TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

Automated Testing of OpenAPI Interfaces Using Attribute Grammars

Jueun Park

jueun.park@mailbox.tu-dresden.de
Born on: 16th June 1998 in Seoul, Korea Republic
Course: Media Computer Science
Matriculation number: 4729331
Matriculation year: 2017

Bachelor Thesis

to achieve the academic degree

Bachelor of Science (B.Sc.)

Supervisors
Dipl.-Inf. Johannes Mey
Dr.-Ing. Karsten Wendt
Supervising professor
Prof. Dr. rer. nat habil. Uwe Aßmann

Submitted on: 17th October 2021

Contents

1 Introduction

Automated testing is one type of software testing which is expected to review, validate software products and find errors. Compared to manual testing, it has a huge benefit in execution of test cases, that numerous test cases are automatically generated. In web-based software architectures using REST interfaces, it also could be a meaningful approach to raise the security and quality.

One kind of testing such architectures is to validate REST interfaces and check if there are errors in them. In the last years, the OpenAPI Specification[1] has become the common way to document the communication endpoints and exchanged data structures of REST APIs. Such documentations enable to test REST APIs in black-box testing approaches. OpenAPI specifications are described in JSON or YAML documents and defined by a semi-formal specification describing permitted and required elements as well as their semantics. While there is no formal definition of the full OpenAPI language, a meta-schema exists for the JSON Schema parts of the language.

This paper investigates how such specifications can be used to test interfaces described by them. Since these specifications are tree-shaped, the investigation focuses on a grammar-based analysis approach, Reference Attribute Grammars (RAGs) [10]. This paper also compares already presented REST API automatic testing approaches and examines whether these can also be implemented in RAGs, whether this implementation is sensible and whether RAG has benefits and feasibility to REST API testing. To answer the questions, we have implemtend RAGO API, the first REST API Fuzzing tool implemented in RAG. As a summary, following three questions are formulated as research questions:

RQ1 : Which approaches and techniques for automated tests of OpenAPI specifications are researched and developed so far?
RQ2 : Are suggested testing approaches from the literatures also available in RAG?
RQ3 : Which advantages can RAG provide with its features at expressing testing approaches?

RQ1 is intended to investigate which approaches might be able to implement in RAG. Corresponding literatures are introduced in Chapter 3. RQ2 focuses on the implementation the data structure and testing methods in RAG and is answered in Chapter 4 and Chapter 5. In Chapter 6, Results for RQ3 shows concretely which features of RAG could be usable and which benefits exist.

[1] `https://swagger.io/specification`

2 Background

In this Chapter, we present basis concepts and technologies of this work to understand the following contents.

2.1 Software Testing

Software testing is an examination to see whether a software product works with its expected functionality and is defect free. The problem without testing is that bugs could be expensive or also critical in terms of security. Solving this problem brings the software huge advantages (e.g. effectiveness, stronger security, robust software quality).
There are several classifications in software testing. One classification is related to goals of testing: testing if a software does not have critical bugs (Functional Testing), testing if a software is effective enough (Non-Functional Testing) and modifying an existent software product to correct appeared bugs (Regression Testing) [3]. If a software product is implemented small enough, it is commonly better to test manually and a tester individually defines test cases. But, in most industrial softwares, like REST APIs, it is necessary to automate tests, because the products might be too large to write all cases that must be tested manually.

There is also a type the testings are distinguished by, box approach. It has three categories, black and white Box Testing [16]. Black box tests are developed without knowledge of the internal structure of the system to be tested, but on the basis of development documents. In practice, black box tests are usually not processed by developers of a target software, but by technically oriented testers or by special test departments or test teams. White box tests are developed on the basis of knowledge about the internal structure of the component to be tested.
This paper focuses on automated testing with a black box approach. In Section 2.3, the selection of the approach is explained.

2.2 REST API

REST API is an API (Application Programming Interface) that conforms the common architectural style for web services, REST (REpresentational State Transfer) and is created to standardize the design and development of World Wide Web. The concept of REST has been proposed by Roy Fielding in 2000 [6]. It offers constraints to increase simplicity, performance, visibility, modifiability, portability and reliability of APIs. Constraints defined in REST are following:

Client-Server: API must be a connection in a client-server architecture.

Stateless: Every request must be stateless (i.e. independent of other requests).

Cache: Data in API must be cacheable.

Layered System: Several servers are able to use in one API.

Code on Demand: Responses might be an executable code (optional).

Uniform Interface: API holds on four additional constraints to be consistent

- Resources in requests are identified by URI
- Client should hold the representation of a resource that has enough information to modify or delete the resource.
- Each message includes enough information to describe how to process the message.
- HATEOAS (Hypermedia as the Engine of Application State) is available.

With this set of constraints system resources are characterized by URIs, sent as requests and modified with CRUD operations (Create, Read, Update, Delete), that are mapped to the HTTP methods (POST, GET, PUT, DELETE). Additionally, URIs, header and body objects are modified by parameters (Query, Path, Body, Header, Form). To change URIs and call an operation with variables, parameters in Query and Path are used.
An example for a web service with REST API might be a web service of a pet store. Example operations for this REST API could be:

`GET /pets`
(returns all pet information)

`GET /pets/findByStatus?k1=v1&k2=v2`
(returns information of users with Query parameters k1 = v1 and k2 = v2)

`GET /users/{id}`
(returns information of an user with the given Path parameter, id)

`POST /users`
(creates a new user)

`PUT /users/{id}`
(updates information of an user with the given Path parameter, id)

`DELETE /users/{id}`
(removes an user with the given Path parameter, id)

When a client request is made via a REST API, it transmits a representation of the resource status to the client or endpoint.
This representation or information is provided in mostly JSON or XML. An example of a request in pet store we used for researches is following:

Listing 2.1 Example of Request

```
POST  /v2/pet HTTP/1.1
Host: petstore.swagger.io
Accept: application/json
Content-Type: application/json
Content-Length: 215

{ "id": 0, ... }
```

After a client sends a HTTP request, a corresponding server is going to send back a HTTP response with headers and optionally a payload (data pack for a GET request). Additionally, a numeric status code will be a part of this response as well. Staus codes are specified in one of five categories:

1xx : Informational provisional responses

2xx : Request is successfully processed

3xx : Request requires more information to complete the request

4xx : Client error (invalid request, non-existent resource, client not authenticated or authorized)

5xx : Server error (Server can not supply a valid response)

HTTP status code is an important component to this work, because it provides to test REST APIs and find out errors (e.g. requests expected as valid return 4xx status code or requests expected as invalid return 2xx status code). A response by the request shown above is in Listing 2.2.

Listing 2.2 Example of Response

```
HTTP/1.1 200 OK
Date: Sat, 1 Jan 2000 00:00:00 GMT
Content-Type: application/json
Connection: keep-alive
Server: ...

{ "id": 0, ... }
```

2.3 OpenAPI Specification

Nowadays, there is a way to document REST APIs which allows both humans and computers to understand the whole structure of a RESTful web service. It is called OpenAPI or also known as Swagger (older than 3.0.0 version).
An OpenAPI document with its specification is written in a structured JSON or YAML file and describes which API operations are available, what kind of details they have, how to reach them using a URI, what parameters and request bodies are required and optional in available operations and what authentication schema is.
OpenAPI specification does not hold on strict definitions and has a tree-shaped structure, which facilitates to use and extend in more than 25 programming languages,

while JSON schema is available in them. An example of an OpenAPI document is shown in Listing 2.3.

As an OpenAPI specification describes the interfaces of a server, black-box testings are available. There are many interesting approaches in an black-box architecture: [2], [4], [11], [13], [18], more concrete explanations to separate approaches are in Chapter 3.
Black-box testing needs the specification of the REST API (in our case, OpenAPI specification) and generates test cases automatically, but, does not require an access to the source code. OpenAPI specifications are language-agnostic. So, it is available for every API regardless of in which programming language it is implemented. On the other hand, it might produce test results which are not expected or valid, because the access to the server is limited.

Listing 2.3 Example OpenAPI

```
{
  "openapi" : "3.0.0",
  "info" : {
    "description" : "This is a sample server Petstore server. For this
        sample, you can use the api key ‘special-key‘ to test the
        authorization filters.",
    "license" : {
      "name" : "Apache-2.0",
      "url" : "https://www.apache.org/licenses/LICENSE-2.0.html"
    },
    "title" : "OpenAPI Petstore",
    "version" : "1.0.0"
  },
  ...
  "paths" : {
    "/pet" : {
      "post" : {
        "tags" : [ "pet" ],
        "summary" : "Add a new pet to the store",
        "description" : "",
        "operationId" : "addPet",
        "requestBody" : {
          "$ref" : "#/components/requestBodies/Pet"
        },
        "responses" : {
          "200" : {
          ...
}
```

2.4 Reference Attribute Grammar

One of main research targets of this work is Reference Attribute Grammar[10], an extension of Attribute Grammar (AG)[12].

The concept of AG was introduced by Knuth [12] as a solution of problems with context-free grammars. Context-free grammars (e.g. EBNF, BNF) can only specify syntax with terminal, non-terminal symbols and production rules. But, in computer programs, it is necessary to define the meaning of semantic rules, data types and

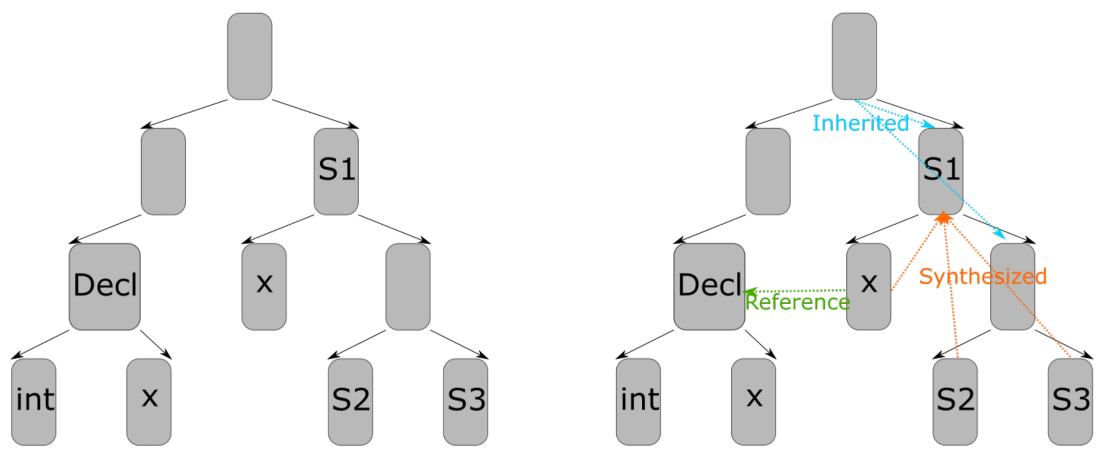

Figure 2.1 Example AST, motivated by [9]

values and AGs allow to define them.
In AGs, terminal and non-terminal symbols are all terminal and non-terminal symbols are characterized as nodes in an Abstract Syntax Tree (AST) and have own attributes to describe which functionality or value they have. The attributes are specified as **synthesized** or **inherited**. In a synthesized attribute, the value is determined by the attribute value at child nodes. Otherwise, the attribute is inherited. AGs are impractical for description of syntax when every dependency of all attributes is local (i.e. definable only with synthesized and inherited) and can follow the syntax tree, but not for descriptions of syntax with non-local dependencies, e.g. a dependency from a root node to a leaf node. This situation leads to problems in terms complexity of analysis and extension.

In 2000, Hedin proposed RAG as an object-oriented extension of AG to solve these problems [10]. This approach enables the reference of nodes in an AST. Every node could be referred and belong to structured attributes e.g. sets, dictionaries, lists, etc. If a node has a reference attribute, the attribute represents a direct connection from an any node that is freely distant (non-local) and to itself. The value of the referred node is directly usable in the referring node without accessing any other nodes in the AST. A graphical example of an AST in RAG compared to a traditional AST without RAG is shown in Figure 2.1.
Such features of RAG represent advantages over AG, largely in efficiency. It is not necessary to duplicate a same value of a node to be utilized in another node and semantic functions in a complex data structure can be split into smaller functions which are completely describable in RAG. Consequently, RAG can extend existing grammars and give them more functionalities.

An extensible system producing language-based compilers and tools is called, JastAdd. It evaluates definitions of structures of AST nodes and attributes and generates modules and tools. There are already several tools implemented and extended with JastAdd (e.g. ExtendJ, JModelica.org, abc, Soot, McLab, Palacom, etc.[1])
Generally, JastAdd needs two input data to generate Java classes. One input data is an ast-file where AST nodes are described. With this input, a class for every terminal node is generated and reconstructed in Java. Other input data is one or more declarative definitions of corresponding attributes. A graphical model of JastAdd is in Figure 2.2. Examples are in Listing 2.4 and Listing 2.5 implementing basic arithmetic operations.

[1] `https://jastadd.cs.lth.se/web/applications.php`

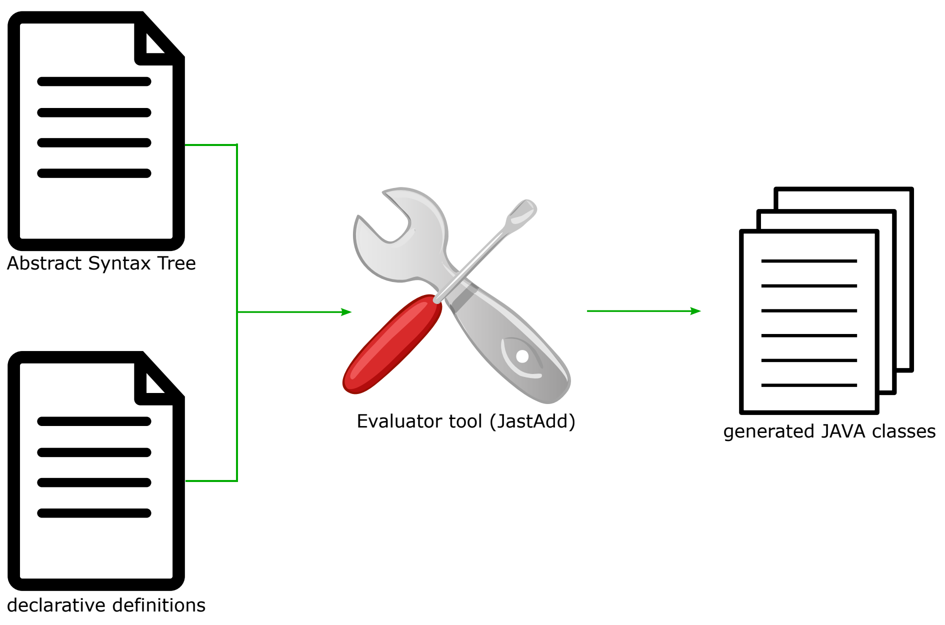

Figure 2.2 Architecture of JastAdd

Listing 2.4 shows how separate expressions can be constructed in AST and Listing 2.5 explains a case of how attributes could be implemented. Especially, synthesized attributes concretized by eq (equation).

Listing 2.4 Example of an AST

```
Root ::= Def* Exp ;
Def ::= <Name:String> <Value:float> ;
abstract Exp ;
abstract UnaryExp:Exp ::= Exp ;
abstract BinExp:Exp ::= A:Exp B:Exp ;
AddExp:BinExp ;
MulExp:BinExp ;
Number:Exp ::= <Value:float> ;
Var:Exp ::= <Name:String> ;
UnaryMinusExp:UnaryExp ;
DivExp:BinExp ;
MinusExp:BinExp ;
```

Listing 2.5 Example of synthesized attributes

```
aspect Printing{
  syn String ASTNode.print();
  ...
  eq AddExp.print() = "(" + getA().print() + " + " + getB().print() + ")";
  eq MulExp.print() = "(" + getA().print() + " * " + getB().print() + ")";
}
```

3 Current Approaches for OpenAPI Testing

This Chapter introduces the current approaches developed for automated testing of REST APIs described in OpenAPI documents. Several approaches suggesting to test APIs with OpenAPI specifications are mostly black-box approaches where an access to a source code is irrelevant [2], [4], [11], [13], [18]. There is also a white box approach [1] which is not the main research target of this paper, but, still an interesting way. Mainly, this Chapter investigates which approaches have been released so far, how they work and what they have achieved.

3.1 Specification-based Approach

In 2018 Ed-douibi proposed a prototype to generate test cases for REST APIs relying on their OpenAPI specifications [4]. This approach receives an OpenAPI specification in JSON file first, configures meta-models (an OpenAPI meta-model and a test suite meta-model) and generates test cases.

After a model extraction (OpenAPI specification into OpenAPI meta-model), it returns a configured OpenAPI meta-model that contains a set of valid models for a language to simplify the integration and modification of information in the specification.
A Model transformation (OpenAPI meta-model into test suite meta-model) considers only properties which are relevant to test cases (HTTP-requests). There are several production rules, for instance, parameter inferences, to define valid and invalid requests and generate them. After the model transformation, test cases are generated and used to test the REST API.
The main research goals of Ed-douibi and his team were not only suggesting a test method, but, also finding out which coverage level (in terms of endpoints, operations, parameters and data definitions of the OpenAPI definition) their tool implementation has and where REST APIs in reality fail mostly in the definitions and implementation.

As results, they could determine that the tool implementation is practically usable in real REST APIs, because 87% of operations, 62% of parameters, 81% of endpoints and 76% of definitions in 91 APIs were testable. They also figured out the main failing points in the definition and implementation: mistakes in the definition (e.g. missing required field, wrong JSON schema) and bad implementation of the APIs (e.g. unhandled exceptions in the server).

3.2 Property-based Approach

QuickREST, the prototype of property-based approach, has been introduced by Karlsson[11]. It suggests a method to generate random test inputs (requests and parameters) applying property-based generation. Test inputs are either completely random or matching to the given OpenAPI specification.

Property-based testing is not only a generation of test inputs, it checks if generated inputs are considered as expected properties with the aid of shrinking. Shrinking means a smallest test input which fails in the same way is searched, if a generated test input is not accepted as expected. With those features, it is possible to formulate and verify properties of the test results e.g. response body, so the testing method produces better results than only reporting HTTP responses.
The first concrete step of QuickREST is generating test inputs including random parameters and request bodies. Test inputs are characterized as URLs.
Secondly, responses to generated requests (test inputs) are checked, if they provide status codes defined in the OpenAPI specification, body payloads also defined in the specification and no 500 status codes. If such conditions are not satisfied, shrinking mentioned above will be executed.

After Karlsson's team has experimented with their proof-of-concept and real APIs (e.g. GitLab) as inputs, they could determine that QuickREST finds real bugs (500 status codes), but, is still limited in industry. For the future work, developments of a model for call sequences considered as interactions of real users and of an automated analysis of logs are needed to improve the effectiveness and size of explorations.

3.3 Constraint-based Approach

The next another automated testing tool is RESTest [13] consisting of a constraint-based black-box approach. Like other approaches, RESTest generates valid and invalid test cases, but, more effectively through the use of automated analysis of dependencies between parameters and test oracles (4xx and 2xx status codes).

This approach is constraint-based and follows also a model-based approach. An OpenAPI specification as input is considered as a system model and optionally describe dependencies between parameters using exclusive libraries (e.g. IDL4OAS, analyzes corresponding dependencies). With this system model and described inter-parameter dependencies, a test model with conformed configuration data is generated. Configuration data of a default test model is manually modifiable. After the generation of a test model, the test model and system model are directly set up for generating an abstract test cases with generation strategies e.g. random input generation. Testing strategies are not only a generation approach, they can also include automated analysis using an IDL extension to find out if an operation defined in the system model (OpenAPI) accords with the analyzed inter-parameter dependencies. Lastly, abstract test cases are rewritten in executable test cases which send requests at the client side.

The research in [13] consists of experiments with 6 commercial APIs. It resulted maximally 99% more and 60% on average valid test cases than random testing and found more than 2000 failures which were not detectable by random testing.

3.4 Operation Dependency Graph Approach

We saw the constraint-based approach tests with parameter dependencies. There is one more way to analyze automatically dependencies of properties in an OpenAPI document. It is able with RESTTESTGEN [18].
What RESTTESTGEN does differently is it analyzes operation dependencies of an API and computes an operation dependency graph.

The graph is a directed graph. If there are two operations derived as nodes and one edge between them, the edge is labeled with a data. This data is an output data of one operation and an input data of the another operation (e.g. an operations getUsers and getUserById could have a data dependency with the data userId). After this analysis is done, RESTTESTGEN is ready to generate valid and also invalid test cases.
Firstly, test cases are automatically generated with the module, Nominal Tester. Inputs of this module are an OpenAPI specification and its analyzed operation dependency graph. With that inputs, the test cases are inferred. Test cases created in this module comform the specification and its constraints.
Subsequently, Error Handling Tester takes generated valid test cases as input and constructs several invalid test cases based on the constructions of nominal test cases. Sending invalid test cases provokes the data validation of the target API and may create unexpected accepted responses.

Experiments with RESTTESTGEN have resulted that this tool is effective in generating test cases, because it was applicable in 87 real-world APIs and had operation coverage of 98%. Developers of RESTTESTGEN still have plans to solve limitations (e.g. authentications are missing, test oracle is only status code, the execution is not iterated), so it has improvements of the capability of testing and presence of security errors.

3.5 Stateful REST API Fuzzer

The last approach related to this paper is RESTler, the first stateful REST API fuzzer [2]. Test cases generated by this tool are also called sequences, i.e. a set of requests, where a request depends on previous requests, if there are several test cases executed.

The focus of the test generation in RESTler is not on whether a request must be valid or invalid in the specification, but on inferring dependencies between requests (e.g. a constraint, executing a request B after a request A, because the input type in B corresponds to the output type in A) and analysis of some hidden contributions (e.g. if a request C is refused after a sequence A after B, C after the sequence A:B will be not executed).

RESTler found 28 bugs in GitLab and several bugs in Microsoft Azure and Office365. In terms of coverage, it is proved that the code coverage increases, before it stops to gain at one time. This tool is not commonly applicable yet, but still valuable to develop more, because it suggests to test APIs with stateful test cases.

3.6 Summary

In this work, we found that most current testing approaches using OpenAPI are based on Fuzzing, sending random, invalid or unexpected data into interfaces and observe responses. HTTP status codes were mostly the basis of test oracles, Specification-based, Constraint-based and Operation Dependency Graph Approaches collect also

Table 3.1 Overview of Approaches in Table

	Type of Errors	Type of Process	Type of Inference	Results
Specification-based Approach	Nominal Test Cases : 4xx/500 Status Codes, Schema Errors Faulty Test Cases : 500, 200 Status Codes	Stateful	Parameter Inference	Bugs found in 37 experimented APIs of 91
Property-based Approach	500 Status Codes	Stateless/Stateful	Inference of Failure Area	In Average, 9.84% Probability of Error Occurrence in GitLab
Constraint-based Approach	5XX Status Codes, OpenAPI Schema Errors, 2XX Related to Parameters, 2XX Related to Parameter Dependencies, 4XX Status Codes	Stateful	Inference of Parameter Dependency by IDLReasoner[a]	Over 2000 Bugs found in 6 commercial APIs
Operation Dependency Graph Approach	Nominal Cases : 5xx Status Codes or Validation Error of Response Error Cases : 2xx or 5xx Status Codes	Stateful	Inference of Operation Dependency	Errors found in 87 selected APIs.
Stateful REST API Fuzzer	500 Status Codes	Stateful	Inference of Request Dependency	28 Bugs in GitLab and several Bugs in Microsoft Azure and Office365 cloud services

[a]https://github.com/isa-group/IDLReasoner

schema validation errors. All of the approaches considered 500 status codes as bugs and several of them expects only 200 status codes by valid test cases and 4xx status codes by error test cases. To result more precisely, they all suggest to inference data of a server or dependency between requests or requests and responses. This usage of inferences requires the stateful process which means requests depend on previous sequences (i.e. requests, responses). The simplified explanantion of current researches is in Table 3.1

4 RAGO API

In this Chapter, we propose RAGO API, the first REST API fuzzing framework modeled in RAG (JastAdd). RAGO API parses the OpenAPI specification in Java to transfer it into a RAG and produces requests that automatically test the target API.

Requests of RAGO API are generated in two basic fuzzing methods. Firstly, generating requests with random values. Secondly, inferring parameters available in responses (i.e. an object returned by a response could be usable as a value in an input parameter).

4.1 Grammar

As mentioned in Section 2.3, OpenAPI specifications are written in structured JSON or YAML and do not hold on strict definitions i.e. programming language-agnostic. It means that specifications can be described and implemented in any programming language or grammar, which also applies to RAG.

To use OpenAPI in RAG, it is firstly necessary to rewrite the OpenAPI structure in an AST. We have constructed this AST in 95 AST nodes to define 30 objects. The version of OpenAPI considered in this framework is 3.0.0. To have a better overview, the definition of Parameter Object in our AST is shown in Listing 4.1 and can be compared with the definition in the OpenAPI official documentation[1]. Every Reference Object has a String token named Ref and refers an object in the OpenAPI document with attributes in Listing 4.2.

Listing 4.1 Parameter Object

```
abstract ParameterOb;
ParameterReference : ParameterOb ::= <Ref>;
ParameterObject : ParameterOb ::= <Name> <In> <Description> <Required:Boolean>
         <DeprecatedBoolean:Boolean> <AllowEmptyValue:Boolean> <Style> <Explode:Boolean>
         <AllowReserved:Boolean> [SchemaOb] <Example:Object> ExampleTuple* ContentTuple*
         Extension*;
```

Listing 4.2 Attributes for Reference

```
coll List<ParameterTuple> OpenAPIObject.parameterTuples() [new ArrayList<>()] root
         OpenAPIObject;
  ParameterTuple contributes this
  to OpenAPIObject.parameterTuples();
```

[1] https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.0.md

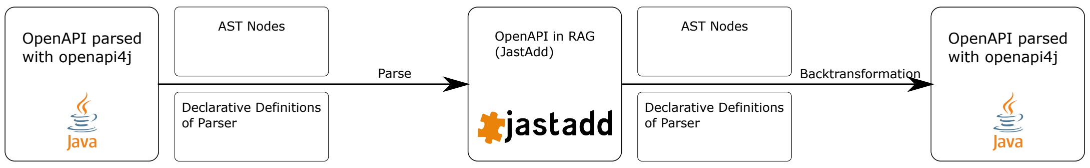
OpenAPI parsed with openapi4j
Java
AST Nodes
Declarative Definitions of Parser
Parse
OpenAPI in RAG (JastAdd)
jastadd
AST Nodes
Declarative Definitions of Parser
Backtransformation
OpenAPI parsed with openapi4j
Java

Figure 4.1 Process of RAGO (Parse, Backtransformation)

```
syn ParameterObject ParameterOb.parameterObject();
eq ParameterObject.parameterObject() = this;
eq ParameterReference.parameterObject() {
  for( ParameterTuple t : root().parameterTuples() ){
    if( t.getKey().equals(getRef().substring(getRef().lastIndexOf("/")+1,
            getRef().length())) )
      return t.getParameterOb().parameterObject();
  }
  return new ParameterObject();
}
```

During transferring the structure from OpenAPI to RAG, several characteristics of JastAdd were detectable:

JastAdd does not support any map structure. So, nodes derived from maps in OpenAPI must be configured in tuples (List of a named tuple containing a key and a value).

For instance, elements of Components Object are defined as maps[2]. JastAdd has only AST classes: ASTNode, List and Opt. This problem is solved with construction of a node saving two child nodes in itself. See an example in Listing 4.3

Listing 4.3 Solution of Map Problem

```
ComponentsObject ::= SchemaTuple* ResponseTuple* ... ;
SchemaTuple ::= <Key> SchemaOb;
ResponseTuple ::= <Key> ResponseOb;
```

Extensions which are properties of an object class in OpenAPI are also defined in nodes.

Extensions with unfixed name and value in Objects of OpenAPI are also solved with such tuples explained in the previous solution.

Listing 4.4 Solution of Extension

```
ResponseObject : ResponseOb ::= <Description> ... Extension*;
Extension ::= <Key> <Value:Object>;
```

4.2 Process, Implementation

After a completion of a syntactic AST structure, values of OpenAPI documents must be stored in AST nodes. A diagram in Figure 4.1 represents a graphic overview of the parser we implemented in a process.

[2] https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.0.md#componentsObject

Before working with attribute definitions, an input specification in JSON or YAML file needs to be parsed in Java to store it into a RAG. A simple JSON or YAML parser does not hold on the schema of an OpenAPI specification. So, we used an additional library, openapi4j[3]. It enables to parse and to validate an OpenAPI document in Java and performs well. It has small limitations e.g. no support for type 'any' and mapping with schema name outside of 'components/schemas' not supported. But, they are not critical to this work.

The next step of processing this framework is transferring parsed objects in Java into AST nodes. This step is done with declarative definitions (not attributes) in jrag file. A small definition example of Parameter Object is in Listing 4.5.

Listing 4.5 Parser for Parameter Object

```
{ ...
  if( parameter.getName() != null )
    parameterObject.setName(parameter.getName());
  if( parameter.getIn() != null )
    parameterObject.setIn(parameter.getIn());
  if( parameter.getDescription() != null )
    parameterObject.setDescription(parameter.getDescription());
  if( parameter.getDeprecated() != null )
    parameterObject.setDeprecatedBoolean(parameter.getDeprecated());
  ...
  return parameterObject;
}
```

First, "parameter" is an object parsed with openapi4j and "parameterObject" is a node which should describe the structure of Parameter Object in RAG. It checks whether values in "parameter" exist and are set into the matching node in the AST (in this case "parameterObject"). At the end of the method, it returns the node.

To see if the parser implemented with JastAdd works correctly, parsed nodes in an AST that describes OpenAPI structure must be transferred back into the structure of openapi4j and validated. The validation of objects after processing is explained in Section 4.3. The way of definitions to transfer RAG into openapi4j is similar to the parser, it only provides the opposite direction of implementation. Listing 4.6 allows to compare itself to Listing 4.5.

Listing 4.6 Back-Transformation for Parameter Object

```
{ ...
  if( !p.getName().isEmpty() )
    parameter.setName(p.getName());
  if( !p.getIn().isEmpty() )
    parameter.setIn(p.getIn());
  if( p.getRequired() != null )
    parameter.setRequired(p.getRequired());
  if( !p.getDescription().isEmpty() )
    parameter.setDescription(p.getDescription());
  if( p.getDeprecatedBoolean() != null )
    parameter.setDeprecated(p.getDeprecatedBoolean());
...
  return parameterObject;
}
```

[3] https://www.openapi4j.org/parser.html

4.3 Validation

For correct (re-)constructions, objects after the back-transformation must be the same before. In that sense, a generated JSON or YAML must be equivalent to the given OpenAPI document. Thankfully, there are several practical libraries to compare two JSONs (JsonNode, JsonDiff, JsonPath) and conditions for the assertion were uncomplicated.

In the validation, differences with empty values and differences in Reference Objects are excluded. After the observation of OpenAPI documents, we have noticed that every author has an individual implementing behavior e.g. description parts are always initialized in some APIs or nodes with empty values do not exist in others. Empty values and Sibling elements of references do not provide semantical differences. So, it was not sensible to generate strictly equivalent values[4]. Its implementation is shown in Listing 4.7.

Listing 4.7 Validation Method

```
JsonNode diff = JsonDiff.asJson(expectedNode, actualNode);
String pathNode;
for( int i = diff.size()-1 ; i >= 0 ; i-- ){
  // get the path of a node involving difference.
  pathNode = "$" + diff.get(i).get("path");
  // check, if this node exists or has an empty value or a reference.
  if( JsonPath.parse(actualNode.toString()).read(pathNode, String.class) == null ||
          JsonPath.parse(actualNode.toString()).read(pathNode, String.class).isEmpty() )
    ((ArrayNode) diff).remove(i);
  else if( !JsonPath.parse(actualNode.toString()).read(pathNode.substring(0,
          pathNode.lastIndexOf(".")).concat(".$ref"), String.class).isEmpty() )
    ((ArrayNode) diff).remove(i);
}
// if the Jsons are equivalent, there is no reason for the text comparison.
// if there is a difference, a text comparison might look better than just the diff.
if (diff.size() != 0) {
  Assertions.assertEquals(actualNode.toPrettyString(), expectedNode.toPrettyString(),
          "JSONs are different:\n" + diff.toPrettyString());
}
```

Firstly, it takes a JSON node which is expected after processing phase and an other node which is actually created. Then, nodes are compared with JsonDiff. It returns an ArrayNode, "diff", describing which differences they have and where the differences have appeared. Unfortunately, it does not have a filter with empty values. So, the validation method tries to get a value of a path in "diff" and checks if it is empty. The differences with empty values or sibling elements of references are removed from this ArrayNode. Afterwards, it executes an assertion with a text comparison, if "diff" has an element. A text comparison gives a bigger overview in Pretty String.

As a result, we have validated the functionality of the structure transfer, while **974 APIs** from the repository of apis.guru[5] are constructed in RAG and reconstructed in openapi4j. APIs involving validation or null pointer errors at openapi4j were excluded. This significant amount of validations presents that this grammar parses and transforms reliably OpenAPI documents which are semantically equivalent to their specifications.

[4]https://swagger.io/docs/specification/using-ref/
[5]https://github.com/APIs-guru/openapi-directory

5 Test Methods

As we have discussed in Chapter 3, most black-box REST API tests principally use Fuzzing, sending unexpected, random data or data providing errors into input interfaces. For this purpose, the newly constructed OpenAPI grammar can be extended with Fuzzing tests. Section 5.1 introduces a basic random test method. In Section 5.2, an approach for the parameter inference we used in this work is presented.

5.1 Random Testing

OpenAPI defines a parameter in an operation in four types, Path, Query, Header and Cookie[1]. In this work, only Path and Query parameters are considered to research the functionality of Fuzzing in RAGO API. These parameters are clearly describable in String values and also comfortably testable, because they target only variable URIs. For the experiments, the OpenAPI document of a Pet Store[2] is mainly used. Additionally, only GET and POST operations are tested to research basic functionality first.

The main code at random testing is following:

Listing 5.1 Random Testing

```
String Uri = getServerUrl();
for (ParameterOb o : operationObject.getParameterObs()) {
  ParameterObject p = o.parameterObject();
  if (p.getIn().equals("path"))
    Uri = p.randomPathParameter(Uri);
  else if (p.getIn().equals("query"))
    Uri = p.randomQueryParameter(Uri);
}
connect(Uri);
```

Listing 5.2 Attribute for Random Parameters

```
syn String ParameterObject.randomPathParameter(String uri); // Generate random Path and
        save in URI

syn String ParameterObject.randomQueryParameter(String uri); // Generate random Query and
        save in URI
```

[1]https://swagger.io/docs/specification/describing-parameters
[2]https://petstore.swagger.io

Initially, the generator for random testing computes a list of parameter objects and iterates all elements in the list (Line 2, Listing 5.1). Subsequently, each iteration examines in which type the parameter is and produces a random URI with synthesized attributes (Listing 5.2). This URI is saved in a String variable (Line 4-7, Listing 5.1). Finally, the test generator sends a request with the generated URI (Line 9, Listing 5.1).

Besides operations with requirements of request bodies, results of this implementation made possible to observe **that parameters were successfully randomized and they produced documented status codes in Pet store[2] (200, 400, 404, 405 status codes).** For the future work, constraints of schema (minItems, maxItems, minLengths, maxLengths, etc.) can be completely extended. In this approach, the generator considers only the existence of enumerations.

5.2 Parameter Inference

Random testing is a one of easiest way to test API and can be useful in some situations. However, it is not effective in REST API testing, because the coverage of the tested API would be particularly low and random values are unusually valid[11]. During the observation in Section 5.1, it was clear to see that random testing mostly produces only requests that receive only 4xx HTTP stauts codes from commercial APIs.

To solve this problem, most of REST API testing approaches use a stateful process, because it enables to analyze elements of APIs and infer inputs which are more appropriate than random inputs. There are several suggestions in Chapter 3. This framework investigates a inference of parameters with algorithms motivated by Specification-based Approach [4] and RESTTESTGEN [18]. Generally, it collects all responses and inferences parameters contributing the same schema of a succesful response. If there is a same schema set in a request and a response, parameters of them are inferred by three strategies:

- **Case insensitive**
- **Id completion** in a field name (e.g. if a property is named with "id", it gets an additional field name available in the specification)
- **Stemming algorithm** (e.g. pet and pets are considered as a same value.)

In the implementation of this work, case insensitive comparison and id completion are utilized to create the basic functionality. Stemming algorithm can be also extended in the future. The follwing code in Listng 5.3 and List 5.4 shows how the parameter inference is compiled with predefined attributes:

Listing 5.3 Parameter Inference

```
generateRequests();
...
for (ResponseTuple t : getResponseTuples()) {
  if (responseCode == 200) {
    SchemaObject respSchema = t.getResponseSchema();
    if (respSchema.getType().equals("array"))
      list = writeDictionaryWithArray(respSchema, response.toString());
    else
      list.add(writeDictionary(respSchema, response.toString()));
  }
}
...
```

```
List<String> paths = new ArrayList<>();
for (ParameterOb o : operationObject.getParameterObs()) {
  ParameterObject p = o.parameterObject();
  if (p.getIn().equals("path"))
    paths = p.addInfPathParameters(pathRef, paths);
  else if (p.getIn().equals("query"))
    paths = p.addInfQueryParameters(pathRef, paths);
}
for (String path : paths)
  connect(path);
```

Listing 5.4 Attribute writeDictionary

```
syn String OperationObject.writeDictionary(SchemaOb schema, String resp) {
  ...
  return inferredParameter;
}

syn List<String> OperationObject.writeDictionaryWithArray(SchemaOb schema, String resp) {
  List<String> list = new ArrayList<>();
  Iterator<JsonNode> props = parseArrayNode(resp).elements();
  ...
  while(props.hasNext())
    list.add(writeDictionary(schema.itemsSchema(), props.next().toString()));
  return list;
}
```

Listing 5.5 Case insensitive comparison

```
syn List<String> ParameterObject.addinfPathParameters(String pathRef,List<String> paths){
  for(InferredParameter i:root().collectInferredParameters()){
    String pathPart=pathRef.substring(pathRef.indexOf("{"),pathRef.indexOf("}")+1);
    if(getName().equalsIgnoreCase(i.name()))
      paths.add(pathRef.replace(pathPart,i.value()));
  }
  return paths;
}
```

Before it starts with the parameter inference, random tests of Section 5.1 are generated first (Line 1, Listing 5.3). During this execution, the status code of a response is checked if it is a successful response with 200 status code (Line 4, Listing 5.3). Afterwards, the response schema of returned values is also checked. If it is in type array, the function "writeDictionary" is iterated, otherwise it only executed once (Line 6-9, Listing 5.3). The attribute "writeDictionary" saves the returend values of a successful response in seperate properties and write them in a dictionary (e.g. properties "id" and "name" are seperately stored with their value in the dictionary). If the schema of a response provides a reference of a schema object, the field name gets a name of a reference as prefix (Listing 5.4). Subsequently, this implementation does the similar way of execution in random testing at the generation phase. Firstly, it iterates all parameter objects (Line 14, Listing 5.3) and examines whether the parameter type is Path or Query (Line 16-19, Listing 5.3). Both attributes return URIs with parameter values inferred by the dictionary and case insensitive comparison (Line 4, Listing 5.5). Generated URIs are put in a list. Lastly, the generator attribute sends requests with the URIs and starts with observation (Line 20-21, Listing 5.3).

As results, the test case generator with parameter inference implemented in this framework could **create maximum over 300 acceptable URIs for the parameter petId in the selected API, pet store[2], at the operation getPetById.** It generated also numerous requests denied by the server. After the observation of several execution iterations, it can be assumed that the API with this operation sends 200 or 404 status codes randomly or according to some rules, because a same URI provided status codes in that way.

6 Suitability of RAGs for OpenAPI Testing

To evaluate how constructive and suitable RAGs are for OpenAPI testing, we review research questions in following sections. In Section 6.1, results for **RQ1** and **RQ2** are summarized. In Section 6.2, RAG's features that were helpful at implementing and features that could be extended more in RAGO API are discussed.

6.1 Feasibility

RQ1 : **Which approaches and techniques for automated tests of OpenAPI specifications are researched and developed so far?**

The results of current existing approaches are shown in Table 3.1. As we can see, most of current approaches to test OpenAPI are in black-box and use Fuzzing method to generate test cases. There are differences between approaches at test generation phase.

Several of them inference parameters or operations or use test model generation. They also have differences at bug types. They all consider 500 status codes as bugs, but, three of them consider 200 status codes in test cases expected errors and 4xx status codes in test cases expected successful responses as bugs. Exact bug reports of approaches which experimented with industrial APIs excluded Property-based Approach formulate that they can be interesting research targets combined with RAG.

RQ2 : **Are suggested testing approaches from the literatures also available in RAG?**

To determine an answer of this question, we have constructed a data structure for OpenAPI specification in Chapter 4 to parse OpenAPI documents and **validated the parser with 974 commercial APIs**, i.e. the parser taken an OpenAPI document returns the same elements except elements with empty values and sibling elements of references in an OpenAPI document. The input and output documents still semantically are same. In Chapter 5, we have also implemented two Fuzzing approaches (Random Testing, Parameter Inference). Parameter Inference is motivated by Specification-based Approach[4] and RESTTESTGEN[18], where seperate elements of responses are collected and used as inputs in a parameter with the same schema of a response.

Several approaches are based on their own metamodels. It is not clear yet, whether such model-based approaches could be developed in RAGs. For implementing metamodel generations of approaches in RAGs, JastAdd frameworks [8], [15] could be helpful to transform a model to another model. It is worthful to discuss and research how such frameworks developed already could be applied to those model-based approaches, because it would save an enormous amount of effort at implementing.

6.2 Benefits

RQ3 : Which advantages can RAG provide with its features at expressing testing approaches?

During the implementation phase of parameter inference, RAG was beneficial at writing codes. At writing codes for parser and back-transformation, attributes could not be applied, because the functional parts change the structure of AST, which conflicts to the definition of attribute and also in JastAdd. So, it could not be realized that attributes can be practical in this time. But, following attributes and features of JastAdd were helpful to construct the generators and implement them:

- As long as a returning value can be called or storable by an AST node, the syntax in RAG is reduced with an equals sign and simpler than in common programming language. Therefore, the source codes are compact. To compare how RAG could be implemented effectively, see the implementation of an attribute in this work in Listing 6.1 and how it is defined in common Java description in Listing 6.2.
- To iterate all elements available in a node, an inherited attribute facilitated the writing effort. Paths Objects are child nodes of an OpenAPI Object, so, all that should have been done was only two lines of code instead of writing an extra for-loop. See Listing 6.3.
- Nodes described as References could refer objects with a collection attribute. Listing 4.1 shows how attributes for such purpose look like.

Listing 6.1 Attribute InferredParameter.value()

```
syn String InferredParameter.value() = getParameter().substring(
        getParameter().indexOf("?") + 1 );
```

Listing 6.2 InferredParameter.value() in Java

```
Class InferredParameter {
...
constructor;
...
  public String value() {
    return getParameter().substring( getParameter().indexOf("?") + 1 );
  }
}
```

Listing 6.3 Inherited Attribute in Paths Object

```
inh boolean PathsObject.inferUrl();
eq OpenAPIObject.getPathsObject(int i).inferUrl(){
  ...
}
```

Additionally, JastAdd can easily extend the grammars. There are several approaches to extend grammars with JastAdd, e.g. [5]. Therefore, JastAdd could also extend the grammar proposed in this paper in that way. Besides it, the grammar could be simply analyzed by JastAdd. Analyses suggested in RAG [15], [7], e.g. Name Analysis, Type Analysis, might also be usable at parameter inference, which the name inference in this paper uses the similar pattern as Name Analysis (Lookup). Relational Reference Attribute Grammars [14] could improve references and enable to build a new test model and connect it to the OpenAPI model implemented in this paper, parallel to [8], [15]. A recently invented framework, RagConnect, enables to connect RAG-based models to models with other basis [17]. Since version 0.2.1, it has a new model connection with REST and might help at generating test cases, because test cases are based on requests and responses of REST APIs. All of these extensions are open topics. So, it might be worthful to discuss and research how they can be used into OpenAPI Testing to suggest tests with a better quality.

7 Conclusion

In this paper, we have researched which approaches to test REST APIs documented by OpenAPI are developed so far. Most of approaches are based on Fuzzing and use concepts e.g. Parameter Inference, Operation Dependency Inference, Model Transformations, Stateful Dynamic Analysis, etc. to get more sensible results than results in random testing.

RAGO API is a first framework that configures OpenAPIs in RAG generates random test requests of an API descriebed in OpenAPI and infers parameters by responses in the API. The OpenAPI model is validated by **974 APIs** selected in the repository of apis.guru[5]. This OpenAPI model could be the basis of published testing approaches introduced in this paper or any other test methods and be extended with JastAdd frameworks released already publicly. We have presented an example of an Fuzzing approach and usage of RAGs in attributes and could generate requests of a pet store[2] and expected responses in this server where **maximum over 300 valid requests were inferred**. In this work, collection attributes were the most helpful attributes, because it simplified the implementation of references used for grammar and parameter inference.

Finally, we have presented several suggestions to extend this tool intended to improve basic functionalities e.g. references with [14] and also to develop new approaches e.g. model transformation with [8], [15], [17]

Bibliography

[1] Andrea Arcuri. “RESTful API automated test case generation with EvoMaster”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.1 (2019), pp. 1–37.

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

[3] Anirban Basu. *Software Quality Assurance, Testing and Metrics*. 2015.

[4] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. “Automatic generation of test cases for REST APIs: A specification-based approach”. In: *2018 IEEE 22nd international enterprise distributed object computing conference (EDOC)*. IEEE. 2018, pp. 181–190.

[5] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 2007, pp. 1–18.

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

[7] Niklas Fors, Emma Söderberg, and Görel Hedin. “Principles and patterns of JastAdd-style reference attribute grammars”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, pp. 86–100.

[8] Sebastian Götz et al. “A JastAdd-and ILP-based Solution to the Software-Selection and Hardware-Mapping-Problem at the TTC 2018.” In: *TTC@ STAF*. 2018, pp. 31–36.

[9] Görel Hedin. “An introductory tutorial on JastAdd attribute grammars”. In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2009, pp. 166–200.

[10] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[11] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. “QuickREST: Property-based test generation of OpenAPI-described RESTful APIs”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 131–141.

[12] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[13] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. “RESTest: Black-box constraint-based testing of RESTful Web APIs”. In: *International Conference on Service-Oriented Computing*. Springer. 2020, pp. 459–475.

[14] Johannes Mey et al. “Continuous model validation using reference attribute grammars”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 2018, pp. 70–82.

[15] Johannes Mey et al. “Reusing Static Analysis across Different Domain-Specific Languages using Reference Attribute Grammars”. In: *arXiv preprint arXiv:2002.06187* (2020).

[16] K.A Saleh. *Software Engineering*. J. Ross, 2009, pp. 224–241.

[17] René Schöne et al. “Connecting conceptual models using relational reference attribute grammars”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2020, pp. 1–11.

[18] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. “RestTestGen: automated black-box testing of RESTful APIs”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 142–152.