transfer of the current representation of the target resource. This is a mechanism for retrieving data and has no other effect.
- **HEAD:** Similar to GET, but without the response body, only the status line and the header section are transferred. This is useful for retrieving metadata written in response headers, without transporting the entire content.
- **POST:** Used for submitting an entity to the target resource. This often changes the state of the resource. This method is often used for both the submission of new resources and the updating of existing resources.
- **PUT:** Requests to create or replace the state of the target resource with the state defined by the representation inside the request message.
- **DELETE:** Removes all current target resource representations.
- **CONNECT:** Opens a tunnel from client to server through proxies, as specified by the target resource.
- **OPTIONS:** Requests a list of all available HTTP methods and other options that the server supports for a specific URL.

- **TRACE:** Used to echo back the received request so that the client can see what changes to the request have been made by the server.

In addition to a method token, a request line also contains a request-target and the protocol version, which defines the structure of the remaining message. The request-target is an identifier of the target resource which this request is addressing. Its format can be written in four different ways based on the method used and whether the request is routed through a proxy. The origin-form is used when the request is sent directly to the server that hosts the resource and the method is not `OPTIONS` or `CONNECT`. In this form, the client only sends the absolute path and optional query parameters in the request-target field. The absolute-form is used when the request is made through a proxy. In this case, the request-target must include the full `URI`. For `CONNECT` requests, the authority-form is used, where the request-target only contains the `URI`'s authority components, which are the host and the port number to which the client is trying to connect. Finally, the asterisk-form is only used when the client wants to request server-wide options, using the `OPTIONS` method. Listing 2.1 shows an example of the request line for each of the four forms.

```
# Origin-form: Used for direct requests to server.
GET /flights/446 HTTP/1.1

# Absolute-form: Used for requests through a proxy.
GET http://www.exampleflightservice.com/flights/446 HTTP/1.1

# Authority-form: Used with CONNECT method.
CONNECT www.exampleflightservice.com:443 HTTP/1.1

# Asterisk-form: Used with OPTIONS method.
OPTIONS * HTTP/1.1
```

Listing 2.1: Examples of `HTTP` request lines demonstrating different request-target form usages.

Status Line

The status line is the first line of the response message. It consists of the protocol version, the status code, and an optional phrase describing the status code, called the reason phrase. The status code is a three-digit integer that indicates how well the server understood and fulfilled the client's request. The first digit of this code indicates the response's category; the remaining two have no significance in classification. There are five possible values for the first digit as described by the `RFC` documentation [37, 38]:

- **1xx** (informational): Informs that the request was received.
- **2xx** (successful): Informs that the request was successfully received, processed and accepted.
- **3xx** (redirection): Indicates that the client should take further action in order for the request to be completed.
- **4xx** (client error): Indicates that the request contains an error, for example, in the syntax.
- **5xx** (server error): Indicates that the request appears to be correct, but the server failed to fulfill it.

Header Section

The header section consists of a sequence of header fields, which are colon-separated name-value pairs. Each header field might modify or extend the message semantics, provide (additional) context, or describe the sender. As defined in `RFC 2616` [39], there are four standard categories

for header fields which provide additional information related either to the message in general, to requests and clients, to responses and servers or to the data inside the body of the message.

Message Body

The body is the final part of the message. It is an optional part for both requests and responses and one of the header fields indicates whether it is present or not. For example, requests that intend to only retrieve information, using `GET` or `HEAD` methods usually do not need a body. Similarly, responses where the status code is sufficient to answer the request, for example, `201 created`, usually do not need a body either. When the message body is present, it carries the actual message content, such as the user input in a `POST` request to the server, or the requested resource that is delivered to the client.

Example

Listing 2.2 presents an example of an `HTTP` request. The corresponding response message would follow a similar structure.

```
POST /bookings HTTP/1.1
Host: www.exampleflightservice.com
Content-Type: application/json
Content-Length: 424

{
    "passengerDetails": {
        "firstName": "John",
        "lastName": "Doe",
        "age": 31,
        "passportNumber": "GA4561237"
    },
    "flightDetails": {
        "flightNumber": "754862",
        "date": "18-07-2024",
        "time": "15:30",
        "from": "Brussels",
        "to": "Copenhagen",
        "class": "Economy"
    },
    "contactDetails": {
        "email": "john.doe@example.com",
        "phone": "+32123456789"
    },
    "paymentInfo": {
        "method": "Credit Card",
        "cardNumber": "1111111111111111",
        "expiryDate": "12/2027",
        "cvv": "123"
    },
}
```

Listing 2.2: An example of an `HTTP` request message.

2.6 OpenAPI Specification

The OpenAPI Specification (`OAS`), formerly known as Swagger, is an `API` description format for `REST APIs`. It defines a standard, language independent interface, so that both humans and machines may discover and understand the features of the service without having to access the source code or the documentation. When an `OAS` is well-defined, a user can comprehend and

interact with the remote service without access to server code or knowledge of the server implementation. It not only allows for a more efficient development process but also supports various automated tools, which can generate documentation, client and server code, test suits, and more.

The example provided in Listing 2.3 shows a segment of an OpenAPI Specification (OAS) document formatted in YAML taken from the open source project restcountries[1]. OAS can also be written in JSON format. The content of this example is only a part of a much larger document and it shows some of the key parameters that are commonly used when defining REST APIs.

2.7 EvoMaster

EvoMaster is an open-source SBST tool that is currently hosted on GitHub [40]. It aims to generate system-level test cases for REST APIs, which is our main focus, and it also supports other types of APIs, such as GraphQL [41]. EvoMaster considers RESTful services in isolation, and it does not handle these services as part of a larger orchestration of multiple services [2]. It is a white-box testing tool that can be used for APIs that run on the Java virtual machine (JVM), such as Java or Kotlin, relying on access to the internal details of the system under test (SUT) and thus it requires the availability of the source code. Additionally, this tool offers a naive black-box mode where basic random generation techniques are used [42]. The mode can run on any API, regardless of the programming language, but generally performs much worse than the white-box mode for reasons such as the lack of code analysis and the usage of a naive algorithm. For the rest of this section, we will focus on the white-box approach.

2.7.1 Architecture

EvoMaster consists of two modules that run separately: a core module and a driver module. The core module is responsible for the main tasks, such as running the evolutionary search and generating test files. Meanwhile, the driver module manages the startup, shutdown and resetting the SUT. It also enables white-box testing by measuring code coverage and branch distance metrics, which assist the fitness function within the core component to guide the evolutionary algorithm. Additionally, the driver itself provides a RESTful API and exports the coverage and branch distance information via HTTP messages to the core component. Figure 2.1 shows a high-level architecture design of EvoMaster's components.

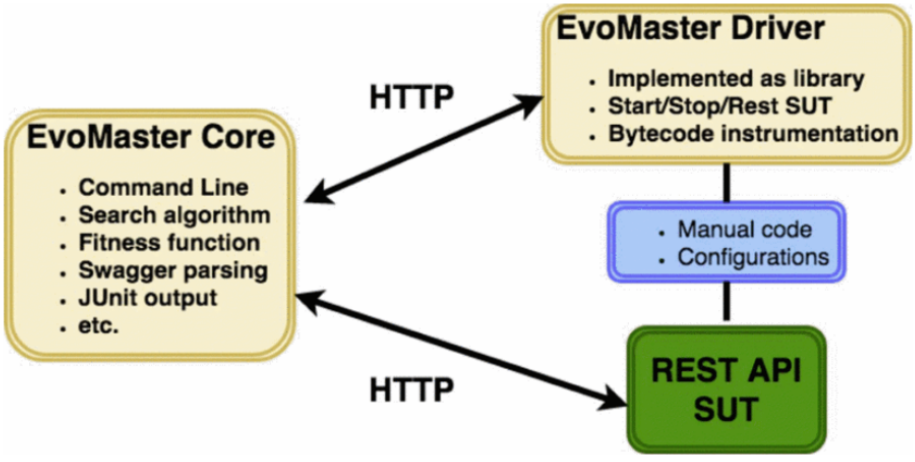


Figure 2.1: High-level architecture of EvoMaster [1].

[1] https://github.com/apilayer/restcountries

```
openapi: 3.0.0
servers:
  - url: 'http://localhost:8080/rest'
info:
  description: 'REST countries API'
  title: REST countries API
  version: v2.0.5
paths:
  /v1/all:
    get:
      operationId: v1All
      responses:
        '200':
          description: Successful response
        default:
          description: Other responses
  /v1/alpha/{alphacode}:
    get:
      operationId: v1Alphacode
      parameters:
        - name: alphacode
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Successful response
        default:
          description: Other responses
  /v1/alpha:
    get:
      operationId: v1Alphacodes
      parameters:
        - name: codes
          in: query
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Successful response
        default:
          description: Other responses
  /v1/currency/{currency}:
    get:
      operationId: v1Currency
      parameters:
        - name: currency
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Successful response
        default:
          description: Other responses
```

Listing 2.3: An example of an OAS document.

2.7.2 Internal Representation of Test Cases

A test case in EvoMaser is considered to be one or more HTTP requests, which follow the structure described in section 2.5.1. It is internally represented as "individual" objects, which are also known as chromosomes in the literature [2]. An individual can hold multiple actions, where each action represents a call made to the SUT, such as REST API calls or SQL operations. These actions are the parts of the test case that can interact with the SUT.

An action consists of multiple genes, which represent the different elements of the action, such as URL parameters, headers, body content, or elements of a SQL query. There are mutable and immutable genes. The mutable genes represent the variable parts of the HTTP requests, such as the query parameters and the content of the body payload. Immutable genes represent the fixed parts, such as the the base URL, port number and necessary HTTP headers. A gene can represent either a query parameter, path parameter, body payload, or an HTTP header.

There are many types of genes, each represents a specific kind of data. To name a few:

- String Genes: Represent textual data, such as URL parameters or text bodies.
- Array Genes: Handle arrays of values, useful for parameters that accept multiple values.
- Date Genes: Composed of three Integer genes representing the year, month, and day, respectively, and are constrained within valid values.

During the evolution process, mutable genes can undergo multiple mutations that are fitting to their data format, depending on their type. For example, string genes can have characters randomly changed, added, or removed, integer genes can be incremented or decremented, boolean genes can be flipped.

EvoMaster uses the OpenAPI Specification (OAS) documents as blueprints for generating the internal representations of test cases. In the OAS, the different endpoints and the parameters are defined. For each endpoint, a chromosome template is created from which individuals will be generated. The actions inside an individual will correspond to the possible interactions with the endpoints. The parameters of these endpoints will be represented as genes within the actions. The genes will later mutate, but respect their constraints as described by the OAS.

Usually, testing a specific endpoint requires previous calls to bring it to a certain state before making the test call. Randomly sampled HTTP calls are unlikely to be valid due to the low probability of generating well-structured HTTP messages that meet all criteria. EvoMaster uses predefined templates to structure the test cases. These templates ensure that the test cases follow a logical sequence of actions that are more likely to be valid. There is one template for each possible HTTP method.

2.7.3 Evolutionary Search Algorithm

EvoMaster supports multiple evolutionary algorithms and uses the many independent objective (MIO) algorithm [43] by default. This algorithm was introduced by Andrea Arcuri in 2017, who is also the author of EvoMaster. This algorithm was specifically designed to improve the scalability and address the complexities of automated test case generation and optimization in SBST.

For each testing target like a line of code or a branch, the MIO algorithm keeps a population of individuals and whether the target is covered by some test or not. Inside a population, the individuals are compared and ranked according to their fitness value for that testing target. EvoMaster considers three types of testing targets: coverage of statements, coverage of

bytecode-level branches, and the coverage of the HTTP status codes. Initially, all populations are empty, and all targets are uncovered. At each iteration, MIO either samples new individuals at random or from one of the populations that are related to an uncovered target. Since the initial populations are all empty, the algorithm only samples at random. For the next iterations, When sampling from a population, the algorithm also mutates the sampled individual. This sampled individual is added to all populations that correspond to uncovered targets and is evaluated and ranked accordingly in each population. Once the size of the population exceeds a certain threshold (10 by default), the individual with the lowest fitness score for that target is discarded. If a target is covered, no more tests are sampled for its population.

The MIO algorithm improves the performance through different techniques. It tracks the number of sampled individuals for each population. Whenever a new individual is added, the counter increments by one, and resets to zero if that individual happens to have the highest fitness score. The algorithm chooses the population with the lowest counter to sample individuals for. This helps the algorithm to focus on targets that are promising and avoids targets that are infeasible to cover. Furthermore, there are also techniques that balance the trade-off between exploration and exploitation, by having a high exploitation rate at the start and becoming more focused on promising targets as time passes. Algorithm 1 shows the pseudo-code of the MIO algorithm.

Algorithm 1: High-Level Pseudo-code of the Many Independent Objective (MIO) Algorithm [2].

input : Stopping criterion C, Fitness function δ, Population size limit n, Probability of random sampling P_r, Start of focused search F
output: Archive of optimized individuals A

```
1  T ← SetOfEmptyPopulations()
2  A ← {}
3  while ¬C do
4      if P_r > rand() then
5          p ← RandomIndividual()
6      else
7          p ← SampleIndividual(T)
8          p ← Mutate(p)
9      foreach k ∈ ReachedTargets(p) do
10         if IsTargetCovered(k) then
11             UpdateArchive(A, p)
12             T ← T \ {T_k}
13         else
14             T_k ← T_k ∪ {p}
15             if |T_k| > n then
16                 RemoveWorstTest(T_k, δ)
17     UpdateParameters(F, P_r, n)
18 return A
```

2.7.4 Fitness Function

The fitness function evaluates candidate test cases (individuals) based on three criteria: statement coverage, bytecode-level branch coverage, and HTTP status code coverage. The MIO algorithm keeps a separate population of individuals for each target. Each individual is assigned a fitness score for its corresponding target, with higher scores meaning that they have better

coverage or higher performance. For each testing target, a numerical score between 0 and 1 is assigned, where 1 means fully covered, and the closer the value is to 0, the further the individual is from covering that target. When two individuals have the same score, the shorter one is preferred. The fitness function uses the code coverage and the branch distance heuristics to estimate how close the individuals are to covering their target. `HTTP` status codes are either covered or not covered (0 or 1), with no values in between. As mentioned in section 2.7.1, the metrics are measured by the driver component.

2.7.5 Manual Setup

To allow EvoMaster to do its work, there are some manual steps we need to take to make sure it can interact with the `SUT` properly. A class that extends the `EmbeddedSutController` from the EvoMaster library needs to be created. This class has two purposes. First, it provides EvoMaster with information about the `SUT`, such as what `TCP` port the server is running on. Additionally, this class defines how the `SUT` should be started and stopped, and where to find the OpenAPI schema.

Before each test execution, the state of the `SUT` environment must be reset. This is typically done by executing `SQL` scripts to reset the database. If the `SUT` requires authentication, developers must provide valid credentials in the `getInfoForAuthentication()` method. This allows EvoMaster to make authenticated requests to the `API`. Listing 2.4 shows a simple example of an implementation of the controller class.

2.7.6 Output

EvoMaster can output test suites in the form of JUnit test cases that use the RestAssured framework.

TOFIX ►◄

```
1
2  public class EMController extends EmbeddedSutController {
3
4      public static void main(String[] args) {
5          EMController controller = new EMController(40100);
6          InstrumentedSutStarter starter = new InstrumentedSutStarter(controller);
7          starter.start();
8      }
9
10     private ConfigurableApplicationContext ctx;
11     private Connection sqlConnection;
12
13     public EMController(int port) {
14         setControllerPort(port);
15     }
16
17     @Override
18     public String startSut() {
19         ctx = SpringApplication.run(WebApplication.class, "--server.port=0");
20         return "http://localhost:" + getSutPort();
21     }
22
23     protected int getSutPort() {
24         return (Integer) ((Map) ctx.getEnvironment()
25                 .getPropertySources().get("server.ports").getSource())
26                 .get("local.server.port");
27     }
28
29     @Override
30     public boolean isSutRunning() {
31         return ctx != null && ctx.isRunning();
32     }
33
34     @Override
35     public void stopSut() {
36         ctx.stop();
37         ctx.close();
38     }
39
40     @Override
41     public void resetStateOfSUT() {
42         ScriptUtils.executeSqlScript(
43             connection,
44             new ClassPathResource ("/empty-db.sql")
45         );
46         ScriptUtils.executeSqlScript(
47             connection,
48             new ClassPathResource ("/data-test.sql")
49         );
50     }
51
52     @Override
53     public ProblemInfo getProblemInfo() {
54         return new RestProblem(
55                 "http://localhost:" + getSutPort() + "/v2/api-docs",
56                 null
57         );
58     }
59 }
```

Listing 2.4: A simple example of the implementation of the `EmbeddedSutController` class [1]

Chapter 3

Related Work

Prior research has explored various aspects of test case generation using search-based techniques for RESTful web services, seeding strategies to improve the test generation in the context of unit testing, and the challenges of finding suitable systems under test for evaluating these techniques.

Golmohammadi et al. [44] conducted a survey of 92 scientific articles on testing RESTful APIs. They found that there has been a dramatic increase in the number of scientific studies on testing REST APIs between 2017 and 2023. Most of the papers papers (66%) in their survey propose a new automated approach for testing RESTful APIs. Both white-box and black-box techniques were investigated, and most of the identified approaches (72%) were black-box. They mention that EvoMaster is the most commonly studied tool (19 papers) in their survey. They also found that OpenAPI specification is widely used as an input format for REST API testing tools.

Fraser and Arcuri [16] provide an overview of seeding strategies in Search-Based Software Testing for Java classes and methods using EvoSuite [11] as the SBST tool. They define seeding as "any technique that exploits previous related knowledge to help solve the testing problem at hand". The paper discusses various seeding techniques, including using source code or bytecode constants, optimizing a search algorithm's initial population, and incorporating previous solutions (see section 2.2). They provide an empirical study where they evaluate the effectiveness of different seeding strategies in the context of generating test suites for Java classes. They found that seeding can significantly improve the ability of SBST tools to generate test cases that achieve higher code coverage.

Rojas et al. [17] investigated the impact of different seeding strategies on search-based unit test generation for object-oriented systems using the same tool. They studied seeding strategies such as using constants from source code or bytecode, values observed at runtime, type information, and incorporating existing tests. The study found that seeding strategies can significantly improve the performance of search-based testing techniques, even for tools that already achieve high coverage. They mention that incorporating previously existing unit tests can help increase code coverage and lead to the generation of more tests, even more so when they are written manually. This is because manually written tests often involve more complex testing scenarios and reflect common object usages and interactions.

Alberto Martin-Lopez et al. [21] compared white-box and black-box test case generation techniques for RESTful APIs by using state-of-the-art tools for each method. They use RESTest [45] for black-box methods and EvoMaster for white-box methods and report the strengths and weaknesses of both. Additionally, they proposed combining both approaches by using the test cases generated by the black-box method as the seed for the white-box method. Their results showed that the hybrid approach outperformed each method in isolation in most cases.

Instead of translating RESTest's test cases directly into a format that EvoMaster can handle, they first translated the seed to an intermediary format. They used Postman [46] for their intermediary format, which is an API platform for building and testing APIs. Postman allows the creation of API tests, which can be exported as a collection of tests in JSON format, referred to as Postman collections. To facilitate this process, they extended RESTest with a Postman test writer and EvoMaster with a Postman test parser.

To address the difficulties of finding suitable SUTs and preparing them, Arcuri et al. [47] created a benchmark of web services called EMB (EvoMaster Benchmark), along with the necessary tools and configurations to setup and run the systems. Initially, EMB was developed for evaluating EvoMaster, but it has been used to evaluate other testing tools as well. The benchmark includes REST, GraphQL, and RPC APIs, with driver classes to programmatically start, stop, and reset the APIs, which makes it easier for researchers to conduct their experiments. The APIs in EMB were taken from open source project and they vary in size and complexity. It also includes artificial case studies (NCS and SCS), which are based on numerical and string functions to validate white-box testing heuristics in system testing.

Chapter 4

Method

4.1 Research Question

Prior research (see chapter 3) has proven the effectiveness of incorporating existing tests as seeds in generating unit tests for Java classes and methods. In this thesis, we will investigate whether these findings also hold in the context of system-level test case generation for RESTful web services by analyzing and comparing the results obtained from EvoMaster. By comparing SBST seeded with pre-existing test cases against those without any seeds, we aim to investigate the impact through how coverage evolves over time during the search using various coverage metrics, the performance of the solutions, and the number of test cases generated. We have formulated our research questions as follows:

- RQ1: How does seeding the initial population influence the evolution of coverage metrics (line coverage, branch coverage, endpoint coverage, and potential fault detection capability) over time in EvoMaster?
- RQ2: Does seeding the initial population lead to the discovery of more unique targets compared to not seeding in EvoMaster?
- RQ3: How does seeding the initial population affect the number of test cases generated by EvoMaster?

4.2 Leveraging Existing Tests

Even though EvoMaster initially only supported SUTs written in languages compiled to JVM, it now supports multiple programming languages, furthermore their REST API tests can be written in any language or framework, because they communicate with the API through network calls. This presents a challenge for translating test cases from all different kinds of frameworks into the internal representation of EvoMaster. Each framework has its own conventions for structuring the tests, test setup/cleanup, etc. The different conventions might also have an effect on how HTTP requests are formatted, and how the response is received and validated. On top of that, different testers have their own way of writing tests, which can differ significantly even when using the same framework.

The lack of a universal pattern makes it infeasible to automatically translate manually written tests into something that represents EvoMaster's initial population of individuals. A possible solution to this problem is to only consider SUTs that use a specific framework, which should be known for its structured way of defining test cases, for example, the RestAssured framework. However, this method will significantly reduce the number of SUTs available for experimentation.

For this research, we chose an alternative and broader approach, where we intercept the HTTP

packets that originate from the handwritten tests and are intended for the SUT. We use network monitoring tools like Wireshark and tcpdump to gather and analyze all the HTTP requests that originate from the tests. This data includes all necessary parts of the HTTP message (methods, URIs, headers, and payloads) and can thus be systematically translated into an EvoMaster-compatible representation of test cases. This method was inspired by the method described in section 2.2.3, where unit tests are reconstructed from execution traces. This seeding strategy can be classified as a strategy that incorporates previous solutions. It also shares similarities with dynamic constant seeding, as it captures values observed during the execution of these existing tests, as described in section 2.2.1.

Instead of translating the captured data directly into EvoMaster's internal representation format, we first convert it into an intermediary JSON format which is equivalent to Postman collections. This approach offers two benefits: (1) By using the Postman format, we can take advantage of the existing Postman parser already implemented in EvoMaster by Alberto Martin-Lopez et al [21] as described in chapter 3. This reduces the amount of work required to conduct this experiment and leverages EvoMaster's existing infrastructure. (2) Using Postman collections allows us to manually verify the intercepted HTTP calls and their responses using the Postman tool. This provides an additional layer of reliability by making sure that the intercepted HTTP data is valid before feeding it to EvoMaster.

Before collecting any HTTP messages, we first need locate and run the REST API test suites with coverage. We do this to collect code coverage information that can be used as a baseline for later comparison with our converted tests. To collect the coverage information, we use IntelliJ's "run with coverage" feature, which provides detailed coverage information and can be exported to HTML reports.

After collecting the HTTP data, EvoMaster processes it to create its own internal representations of the tests. We can then use the custom code that we integrated into EvoMaster to export these processed internal representations of the seed into an executable format. The coverage information of these converted tests can now be compared with the baseline coverage to evaluate the effectiveness of our conversion strategy. We do not expect the coverage to be identical since some information is inevitably lost during conversion from handwritten tests to Postman collections, such as dependencies between HTTP calls or specific database setups before making the call.

4.2.1 Test Extraction

In order to intercept HTTP packets that are transmitted between the test framework and the SUT, we use a network monitoring tool called tcpdump [48]. tcpdump is a powerful command-line network analysis tool that is widely used for monitoring network traffic on Unix and most Unix-like operating systems. It can be used to capture packets that flow through the network. This allows us to see which packets are transmitted and which ones are received. Additionally, tcpdump has filtering capabilities, making it easier to focus only on the packets we are interested in. Filters can be set for IP addresses, port numbers, protocol types, and more.

tcpdump outputs the captured data in a readable format, called pcap (packet capture) format, which is a standard format for network packet analysis. This format shows each packet's content if it is not encrypted. Alternatively, Wireshark [49] can also be used to capture network traffic. It provides a graphical user interface and is compatible with both Unix-like operating systems and Windows. However, we primarily use it for reading the pcap files, since it offers an interactive way of analyzing the packet data. Figure 4.1 illustrates the process of examining the traffic for a port using Wireshark. In Figure 4.1a, an unfiltered capture for port 38888, is dis-

played, showcasing all packets sent to that port during the testing process. Figure 4.1b showcases a filtered capture for the port where only HTTP requests are shown. Lastly, Figure 4.1c provides an example of an HTTP packet, showing the specific elements of the captured request. In order to use tcpdump in a correctly, it is important to first identify which port(s) the tests are using to run the software under test. This information can be deduced from the SUT's documentation, its configuration files or from its source code. Additionally, command-line tools such as `netstat` or `lsof` can be used to figure out which ports are currently being used by which processes.

Before executing the tests that interact with the SUT, we first start tcpdump to capture the traffic. This can be done using the following command:

```
sudo tcpdump -i any port <port number> -w <output path>
```

This command starts tcpdump and makes it capture all traffic on the specified port and writes the data to a pcap file at the specified output path. The traffic can later be filtered and processed. The capturing will continue until manually stopped, after which the pcap file can be opened for analysis.

Now that the interception tool is running, we can execute the REST API tests suites, and their interaction traffic with the SUT will be automatically captured and saved in the specified file path. We created a shell script to process the pcap files and extract the relevant HTTP request information. It filters out the unnecessary data and exports the essential components needed to create Postman collections into a CSV file. This file can later be converted into a Postman collection.The information needed to create Postman collections includes, for each call, the method, raw URL, protocol version, host, path, query parameters, headers, and body payload.

4.2.2 Test Data Transformation

To process the pcap files and extract the relevant HTTP request information, we used tshark [50], the command-line version of Wireshark. tshark allows us to filter and extract specific fields from the captured packets, and it can be used as follows to export information to a CSV file:

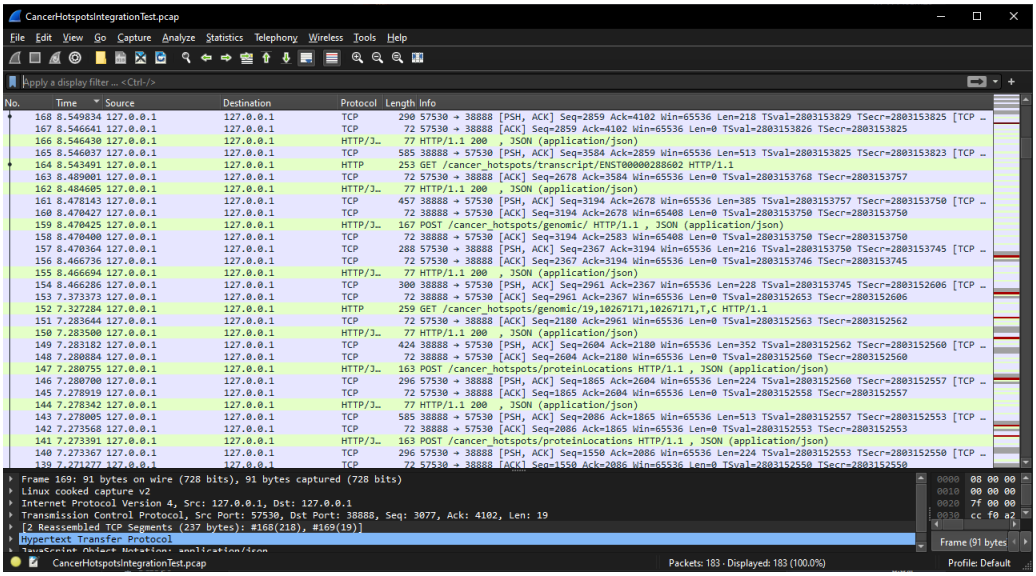
```
tshark -r "<pcap_file>" -Y http.request -T fields \
       -e http.request.method \
       -e http.request.full_uri \
       -e http.file_data \
       -E header=y -E separator='|' -E occurrence=f > "<output_file>"
```

This command processes the specified pcap file, filters for HTTP requests, and extracts the necessary fields. The output is saved to a CSV file.

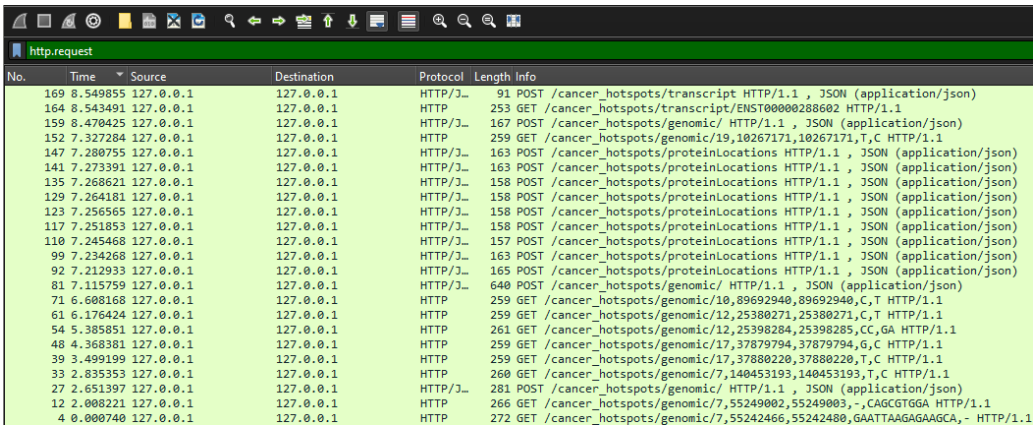
We created a Python script for generating a JSON test file that is compatible with Postman collections. This script reads the CSV file that was previously generated and maps each entry to the correct field in the Postman collection. The JSON data is structured to match the structure of Postman collections, however there are no guidelines for generating Postman collections. Our approach is based on inspecting existing Postman, from which we identify the necessary information that is needed to create these collections.

Once the Postman collection is generated, we import it in the Postman tool to verify that each request in the collection is valid and properly formatted, by executing it against the SUT. Importing and executing a Postman collections into Postman is very user-friendly, it requires only a few clicks to complete the process.

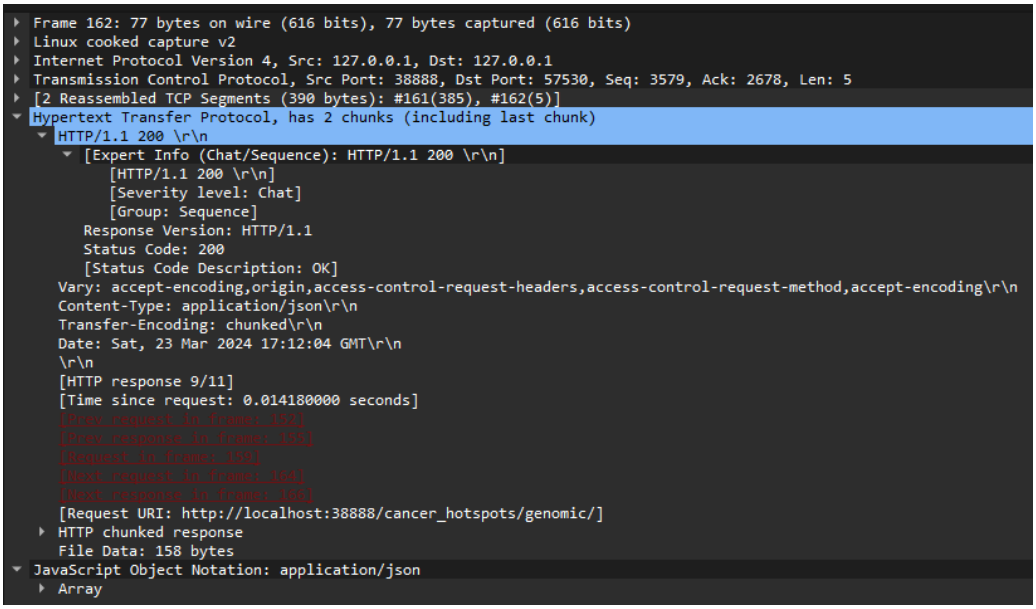
After this verification step, the Postman collection is ready to be used by EvoMaster. The Postman parser within EvoMaster reads the collection and converts it into the internal repre-



(a) Unfiltered capture for port 38888 showing all packets.



(b) Filtered capture for port 38888 showing only HTTP requests.



(c) Example of a single HTTP packet.

Figure 4.1: Examples of pcap file analysis in Wireshark: (a) Unfiltered capture, (b) Filtered capture showing only HTTP requests, (c) Example of a single HTTP packet.

sentation of test cases. These test cases are then used as the initial population for the search-based testing process.

4.3 Test Generation

First, we manually write the driver, which can start, stop, and reset the `SUT` as described in section 2.7.5. Usually, additional setup is needed, such as creating Docker containers and running them separately. When everything is set up, this controller can be executed. If everything goes well, it will listen by default on port `40100` for signals from EvoMaster's core component.

EvoMaster can run from the terminal and can be configured using several options to control its behavior. The `maxTime` option specifies the duration for which EvoMaster should run. It accepts a format like 1h10m12s for hours, minutes, and seconds. The `sutControllerHost` option specifies the hostname of the `SUT` controller, which is useful when it is not running on `localhost`. The `sutControllerPort` option specifies the port on which the `SUT` controller is running if it is different from the default `40100`.

When the core component establishes a connection with the `SUT` controller, the `SUT` is started immediately, and the test generation process begins. During the seeded run, EvoMaster reads the Postman tests one by one and adds them to the archive. However, it only adds tests that cover new targets previously not covered. This means the `MIO` algorithm might discard some of the tests in our Postman collection. Once all the tests have been read, the test generation process starts.

It is possible to adjust the probability of random sampling using the `probOfRandomSampling` option, which specifies the probability of creating new random tests during the evolutionary search process instead of sampling from the archive. By default, this probability is set to 0.5.

To monitor the progress and outcomes of the test generation we use the `writeStatistics` option, which instructs EvoMaster to write statistics about the test generation process, and the `snapshotInterval` option, which defines how often, as a percentage of the total time budget, statistics snapshots should be collected (e.g., every 5% of the time).

However, setting the snapshot interval as a percentage can be problematic for long runs. For example, with a 12-hour run, the first snapshot at 1% would be after 7 minutes. This way, we will probably miss crucial early coverage information. To address this issue, we tweaked the EvoMaster code to save statistics at every search step, ensuring no coverage information over time is missed.

For our seeded test generation approach, we use some additional options. The `seedTestCases` option tells EvoMaster to use existing tests as seeds. The `seedTestCasesPath` option specifies the path to the seed test cases, which are our generated Postman collections. The `seedTestCasesFormat` option defines the format of the seed test cases, where POSTMAN is currently the only valid value for this option.

4.4 Evaluation Metrics and Data Analysis

EvoMaster aims to maximize target coverage (see section 2.7.3), where targets can represent various elements like lines of code, branches, or `HTTP` status codes. Internally, targets are represented by numerical IDs. EvoMaster's driver component analyzes the system, observes the targets, maps each target to a numerical ID, and sends this information to the core component

via HTTP (see section 2.7.1). However, since it assigns these IDs based on the order of discovery during each run, the numerical IDs for the same targets can vary across different executions. To evaluate and compare different setups consistently, we need a stable mapping of IDs across runs.

The core component also receives a descriptive-to-numeric ID mapping from the driver. The descriptive ID is a unique, human-readable identifier that provides information about the target. To achieve consistency across different executions, we reverse this mapping and write the descriptive IDs of all reached targets out to a JSON file. This approach allows us to compare the targets reached with each setup consistently.

An example of the written-out targets with descriptive IDs from a seeded execution is shown in listing 4.1. The JSON file contains two arrays: `TargetsReachedBySeed` and `allTargetsReached`. `TargetsReachedBySeed` contains the targets that were covered by the initial seed, while `allTargetsReached` contains all targets that are covered during the entire execution, including those found due to the seed. In an unseeded execution, the `TargetsReachedBySeed` array would be empty.

```
{
    "TargetsReachedBySeed": [
        "Line_at_org.GlobalExceptionHandler_00021",
        "404:GET:/pdb/header/{pdbId}",
        "HTTP_SUCCESS:GET:/pdb/header/{pdbId}",
        "PotentialFault_PartialOracle_CodeOracle GET:/pdb/header/{pdbId}",
        "Line_at_org.PfamController_00058",
        "Line_at_org.PfamDomainServiceImpl_00029",
    ],
    "allTargetsReached": [
        "Line_at_org.GlobalExceptionHandler_00021",
        "404:GET:/pdb/header/{pdbId}",
        "HTTP_SUCCESS:GET:/pdb/header/{pdbId}",
        "PotentialFault_PartialOracle_CodeOracle GET:/pdb/header/{pdbId}",
        "Line_at_org.PfamController_00058",
        "Line_at_org.PfamDomainServiceImpl_00029",
        "Line_at_org.PfamDomainNotFoundException_00008",
        "Class_org.CanonicalTranscriptAnnotationEnricher",
    ]
}
```

Listing 4.1: Small example of written out targets from a seeded execution

Using this information, we can compare unseeded executions with seeded executions in terms of the number of targets reached. Additionally, we can do these comparisons for each target type; for example, we can compare which method covers more lines or which one finds more potential faults.

To assess the impact of seeding on test generation, we first compare the total number of targets covered in both seeded and unseeded executions. However, a direct comparison might be misleading, as the seeded execution starts with a set of targets that are already covered by the seed. To isolate the effect of seeding on the search process itself, we make a second comparison where we exclude the targets covered by the seed. The results from unseeded executions might also overlap with the targets reached by the seed. Covering these already covered targets again would not add any value to the already existing handwritten tests, so this overlap is excluded as well.

Let S be the set of targets covered by the seed, E_S be the set of targets covered by the seeded

execution, which also includes S, and E_U be the set of targets covered by the unseeded execution. Figure 4.2 provides a visual representation in the form of a Venn diagram for these sets. We will be comparing the sizes of E_S with E_U. We ignore the intersections $E_S \cap S$ and $E_U \cap S$ for a more fair comparison between these sets. Additionally, the intersection $E_S \cap E_U$ can also be ignored, leaving us with a set of targets that is unique to the seeded execution and one that is unique to the unseeded execution, pure red and pure blue in the Venn diagram. In other words, we will be comparing the sizes $|E_S \setminus (S \cup E_U)|$ and $|E_U \setminus E_S|$.

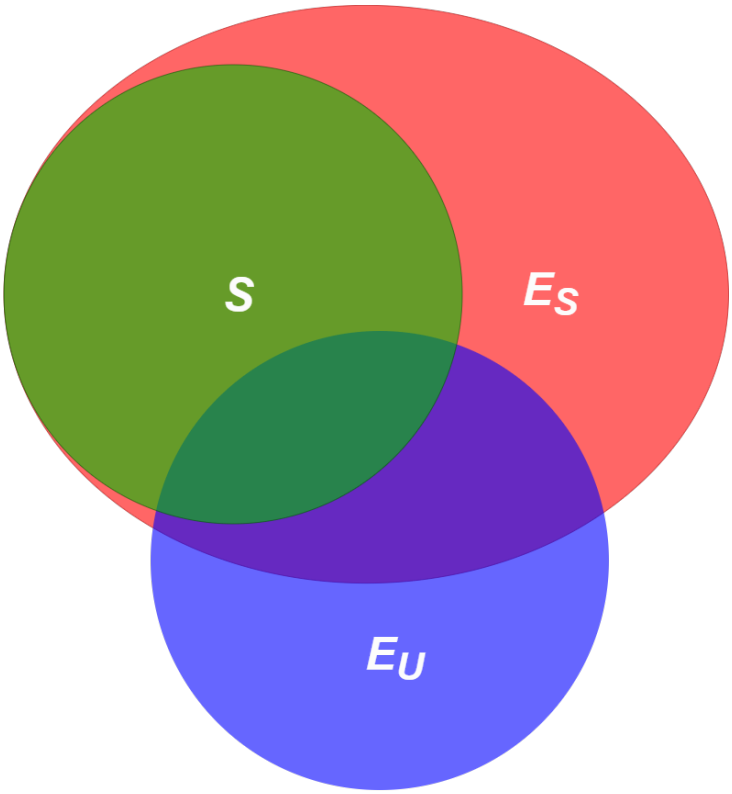


Figure 4.2: Venn Diagram of Targets Reached by Different Testing Strategies. The diagram shows the set of targets S reached by the seed (green), the set of targets E_S reached by the seeded execution (red), and the set of targets E_U reached by the unseeded execution (blue). Overlapping areas represent shared targets between these sets.

Given that each covered target is associated with a descriptive ID, we can filter these targets based on their type (e.g., line coverage, branch coverage, `HTTP` status code). This enables us to perform the analysis for each target type in isolation.

To partially verify our results and reduce possible bias, we use IntelliJ's coverage tool to measure line coverage for the processed seed after being converted to RestAssured code, for the full seeded execution output, and for the full unseeded execution output. This allows for a secondary comparison using a well-established coverage tool.

In addition to comparing the coverage set sizes, we also compare the evolution processes. EvoMaster's ability to take snapshots during the search provides us with data that we can use to construct graphs. This way, we can visually compare both search processes as they evolve, for example, by plotting the coverage achieved over time or the number of potential faults detected (i.e., status code `500`). This allows us to observe whether seeding leads to faster convergence compared to the unseeded approach.

4.5 System Under Test Selection

4.5.1 Selection Criteria

We define a set of selection criteria for the systems under test. Having such criteria definition helps us identify systems that are directly applicable to this research and ensure that the selected systems are suitable for evaluating the effectiveness of our seeding strategy. The following are the criteria used:

- **Open source:** In order to have free and unrestricted access to the source code and to enable validation and replication of the experiments, the system must be open source.
- **REST API:** The SUT should adhere to the principles of REST as described in section 2.4. Systems without full RESTful design may show inconsistencies that hinder EvoMaster's test generation process.
- **Real World Representative:** It should be a realistic and fully functional software product that is designed for realistic purposes. Educational examples or artificial products created for test generation purposes are not considered. This ensures that this research is relevant for real-world applications.
- **Sufficient REST API Tests:** It must have sufficient existing REST API tests, which are necessary for our experiments because they provide valuable initial data for the test generation process. The tests must involve real network communication with the system under test, with real HTTP requests and responses. This ensures that we can intercept the HTTP packets and translate them into EvoMaster's internal representation of test cases. These are used as the initial population for the search algorithm. The quality and coverage of these tests can impact the potential of the seeding strategy. The tests also help to ensure that the functionality of the API has already been tested.
- **OpenAPI Specification** : The system must have a well-defined OAS. An incomplete OpenAPI Specification prevents EvoMaster from generating tests for missing components. Similarly, mismatches between actual resources and those described in the OAS can potentially result in the generation of invalid test cases.
- **Diversity** The selected systems must be diverse in size, the number of endpoints, and the application domain. This diversity ensures that the results are not biased towards a set of properties of the SUT.

Finding systems that fully conform to the criteria is not trivial. Fortunately, there exist benchmarks designed to collect systems with certain characteristics. In our case, The EvoMaster Benchmark (EMB) is suitable since it collects systems that are proven to work well with EvoMaster. Additionally, this benchmark provides drivers for each SUT, which can programmatically start, stop, and reset the state of the SUT. This is extremely helpful since setting up and starting SUTS, which we are unfamiliar with, can be challenging and time-consuming. Currently, EMB consists of 38 SUTS, from which 27 are REST APIs. This allows us to choose from a variety of systems that are already optimized for use with EvoMaster.

Chapter 5

Results

5.1 Details of Selected Systems

For this evaluation, we selected three open-source systems from the benchmark, with varying complexity, size, number of endpoints, and quality of existing test suites. Table 5.1 shows, for each selected SUT, its programming language, the database used, the size in terms of files, the total number of code lines, and the number of endpoints. Table 5.2 provides the amount of HTTP requests intercepted using Wireshark for each SUT.

Table 5.1: SUT details

SUT	Language	Database	Files	LOC	Endpoints
Genome Nexus	Java	MongoDB	405	30004	45
Scout API	Java	H2	93	9736	49
Catwatch	Java	H2	106	9636	14

Table 5.2: Amount of HTTP Requests Intercepted

SUT	HTTP Requests Caught
Genome Nexus	73
Scout API	91
Catwatch	14

5.1.1 Seed Quality

Not all existing tests are created equal, some have high quality while others do not. Additionally, we lose some quality through the translation process as described in Section 4.2. Table 5.3 displays the quality of the seed of each SUT across various metrics, measured by EvoMaster. The metrics include line coverage, branch coverage, and the number of endpoints from which a 2xx response was obtained. These numbers were obtained by having EvoMaster export the coverage metrics immediately after it finishes reading and processing the seed, but before the search starts. It is clear from the table that Genome Nexus is the only SUT with a decent seed quality.

Table 5.3: Seed Quality

SUT	Line Coverage	Branch Coverage	Endpoins Covered
Genome Nexus	43%	25%	22
Scout API	5%	5%	10
Catwatch	7%	8%	2

5.2 Seed Conversion Effectiveness

To assess the effectiveness of our seed conversion strategy (see section 4.2), we compare the line coverage achieved by the original handwritten tests with the line coverage achieved by the processed seed after being converted to RestAssured code. This is done by executing the tests using IntelliJ's "run with coverage" feature. Table 5.4 shows the results of this comparison for each SUT.

For the existing test coverage, we focus exclusively on tests that interact with the REST API through network calls. Before executing the IntelliJ coverage tool on the tests, we remove the test files that do not involve network communication. For the processed seed coverage, we simply run the tests with coverage, but it is important to specify the target packages in the run configuration settings to ensure accurate coverage measurement for the intended packages.

From the comparison it seems that we lose performance in terms of line coverage for each of the conversions. It appears that the loss increases with the amount of database preparation that is needed for the tests. Additionally, a lot of header information gets lost during the conversion process.

From tables 5.4 and 5.3, it seems that there is a discrepancy between the line coverage measured by EvoMaster and IntelliJ IDEA. We also notice that the line coverage measured by EvoMaster is consistently lower across all systems.

Table 5.4: Line Coverage Before and After Seed Conversion

SUT	Before	After
Genome Nexus	60%	56%
Scout API	39%	16%
Catwatch	38%	15%

5.3 Importance of a Complete OAS

It is important to note that some SUTs, like Genome Nexus, do not provide a complete OAS of the entire system due to their usage of public and private APIs. Endpoints meant for the public are included in the OAS, while private endpoints are not. As mentioned in Section 2.7.2, EvoMaster heavily relies on the OAS to discover endpoints and their operations. With a partial specification, EvoMaster will miss endpoints and ignore any seeding related to them. In such cases, manual intervention was necessary to ensure all endpoints were included in the OAS. This intervention had a significant impact on EvoMaster's performance in both seeded and unseeded executions.

Figure 5.1 shows the effect of our manual interventions on Genome Nexus seeded 2-hour executions on the line coverage, obtained using EvoMaster's internal metrics. This can be done for all SUTs with incomplete specifications. We can clearly see that completing the OAS for seeded executions significantly improves the coverage. With a complete OAS, the percentage of covered lines reaches approximately 62%, compared to around 51% with an incomplete OAS. Ad-

ditionally, the seeded execution with a complete OAS continues to grow at a faster rate and starts at a higher coverage compared to the incomplete OAS. This is because EvoMaster discards all HTTP requests that invoke an unknown endpoint. The effect on the unseeded executions is similar. For the rest of this chapter, we will only consider runs with complete specifications.

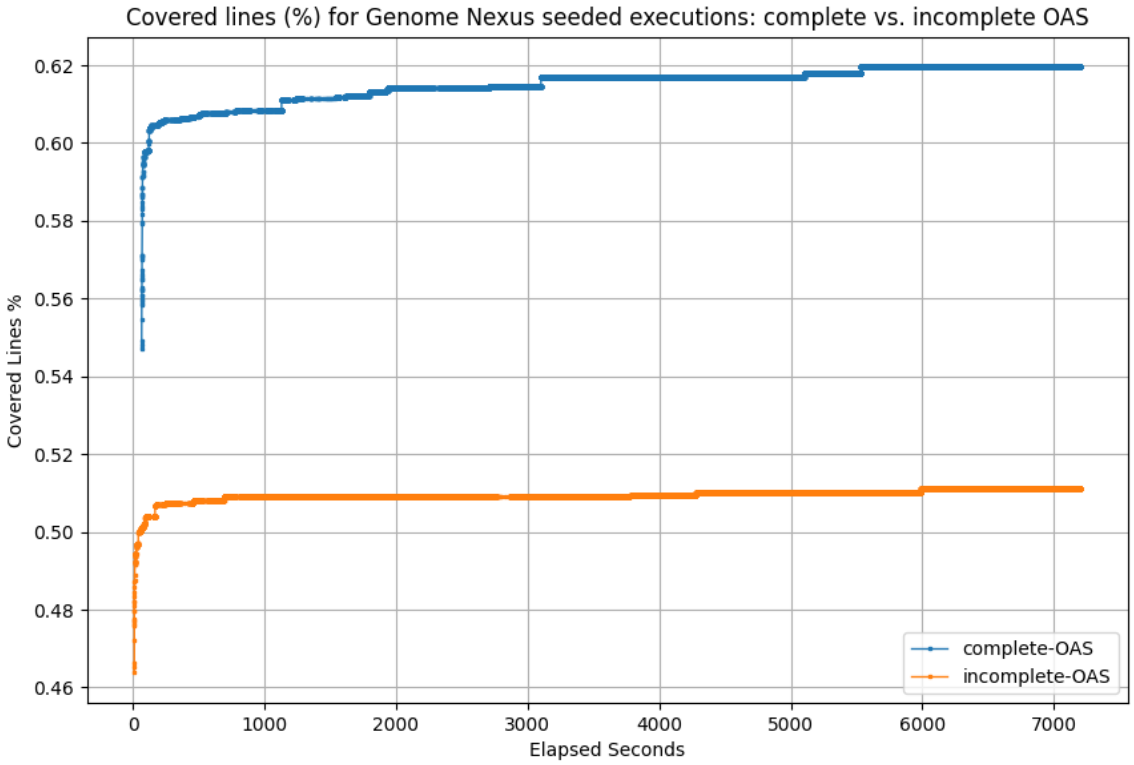


Figure 5.1: Effect of Complete vs. Incomplete OAS on Genome Nexus Seeded Executions

5.4 Coverage Evolution Over Time

Figures 5.2, 5.3 and 5.4 show the average coverage results over time for Genome Nexus, Scout API and Catwatch, respectively, using various metrics. At first glance, we can see that the seeded execution outperforms the unseeded execution across most metrics for all SUTs. Additionally, for all metrics, the seeded execution starts at a higher initial coverage level, but the level depends on the seed quality. The seeded graphs are also slightly shifted to the right. This is due to the time it takes to read and process the seed data.

The covered targets metric is a combination of all the metrics used by EvoMaster, including line coverage, branch coverage, successful endpoint responses, and more. In terms of overall target, line and branch coverage, seeded executions consistently start at a higher coverage level and maintain a lead throughout the testing period across all SUTs. Although the seeded graphs do not appear steeper, it is impressive that they can maintain a consistent distance over time compared to the unseeded graphs, which have lower coverage and thus have a higher potential for improvement, especially in the case of Genome Nexus.

In terms of covered 2xx endpoint responses, seeded executions reach a higher number of successful HTTP responses earlier and maintain this lead throughout the execution period.

Potential faults are identified based on HTTP responses with 500 status codes or discrepancies between the API responses and the expectations defined by the OpenAPI schemas. The detection of potential faults does not show a similar trend as the other graphs. This is expected because existing tests are designed to pass and mostly do not provide insights into how to detect faults. Both seeded and unseeded executions perform similarly in terms of identifying potential faults, with only small advantages for seeded executions.

RQ1: Seeding the initial population in EvoMaster results in higher initial coverage in seeded executions compared to unseeded executions across line coverage, branch coverage, endpoint coverage, and potential fault detection capability. This advantage is consistently maintained throughout the test generation process for all metrics except for potential fault detection.

5.5 Analysis of Target Coverage

When analyzing the sets of targets obtained from the different executions, we immediately notice that the set sizes do not exactly match the results shown in the previous section. For example, an average seeded execution of Genome Nexus covers approximately 7900 targets according to the numbers exported by EvoMaster, as shown in figure 5.2a. However, the data we exported from EvoMaster indicates only 5660 targets. The reason behind this discrepancy is unknown to us.

Figure 5.5 shows Venn diagrams, which visually represent the average amount of targets covered by the seed, the seeded execution, and the unseeded execution for the three SUTs over the three executions discussed in the previous section. From the figures we can see that the average amount of targets covered uniquely by the seeded executions (excluding those covered by the seed itself) is higher than the average amount of targets uniquely covered by the unseeded executions. This indicates that the seeding strategy does have an impact on the search. However, the unique amount of targets covered by the seeded execution is relatively low. A big portion of the targets is covered by both the seeded and unseeded executions.

When we consider the average number of targets covered by the seed and both executions combined, it is clear that Genome Nexus has the highest percentage of targets uniquely covered by the seed, as shown in table 5.5. However, its seeded execution results in the relatively lowest amount of targets uniquely covered by the seeded execution when excluding the targets in the seed itself.

Table 5.5: Total and Seed-Exclusive Target Coverage

SUT	**Total Targets**	**Uniquely by Seed**	**Uniquely by Seeded Execution**
Genome Nexus	5768	27.5%	3.6%
Scout API	2253	1.0%	8.5%
Catwatch	1762	3.8%	11.9%

Looking closer at the unique targets covered by each execution exclusively, as shown in table 5.6, we can clearly see that, on average, the seeded execution covers more unique targets than the unseeded execution. We calculated the gain using the following formula:

$$Gain = \frac{|E_S \setminus (S \cup E_U)| - |E_U \setminus E_S|}{|E_U \setminus E_S|}$$

Table 5.6: Comparison of Unique Targets Covered by Seeded and Unseeded Executions

SUT	**Unseeded Execution**	**Seeded Execution**	**Gain**
Genome Nexus	108	206	90.74%
Scout API	171	191	11.7%
Catwatch	79	210	165.8%

RQ2: Seeding the initial population in EvoMaster leads to the discovery of more unique targets compared to not seeding, with gains ranging from 11.7% to 165.8% on average across the three different systems. However, the number of unique targets covered by each execution is relatively low, so their impact is limited even though the percentage gains are high because most targets are covered by both seeded and unseeded executions.

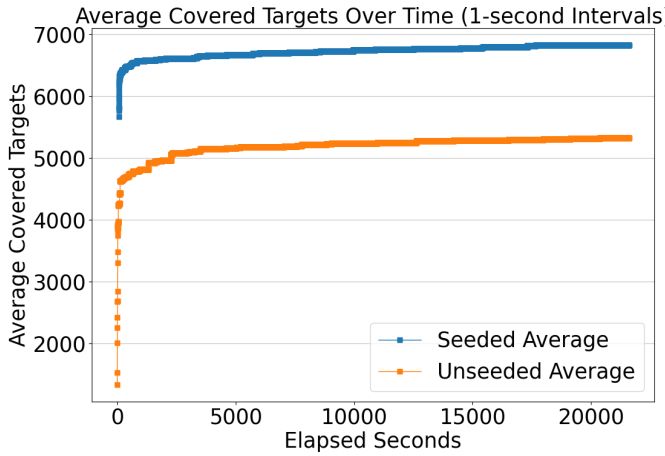
5.6 Impact on the Number of Generated Tests

The seeded executions generally produce a higher number of test cases compared to unseeded executions, as shown in table 5.7. It is important to note that the "Seeded Execution" column includes the initial seed tests. Having more tests is not necessarily better; a smaller test suite with the same coverage is preferred over a large one. On average, Genome Nexus is the only SUT for which the seeded execution generated more tests than the combined total of the initial seed tests and the number of tests produced by the unseeded executions.

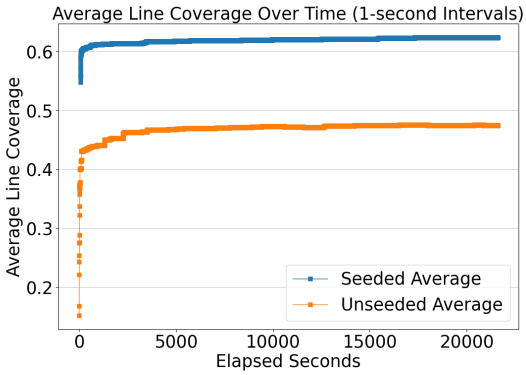
Table 5.7: Average Amount of Tests Generated by Each Execution

SUT	**Seed**	**Seeded Execution**	**Unseeded Tests**
Genome Nexus	33	296	237
Scout API	34	219	208
Catwatch	7	69	66

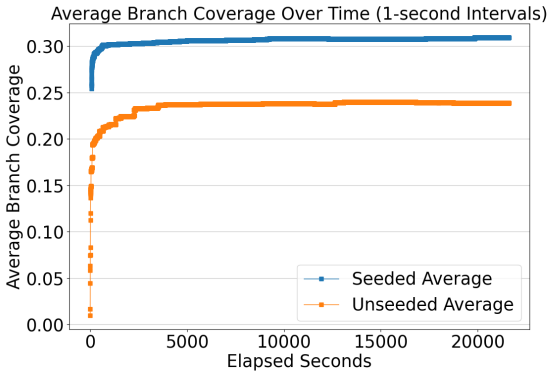
RQ3: Seeding the initial population in EvoMaster leads an increased amount of generated tests.



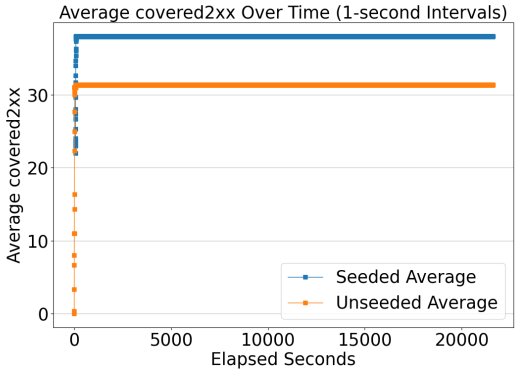
(a) Covered Targets Over Time



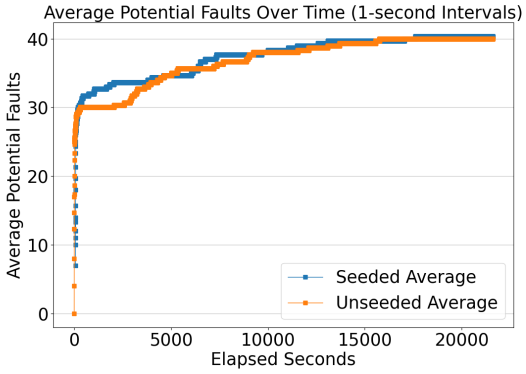
(b) Line Coverage Over Time



(c) Branch Coverage Over Time

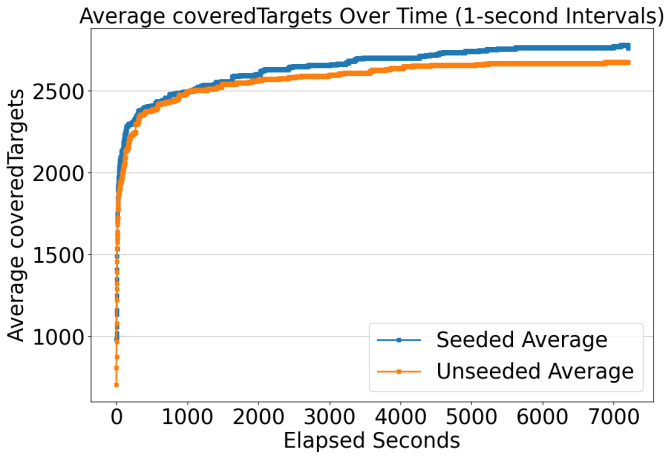


(d) Covered 2xx Endpoint Responses Over Time

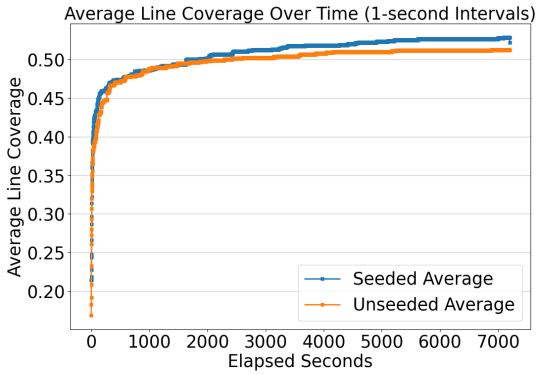


(e) Potential Faults Detected Over Time

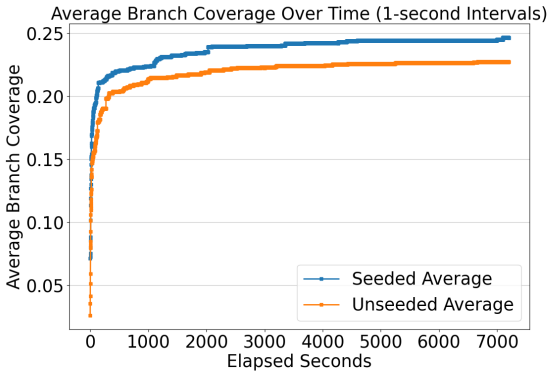
Figure 5.2: Coverage Results for Genome Nexus: Seeded vs. Unseeded Executions. Each graph shows the average results over three runs, with each run lasting 6 hours.



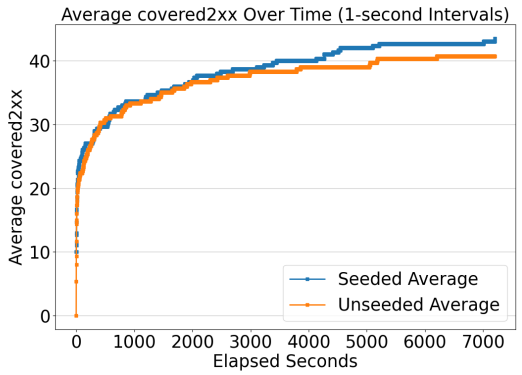
(a) Covered Targets Over Time



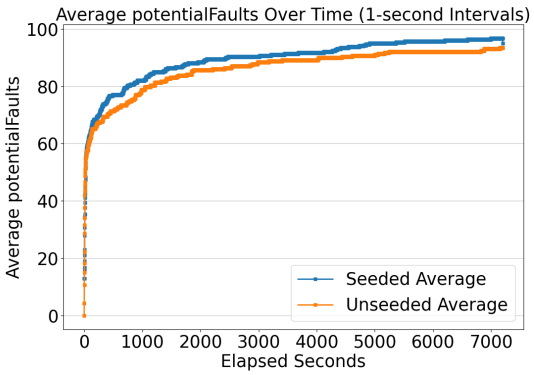
(b) Line Coverage Over Time



(c) Branch Coverage Over Time

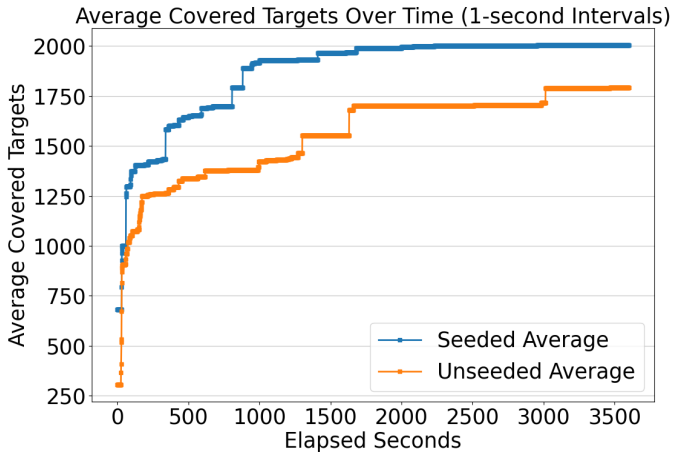


(d) Covered 2xx Endpoint Responses Over Time

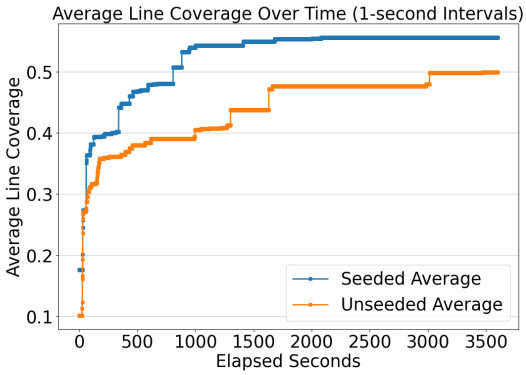


(e) Potential Faults Detected Over Time

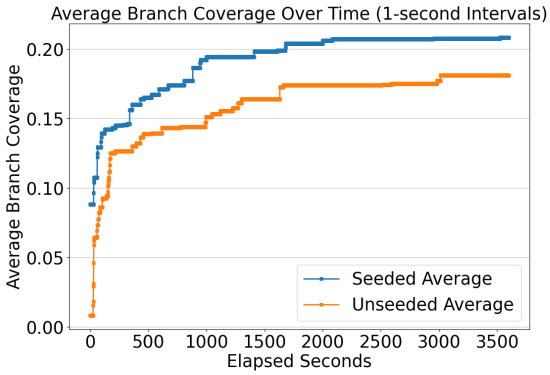
Figure 5.3: Coverage Results for Scout API: Seeded vs. Unseeded Executions. Each graph shows the average results over three runs, with each run lasting 2 hours.



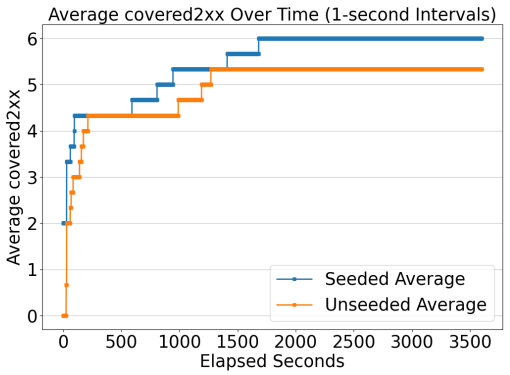
(a) Covered Targets Over Time



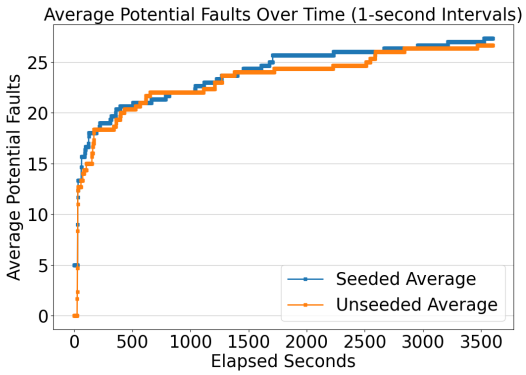
(b) Line Coverage Over Time



(c) Branch Coverage Over Time

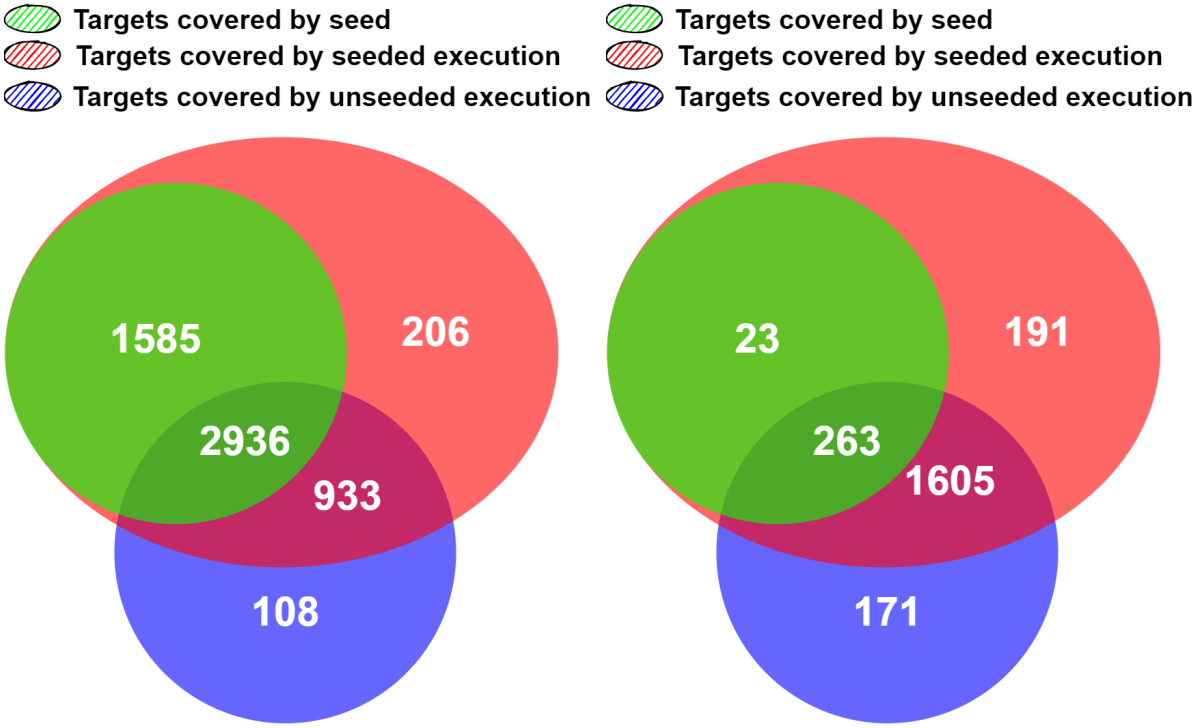


(d) Covered 2xx Endpoint Responses Over Time

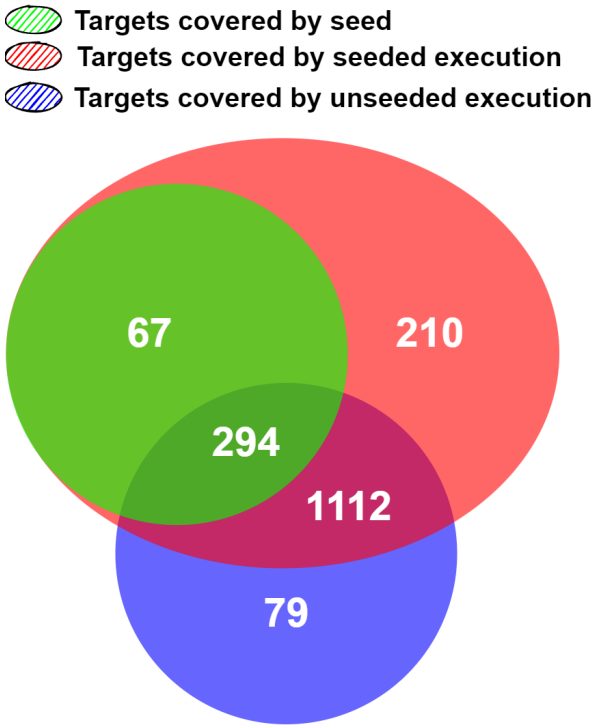


(e) Potential Faults Detected Over Time

Figure 5.4: Coverage Results for Catwatch: Seeded vs. Unseeded Executions. Each graph shows the average results over three runs, with each run lasting 1 hour.



(a) Genome Nexus (b) Scout API



(c) Catwatch

Figure 5.5: Venn Diagrams of Target Coverage Results Across Different SUTs. The Venn diagrams illustrate the average amount of targets covered by the seed, the seeded execution, and the unseeded execution for the three SUTs. The green area represents targets covered by the seed, the red area represents targets covered by the seeded execution, and the blue area represents targets covered by the unseeded execution. The overlaps are the shared coverage between different executions.

Chapter 6

Threats to Validity

In this chapter, we analyze the threats to the validity of this research.

One threat to external validity comes from the fact that we only used three systems under test to evaluate the method and that they all come from the EvoMaster benchmark. The SUTs are all written in Java and only use either MongoDB or H2 as their database. There is a lack of diversity in technologies. This is due to the difficulty of finding Restful web services that fulfill our selection criteria. To mitigate this threat, future research must include more SUTs with a wider range of technologies.

Another threat to external validity is that the default configurations of EvoMaster with minor modifications were used. EvoMaster offers many configuration options that can affect the test generation process, especially for seeding strategies. Future research should experiment with different configurations to determine whether the observed effects are consistent.

For each SUT, the algorithm was executed for 10 hours initially. Based on the time it took for coverage to plateau, this time was then used for all subsequent runs with the maxtime option. There is a small chance that new targets would be discovered after the run has reached the stopping condition. This introduces a threat to internal validity because the algorithm might not have fully explored the search space within the determined time frame.

The usage of the amount of covered targets is a threat to conclusion validity. We use targets because they are used in related research and because we believe they are a fair metric since they represent a combination of coverage metrics with web service-related metrics such as endpoint coverage. However, there is a discrepancy between the number of targets reported in the graphs generated by EvoMaster and the number of targets we extracted for analysis. The reason for this discrepancy is unclear, and it raises concerns about the accuracy of our conclusions based on the extracted data.

Chapter 7

Conclusion

This research focused on the impact of seeding on test generation using EvoMaster. Three open-source systems were selected for this study Genome Nexus, Scout API, and Catwatch. Various metrics were used to assess the effectiveness of seeded versus unseeded executions.

To conduct the experiments, existing tests were translated into EvoMaster-compatible formats. This involved intercepting HTTP packets from the original tests, converting them into Postman collections, and then importing these collections into EvoMaster.

We compared our conversion approach by measuring the line coverage of the tests before and after the conversion. For Genome Nexus, the line coverage decreased from 60% to 56%, for Scout API from 39% to 16%, and for Catwatch from 38% to 15%. Genome Nexus was the only system under test with a decent seed quality.

The coverage evolution over time graphs demonstrated that seeded executions consistently started with a higher initial coverage level and maintained this lead throughout the testing period across all SUTs for most metrics. Although the seeded graphs do not appear steeper, it is impressive that they can maintain a consistent distance over time compared to the unseeded graphs, which have lower coverage and thus have a higher potential for improvement, especially in the case of Genome Nexus.

We compared the average amount of unique targets covered by seeded and unseeded executions. We found that seeded executions covered more unique targets compared to unseeded executions. The number of targets covered by the seed tests ranged from 1% to 27% of all covered targets, while the number of targets covered by the seeded execution, excluding the seed itself, ranged from 3.6% to 11.9%. Additionally, there was a significant relative increase in the number of uniquely covered targets by the seeded execution compared to the unseeded execution, which ranged from 11.7% to 165.8%.

Seeded executions also produce a higher number of test cases compared to unseeded executions. However, the focus should be on coverage and fault detection capabilities, not the quantity of produced tests.

The results indicate that seeding positively impacts the test generation process. It improves both the initial coverage and the number of unique targets covered.

This thesis is the first to investigate the impact of seeding in the context of REST API testing. We provided a methodology for extracting and transforming HTTP request data from existing test suites into a format compatible with EvoMaster. We demonstrated that seeding can improve coverage metrics and increase the number of generated tests. Additionally, we provided

evidence that seeding has little impact on improving potential fault detection capabilities.

The method we used for test extraction and translation is not ideal for existing handwritten tests. However, it seems promising in intercepting user interactions with the `SUT`. Future research could explore how capturing user interactions with the system can help the algorithm achieve higher coverage and detect more faults. Additionally, future work could focus on optimizing configurations, such as determining the ideal probability for the algorithm to sample a test from the archive (influenced by the seed) versus generating new random tests.

Bibliography

[1] 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.

[2] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), jan 2019.

[3] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*, pages 342–357, 2007.

[4] Manju Khari and Prabhat Kumar. An extensive evaluation of search-based software testing: a review. *Soft Comput.*, 23(6):1933–1946, mar 2019.

[5] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[6] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.

[7] Myra B. Cohen. The maturation of search-based software testing: Successes and challenges. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 13–14, 2019.

[8] W. Miller and D.L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, 1976.

[9] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[10] 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.

[11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[12] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[13] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. Reformulating software engineering as a search problem. *IEE Proceedings-software*, 150(3):161–175, 2003.

[14] Filomena Ferrucci, Carmine Gravino, Rocco Oliveto, and Federica Sarro. Genetic programming for effort estimation: An analysis of the impact of different fitness functions. In *2nd International Symposium on Search Based Software Engineering*, pages 89–98, 2010.

[15] Federico Formica, Tony Fan, and Claudio Menghi. Search-based software testing driven by automatically generated and manually defined fitness functions. *ACM Trans. Softw. Eng. Methodol.*, 33(2), dec 2023.

[16] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:121–130, 04 2012.

[17] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016.

[18] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. Small-amp: Test amplification in a dynamically typed language. *Empirical Software Engineering*, 27(6), July 2022.

[19] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. Ampyfier: Test amplification in python, 2021.

[20] Aitor Arrieta, Pablo Valle, Joseba A. Agirre, and Goiuria Sagardui. Some seeds are strong: Seeding strategies for search-based test case selection. *ACM Trans. Softw. Eng. Methodol.*, 32(1), feb 2023.

[21] 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.

[22] Shin Yoo and Mark Harman. Test data regeneration: Generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22:171 – 201, 05 2012.

[23] World Wide Web Consortium. Web Services Architecture. https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis, 2004. Accessed on: 2024-04-25.

[24] World Wide Web Consortium. Web Services Glossary. https://www.w3.org/TR/ws-gloss/#webservice, 2004. Accessed on: 2024-04-25.

[25] Amazon Web Services. The Difference Between SOAP and REST. https://aws.amazon.com/compare/the-difference-between-soap-rest/, 2024. Accessed on: 2024-04-25.

[26] RESTful API. SOAP vs REST APIs. https://restfulapi.net/soap-vs-rest-apis/, 2023. Accessed on: 2024-04-25.

[27] Fielding Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[28] Amazon Web Services. What is a RESTful API? https://aws.amazon.com/what-is/restful-api/, 2024. Accessed on: 2024-04-25.

[29] RESTful API. What is an API? https://restfulapi.net/what-is-an-api/, 2023. Accessed on: 2024-04-25.

[30] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[31] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. *ACM Comput. Surv.*, 54(8), oct 2021.

[32] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[33] 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.

[34] Fikri Aydemir and Fatih Basciftci. Application of hateoas principle in restful api design. In *2022 IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics (CINTI-MACRo)*, pages 000051–000056, 2022.

[35] Postman Blog. What is an API Endpoint? https://blog.postman.com/what-is-an-api-endpoint/, 2023. Accessed on: 2024-05-06.

[36] Fielding, R. and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://datatracker.ietf.org/doc/html/rfc7230, 2014. Accessed on: 2024-05-06.

[37] Fielding, R. and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://datatracker.ietf.org/doc/html/rfc7231#section-4, 2014. Accessed on: 2024-05-06.

[38] Hardie, T. and Livingood, J. HTTP Semantics. https://datatracker.ietf.org/doc/html/rfc9110, 2022. Accessed on: 2024-05-06.

[39] Fielding, R. and Gettys, J. and Mogul, J. and Frystyk, H. and Masinter, L. and Leach, P. and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. https://datatracker.ietf.org/doc/html/rfc2616, 1999. Accessed on: 2024-05-07.

[40] EM Research. EvoMaster: A Tool for Automatically Generating System-Level Test Cases Using Search-Based Software Engineering. https://github.com/EMResearch/EvoMaster. Accessed on: 2024-05-08.

[41] Asma Belhadi, Man Zhang, and Andrea Arcuri. Evolutionary-based automated testing for graphql apis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '22, page 778–781, New York, NY, USA, 2022. Association for Computing Machinery.

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

[43] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In Tim Menzies and Justyna Petke, editors, *Search Based Software Engineering*, pages 3–17, Cham, 2017. Springer International Publishing.

[44] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing restful apis: A survey. *ACM Trans. Softw. Eng. Methodol.*, 33(1), nov 2023.

[45] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 682–685, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Postman. Postman. https://www.postman.com/. Accessed on: 2024-05-15.

[47] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P. Galeotti, Bogdan Marculescu, and Susruthan Seran. Emb: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433–442, 2023.

[48] Tcpdump Developers. Tcpdump. https://www.tcpdump.org/. Accessed on: 2024-05-16.

[49] Wireshark Developers. Wireshark. https://www.wireshark.org/, Year of publication. Accessed on: 2024-05-16.

[50] Wireshark Developers. Tshark. https://www.wireshark.org/docs/man-pages/tshark.html. 2024-05-18.

Appendix A

Relation with Research Projects

Not Applicable.