RESTAssured: Formally Verifying RESTful API Specification Conformance in Server-side Web Applications

by

Ye Shu

舒烨

Professor Daniel Barowy, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 19, 2024

Contents

List of Figures

List of Tables

List of Definitions

List of Code Listings

Abstract

Modern-day web applications rely heavily on RESTful APIs to facilitate communication between back-end code running on the server and front-end code running on browsers, mobile phones, or desktops. RESTful APIs are usually formally specified so that the back end and front end can be developed independently and in parallel. Therefore, it is critical to ensure that the RESTful API implementations conform to the specification and that the specification is written "correctly"[1].

Traditionally, the specification conformance of RESTful API implementations is approximated via testing, which requires substantial maintenance labor and cannot provide proof of correctness. This thesis aims to formally verify the property. I propose a static white-box analysis tool, RestAssured, that formally verifies a RESTful API service implemented with Express.JS in JavaScript against its OpenAPI specification. Specifically, RestAssured verifies that for each specified endpoint, the implementation must contain the corresponding route handler, and for any valid request defined in the specification, the implementation must also produce a specified response.

Using 19 synthetic benchmarks, we show that RestAssured is accurate in verifying the conformance property and detecting discrepancies. When a violation is found, or when the implementation disagrees with the specification, RestAssured can generate actionable feedback to the user, including concrete counterexamples to help the user understand and reproduce the discrepancy. We also show that RestAssured scales quadratically with the number of endpoints and can handle complicated programs with close to 1,000 branches in less than a minute.

[1] "Correctly" is quoted here, because specifications are by definition always "correct". In this thesis, an "incorrect" specification is either incomplete or (if written after implementation) fail to correctly describe the implementation.

Acknowledgments

I would like to thank my wonderful advisor, Prof. Daniel Barowy, for his guidance, support, mentorship, and words of encouragement. His patience and enthusiasm—along with many spontaneous on- or off-topic discussions of woodworking, Aquinas and Kuhn, and everything about programming languages—have been a constant source of inspiration and motivation. I am forever grateful for all the opportunities he has presented me to grow as a researcher and a person. I would also like to thank my second reader, Prof. Stephen Freund, for his expertise, insights, and thoughtful feedback on this work. This thesis would not have been possible without them.

I would also like to thank the Computer Science Department at Williams College for creating a supportive and intellectually stimulating environment that has allowed me to explore my interests in so many different topics of Computer Science. I want to thank professors Jeannie Albrecht, Duane Bailey, Kelly Shaw, Rohit Bhattacharya, and so many more who have taught, inspired, and supported me throughout my time at Williams.

I would not be who I am today without the support of my family and friends, and the contingencies of life that have connected me with so many wonderful people. I would like to thank my family for their unwavering support and encouragement throughout my life. I would also like to thank all my friends with whom we have cooked together in the Prospect and Fayeweather basement (mostly they cooked and I ate XD). Special thanks to Danny Huang '11, who has been a great friend and mentor to me. I am also grateful for the privilege I have had at Williams outside of Computer Science, including the weekly philosophy salons organized by Prof. Joseph Cruz, the opportunity to see a total solar eclipse and the aurora borealis in my senior year, the trees and mountains of the Berkshires, and the night sky above the Purple Valley free from light pollution. Last but not least, I want to thank Sir Artorias Demetrius "Boop", the cat who is not mine, for his ever-so-soft touch and his ever-so-aloof presence.

Chapter 1

Introduction

If the comment and code disagree, both are probably wrong.

— Bjarne Stroustrup, in CppCon 2017 Opening Keynote [20]

1.1 Motivation

Modern-day web applications are usually separated into a front end and a back end, as exemplified in Figure 1.1. The two are also referred to as the client and the server, respectively. The separation of the two is a perfect example of separation of concerns, a design principle in software engineering that separates a program into different sections, such that each section addresses a separate concern. The front end is the part of the application that runs in user's browsers or mobile phones, directly interacts with the user, and renders the UI given data pulled from the backend. The back end, on the other hand, is the part that runs on the server, processes business logic, and feeds data to the front end. The separation allows the front end and the back end to be developed independently and in parallel, even by different teams with different levels of expertise. Naturally, the two pieces of software need to communicate with each other with some type of interfaces, known as Application Programming Interfaces (APIs). The most common type of API interface for this purpose is the REST API [36]. Contrary to popular beliefs, the REST API is not a well-defined standard, but a set of design principles to create web services that are scalable, stateless, and platform-independent.

In order for the front end and the back end to be developed independently, the API must be well-documented, and the documentation must be kept up-to-date with the implementation. The front-end developers must be able to know what data are to be sent to the back end and what data to expect in the responses, and possible types of errors to be handled when things go wrong[1]. In practice, the documentation is usually written in the form of a formal specification, or a machine-readable and unambiguous format, known as OpenAPI specifications [68] that describes the API's

[1] It is important for the front-end developer to know the set of possible errors, so they can handle them properly. For example, for errors like 401 Unauthorized, 403 Forbidden, the front end should present a nice prompt to the user, e.g. "username/password incorrect", "you do not have permission to view this page". For other errors like token expired, the front end should automatically refresh the token and retry the request.

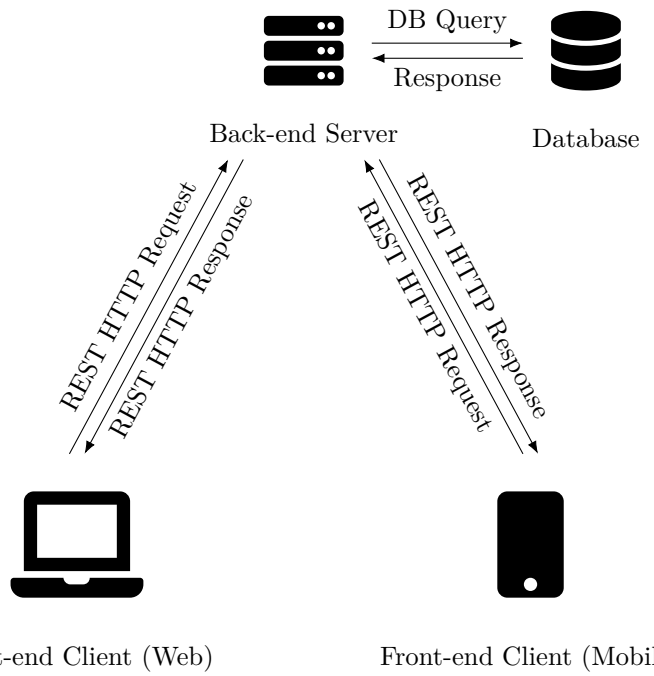


Figure 1.1: The general structure of a modern web application, which separates the front end and the back end. The front-end clients may run on heterogeneous platforms, such as web browsers or mobile apps. The back-end server listens for incoming requests from the front end, processes the requests, queries the database if necessary, and sends back responses to the front end. In this example, the front end and the back end communicate via RESTful HTTP requests and responses.

endpoints, parameters, and responses. Needless to say, for the front end and the back end to work together, the back end must implement the APIs according to the specification, and the specification must be written "correctly".

However, both of these tasks are challenging. It is quite common to write "incorrect" specifications or implement the APIs incorrectly, even for experienced programmers; and it is hard to implement a program fully conformant to specifications. When an implementation disagrees with the provided specification, at least one of these two things (and quite possibly both at the same time, as the chapter epigraph suggests) is happening:

1. The programmer implemented the APIs incorrectly according to the specifications. For example, their code may be unable to handle certain types of input specified; or they may return a type of response not specified.

2. Alternatively, the specifications may be written "incorrectly". Although specifications are by definition the ground truth and always "correct", in reality, specifications are often vague, not kept up-to-date with implementations, incomplete, or even missing or internally inconsistent [17, 55], which we shall collectively refer to as "incorrect". In fact, writing a 'correct' specification is very difficult, and the difficulty is often underappreciated [17].

In either case, the frontend developers would not know how to handle the errors properly, leading to unexpected behaviors in the frontend. To make it worse, these errors often do not surface until the application is in production, and the frontend developers are left to debug the issue with little information. This is the problem we aim to address in this thesis.

In this thesis, we propose a static white-box analysis tool, RestAssured, that formally verifies the conformance property of RESTful API implementations to their specifications. It is a white-box analysis tool in that it takes in both the specification and the source code of the implementation; unlike black-box testing, which only has access to a binary or a running instance of the implementation. It is static in that it does not run the program, but rather examines the source code and executes it symbolically. It is a formal verification tool in that it uses formal methods like symbolic execution to prove the correctness of the implementation with respect to the specification, or prove disagreements between the two (known as "specification nonconformance"). Specifically, RestAssured targets server-side RESTful API implementations written in the library `Express.JS` and specifications written in OpenAPI format. RestAssured builds upon a custom symbolic execution engine, which targets a restricted subset of JavaScript/TypeScript language that is commonly used in `Express.JS` applications, which we define formally and name RestScript. RestAssured is sound, meaning that if it reports specification conformance, then the implementation and specification indeed agrees with each other. However, it is not complete, meaning that it may not be able to prove conformance for some implementations even when it agrees with the specification.

1.1.1 A Minimal, Concrete Example

Real-world server-side code is often complicated. They are loaded with layers of abstractions and indirections; and have to interact with databases, caches, and other external services. To illustrate the problem clearly, we shall use a minimal, concrete example of a RESTful API endpoint for user authentication written in JavaScript, as shown in Listing 1.1.

Consider the server-side JavaScript code in Listing 1.1. The code implements a login endpoint that takes in a username and a password and returns an HTTP 200 OK with success message if the username and password match (L55). When the username or the password is missing, the server returns a 400 error (L49), and when the username and the password do not match, the server returns a 401 error (L57).

This API endpoint is formally specified in the function comments in a format similar to OpenAPI specifications [68, 74]. The comments (L5–L44) describes the endpoint's path, HTTP method, request schema, and all possible responses with each schema. Ideally, the developer should include one such comment for each API endpoint in the project. Then, the actual OpenAPI specification is generated from all these function comments using tools like [97]. The actual OpenAPI specification generated from the example is shown in Appendix A.

However, observe that the specification is incomplete as it omits the case of HTTP 400 response when the username or the password is missing. With an incorrect specification, the frontend developers would not know that the server could return a 400 error. Hence, the frontend developer

```
import express from "express";
const app = express();

/**
 * @openapi
 * /login:
 *   post:
 *     summary: Login to the application.
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               username:
 *                 type: string
 *               password:
 *                 type: string
 *     responses:
 *       200:
 *         description: Login successful.
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 message:
 *                   type: string
 *                   description: A success message.
 *                 username:
 *                   type: string
 *                   description: Username.
 *       401:
 *         description: Login failed
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 message:
 *                   type: string
 *                   description: An error message.
 */
app.post("/login", (req, res) => {
  const { username, password } = req.body;

  if (!username || !password) {  // field missing
    res.status(400).json({ message: "Malformed request" });
    return;
  }

  const user = users.find(/* ... simulate database check ... */);
  if (user) {  // found
    res.json({ message: "Login successful", username });
  } else {  // not found
    res.status(401).json({ message: "Login failed" });
  }
});
```

Listing 1.1: A RESTful API endpoint for user authentication written in JavaScript. The comments provide a formal specification in OpenAPI format. Observe that the HTTP 400 response is left out in the specification, rendering the specification incomplete.

cannot handle the 400 error properly, leading to unexpected behaviors in the frontend. The reader could also observe how complicated the specification already is, even for such a simple API endpoint, and how easy it is to make mistakes in the specification.

1.1.2 Personal Motivation

This work is motivated by my own experience leading and developing web applications for the student club Williams Students Online (WSO)[2].

To share a personal anecdote, in 2022, Prof. Bill Jannen told me that he cannot access the WSO website, which shows a blank page for him. The same issue was reported by a few other users as well. They shared one commonality: they have all been logged in to WSO for over half a year at the time, and it turns out that their logins have expired. WSO issues and uses two types of tokens, a short-lived API token for access into different WSO services and a long-lived identity token attached to the user logins. The short-lived token expires within hours and needs to be refreshed using the identity token. The front-client code is able to refresh the short-lived token automatically, but when the identity token expires, refreshing the short-lived token would fail with a 401 Unauthorized error. The front-end code should have caught this error, deleted the expired identity token from the browser cache, and prompted the user to log in again. However, not expecting this error, the front-end code failed silently, rendering a blank page to the user since it does not have a short-lived token to fetch content from the server with. This led to much user frustration as the issue would persist even if the user refreshes the page, and the user would have to manually clear the browser cache as a workaround. I fixed the bug later by adding in this missing error handling logic [93].

It turns out that the similar issue is present throughout the WSO codebase. In addition to the defined HTTP status codes, many endpoints return custom status codes, such as `1351` for authentication failure, `1401` for user not in `student` scope, and `1404` for user not in `OnCampus` scope. However, these custom status codes are not documented in the OpenAPI specification, and the front-end developers have no idea that these codes may be returned. As a result, the front-end code does not know how to handle these custom status codes properly. This leads to unexpected behaviors in the front end, such as showing a blank page, or redirecting the user to a general error page, without fixing the underlying error. I could in principle go around the specification and add in the missing status codes to the front-end codebase. However, this is not a scalable solution, and the issue would likely resurface in the future. It would be much better if the front-end developers could know with confidence what the server may return, so they will not be caught in surprise when things go wrong.

1.2 Goal and Contributions

In this thesis, we propose a tool named RestAssured that formally verifies the conformance of server-side RESTful API implementations to their OpenAPI specifications. In other words,

[2]The WSO website is accessible at https://wso.williams.edu/. Our codes for the frontend repository and our REST API client are open sourced on GitHub under organization https://github.com/WilliamsStudentsOnline/.

RestAssured checks whether the server-side code has implemented all of the endpoints in the specification, and whether the responses produced by the server, given a valid specified request, are in accordance with the specification. The goal of RestAssured is to help frontend developers to know with confidence what the server may return, so they will not be caught in surprise when things go wrong. If the server implementation is not conformant to the specification, RestAssured will report the discrepancies to the developers with counterexamples to assist them in identifying and fixing the issues.

Our contributions can be summarized as follows:

- We formally define the conformance property with regard to some OpenAPI specifications and some RESTful API implementations.
- In addition to the conformance property, we identify some other properties that indicate disagreements between the implementation and the specification, such as unspecified endpoints and overspecified responses.
- We propose and formally define a restricted subset of the JavaScript language that is commonly used in `Express.JS` applications, which we name RestScript.
- We have built a symbolic execution engine for RestScript, which serves as a key component in RestAssured.
- We have built RestAssured, a static white-box analysis tool that uses symbolic execution to check for conformance of server-side RESTful API implementations to their OpenAPI specifications.
- We evaluate RestAssured on a set of synthetic and real-world benchmarks, showing that it is accurate, produces actionable outputs, and scales to complicated applications with large number of branches.
- We discuss the limitations of RestAssured and propose future works to address them.
- We share our experience in building RestAssured and discuss some of the challenges we faced and lessons we learned.

1.3 Organization

This thesis is organized as follows:

Chapter 1 (this chapter) provides an introduction and motivation behind this work.

Chapter 2 goes into detail on the background of modern Web Services and RESTful API services; the evolution of specifications, especially formal specifications like OpenAPI specifications; and formal verification techniques like symbolic execution.

Chapter 3 formally defines our problem. It formalizes the RESTful API specification; the conformance properties; RestScript, the restricted subset of JavaScript/TypeScript language our static analysis tool targets; and the behaviors of the Express.JS library we model.

Chapter 4 describes the algorithm and general workflow of RestAssured. It details our preliminary work in building a symbolic execution engine for RestScript, including the specific symbolic execution techniques and optimization methods we use. It also describes how RestAssured generates symbolic requests, performs symbolic execution on the server-side code to produce symbolic responses, and checks for conformance with the OpenAPI specification.

Chapter 5 describes the implementation detail of the tool, including how we parse RestScript code, how we symbolically execute the server program as well as request handlers, and how we check the set of responses generated by the server. It also describes how we handle many implementation challenges, such as building an internal model for Express.JS.

Chapter 6 evaluates the tool from three aspects: the accuracy of our verification algorithm, the performance of the tool in practice, and its usability. It also discusses how we build the synthetic benchmarks, drawing inspiration from real-world applications.

Lastly, **chapter 7** concludes the thesis with discussions of limitations, future works, and many takeaways I have learned throughout the process.

Chapter 2

Background

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

— Edsger Dijkstra, 1972 Turing Award Lecture [27]

2.1 Web Services and RESTful APIs

Web services are software systems that provides communication interfaces for other software systems on the Internet. They have played a crucial role in the modern day Web [58, 62]. Most major web services, like Google, Microsoft, Amazon, Twitter, Salesforce, and Slack, have all offered varieties of public web APIs for automated access to their web resources.

Early web services were designed based on the Simple Object Access Protocol (SOAP) [22]. SOAP is a simple XML-based protocol that exchanges web information via HTTP and RPC. A SOAP service is said to be operations-oriented, in that it exposes operations or actions to the user. For example, if a user wants to create a post in an online forum, the SOAP API might expose a function like `CreatePost`.

Over the last decade, web services have rapidly switched to the Representational State Transfer (REST) [36] architectural principles [58, 62]. According to a paper in 2021, 95.8% of the APIs provided by the Alexa.com top 4000 most popular sites are RESTful [71]. Web Services conforming to the REST principles will offer a set of Application Programming Interfaces (APIs). Hence, they are commonly referred to as Web APIs, REST APIs, or RESTful APIs [87]. Unlike SOAP, REST is data-oriented, in that it exposes Web resources named by URLs. In addition, REST is not a protocol but a set of architectural principles. It utilizes the HTTP methods (e.g. `PUT`, `GET`, `POST`, `DELETE`) on a given URL and maps them to the respective CRUD (Create, Read, Update, and Delete) operations. Thus, for the same example where a user wants to create a post in an online forum, the RESTful API may expose an API endpoint `/api/v1/post`, and the user can send an

HTTP `PUT`[1] request to create the post.

In more recent years, the use of RESTful APIs have surged due to the emergence of Software-as-a-Service (SaaS), which prompted developers to leverage third-party software. For example, a developer may use Stripe APIs to handle payments, or invoke Slack APIs to send push notifications [43]. According to a few industrial report and surveys [80, 85, 95] conducted in the past two years, 75.1% of participants across all industries have worked on internal APIs, 53.9% of participants have worked on third-party APIs, and 75.7% participants report that the API economy is or will be a prioritization of their organization [85]. 86% respondents have used REST, whereas only 26% used SOAP [80].

With great popularity, the correctness of RESTful APIs have also become a concern more than ever before. In Smartbear's report, 78% of respondents found API Quality to be Very Important or Extremely Important [95]. However, according to Postman's report, only 45% of respondents have reported their deployment failure rate to be less than 5% [80]. Along the process, problems of RESTful APIs have also surfaced. Since REST is only a set of architectural principles without formal standards or specifications, it is up to the API service developers to make many crucial decisions. In many cases, this lead to poor design decisions, such as using the same HTTP method for retrieval and deletion, inconsistent naming conventions, and more importantly, lack of a standard documentation means [71]. In the 2023 report published by Postman, the most popular API developing tool, 52% of respondents said lack of documentation was the biggest problem for them when using the APIs of others [80]. According to another report published by Smartbear, the company behind Swagger, in 2023, developers report that "confidence in documentation provided for API consumers has decreased year over year", where less than half of the respondents report that they have good documentation for their consumers [95, p. 54].

2.2 Formal Verification

Ensuring the correctness of software is a long-standing problem in computer science. Today, the most common approach to ensure correctness is testing. We may write or systematically generate different types of test cases (unit tests, stress tests, proper-based testing, fuzzing) to examine the software's behavior under different inputs and conditions. However, testing can only show the presence of bugs, but not their absence. Instead, one should seek a proof of the program's correctness, as Dijkstra famously claims in his 1972 Turing Award lecture (the original quote is included as the epigraph of this chapter) [27].

Another approach to ensure the correctness of a program once and for all is formal verification. Formal verification refers to using mathematical logic and automated tools to prove that a program satisfies a certain property. In our case, we aim to prove the property of specification conformance. Usually, formal verification involves two main steps, formalization and verification. During formalization, the program is first translated into a representation amenable to automated

[1] According to the REST principles, the API should always use `PUT` for creation, and `POST` should be reserved for updates only. However, many services use HTTP `POST` for all write actions, including creation, update, and deletion [71]. In reality, the user might be asked to send an HTTP `POST` request instead.

reasoning. Then, automated tools such as theorem provers and model checkers are employed to prove the property of the program leveraging the formal model.

2.2.1 Genealogy of Formal Verification

The idea of formal verification, *i.e.* proving that programs are correct using math and logic, has been around for a long time. Some claim it can be traced back to the pre-electronic phase of "Charles Babbage's and Ada Lovelace's work on the difference and analytical engines" to verify "the formulae placed on the [operation] cards" [8]. A perhaps more complete and workable method is presented by Alan Turing in 1949 to prove the correctness of a factorial algorithm based on additions [69, 104]. However, it is not until the 1960s and 1970s that systematic techniques are developed by Peter Naur [70], Robert Floyd [38], and C.A.R. Hoare [46].

Among all formal systems, the Hoare logic [46] (sometimes also known as Floyd-Hoare logic) is perhaps the most influential. At its core is a notation system later known as the Hoare triplet:

$$P\{Q\}R$$

where P is the precondition, Q is the program, and R is the postcondition, or the result of execution. According to Hoare, "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [46, p. 577]. It shall also be noted that the Hoare logic only concerns partial correctness, *i.e.* if the program terminates, then the postcondition should be satisfied. To establish total correctness, one must also prove that the program terminates. The termination proof is a much harder problem and cannot be proved for all programs since the halting problem is undecidable. Therefore, a better way to understand the Hoare logic is: if the precondition P is true, and the program Q terminates, then the postcondition R will be true.

2.2.2 Formal Verification Techniques

There exists a wide array of formal verification techniques. Formal verification techniques can be broadly classified into three categories: model checking, theorem proving, and symbolic execution [72, 107]. This thesis will focus on symbolic execution, as it is the most suitable technique for verifying RESTful APIs.

Model checking [16] checks a finite-state model of the target system to ensure that it satisfies a specification expressed using propositional temporal logic like Linear Temporal Logic (LTL) [79]. It does so by searching through all possible states of the system, usually treated as a state transition graph, to check if the property holds in all states. However, model checking often requires some form of abstract and finite-state model of the system, which can be hard to construct for complex systems. As a result, it often requires the programmer to provide the model manually or at least some annotations for the automatic synthesis/construction of the model. For systems that are constantly evolving, like web services, keeping the models and/or annotations up-to-date with every change in the target system can be a daunting task and may not be feasible. At the end of the day, model checking is only as good as the model itself. It only reasons about a model that may not fully

capture necessary details of the target system [77].

Theorem proving, on the other hand, transforms the program and its specification into a set of logical formulas. It often requires the programmer to write some machine-checkable proofs to prove to the theorem prover that the program satisfies the specification. Theorem proving is often more powerful than model checking, as it supports a richer set of formalisms that are not in principle solvable in automated methods like model checking. Some of the most well-known theorem provers and proof assistants include Coq [19], Isabelle [73], and Lean [25]. There also exist verifiers that do not require the programmer to write proofs, such as ESC/Java [37], Dafny [57], and the deductive proof plugin (WP) of Frama-C [21]. In these cases, the programmer need only provide annotations for their programs; such as preconditions, postconditions, and invariants; and the verifiers will generate the proof obligations and check if they are satisfied using an automated theorem prover like Z3 [24] or Alt-ergo [18].

Last but not least, symbolic execution systematically explores all possible execution paths of a program to reason about the program's behavior or to generate concrete test case inputs that lead to certain program behaviors. [4]. It executes programs with symbolic inputs instead of concrete inputs, either statically or dynamically. It may even concolically execute the program, by combining symbolic execution with concrete execution for certain parts of the program [91]. The symbolic execution engine represents a program's execution path as a symbolic formula (known as *path constraints* or *path predicates*). Eventually, a model checker, typically a SMT solver, is used to check if the program satisfies the desired property along each execution path, and whether the path constraint is satisfiable, thus whether the path is reachable.

Of the three types of formal verification techniques, symbolic execution is the most suitable for building a low-effort RESTful API specification conformance verifier. Model checking often faces fidelity issue as the models are often overly simplified and does not accurately represent the actual target system running on actual machines interacting with its environments. Alternatively, the programmer can manually construct formal models of the system, which is time-consuming and can be hard for most programmers who have not received formal training in logic and formal methods. Theorem proving similarly requires the programmer to write machine-checkable proofs or annotations, which is even more daunting and time-consuming. Symbolic execution, on the other hand, only requires the program and the specification. The symbolic execution engine will automatically generate all possible execution paths from the program and check if the program satisfies the specification. However, while symbolic execution does not suffer from fidelity, it does suffer from scalability issues, which we will need to address.

It is a commonly shared impression that formal verification methods are very costly and time-consuming, and are not practical for most real-world software development other than critical systems like avionics, medical devices, and nuclear power plants [77, 82, 86]. However, perhaps every programmer is using some formal systems on a daily basis. The most common, yet often overlooked, formal system are type checkers. Type checkers ensure that the program is free of type errors. For example, a type checker ensures that an integer cannot be used in the place of a string, or that a function expected to return an array does not return a dictionary. Like other types of formal verification techniques, the programmer is often required to provide some form of annotations (in

the form of type declarations) to the type checker. For example, in TypeScript, the programmer must provide type annotations as shown in Listing 2.1.

```
function add(a: number, b: number): number {
  return a + b;
}

let x: number;
x = 1;           // OK
x = '1';         // Error: Type 'string' is not assignable to type 'number'.
x = add(1, 2);   // OK
x = add(1, '2'); // Error: Argument of type 'string' is not assignable to
                 //        parameter of type 'number'.
x = add(1);      // Error: Expected 2 arguments, but got 1.
```

Listing 2.1: Example of TypeScript type annotations and type errors

Unfortunately, type checking alone is not sufficient to ensure the specification conformance of a program. This is because the RESTful API specification not only specifies the types of the input and output, but also (sometimes) specifies value constraints, and behaviors the program. For example, the specification may require a field to be an enum of a certain set of values, or a number to be in a certain range. These attached predicates are only expressible in a refinement type system [42]. In addition, the specification may require the program to authenticate the user with a token, or to return an HTTP 401 response if no token is included/token is invalid; which requires a dependent type system [109]. These type systems are much more expressive than the nominal type systems used in most languages, however they are also less decidable and require more power techniques to check against. As a result, we choose to employ more powerful formal verification techniques like symbolic execution to ensure the specification conformance of RESTful APIs.

2.3 Documentation and (Formal) Specifications

Documentation and specifications are traditionally deemed as separate documents. Documentation are written for humans, either consumers or subsequent maintainers of the software, to understand and use the software. It is vaguely defined and exists in many forms, architectural description of the software, examples, step-by-step tutorials, design decisions, user requirements, ...[26]. In this thesis, we focus narrowly on the user-facing documentation. For example, quick start guides, functional descriptions on how to use the software, and interactive interfaces for developers to run example API calls [49]. On the other hand, specifications are usually part of the design process, specifying how the software is to behave, and the formats for input/output [49]. They are usually consumed by the developers implementing the software and testers carrying out functional tests. Sometimes, they are written in machine-readable formats, so the testing process is automated by machines rather than human.

However, the line between documentation and specification are blurred in the field of Web APIs, with the emergence of agile development and the need to ship products rapidly within a limited

timeframe. In recent years, many developers, especially those in small group settings, have opted to use informal channels instead of well-formed documentations [26]. In these settings, documentation and specification are no longer hand written by human. Thanks to the automation technology commonly deployed in present-day RESTful Web API development, developers often generate documentation [101] and specifications [97, 100] automatically from code comments. For many small teams, these code comments are often written alongside, or even after their software implementation, instead of generated in a separate design phase. Thus, their specification are generated afterwards rather than beforehand during design phase. Therefore, it is the codes rather than the specifications that genuinely reflect the intentions and behaviors of the API designers and/or developers. Some even argue that these API "specifications" are instead API "definitions" [49]. In this thesis, I shall continue to use the term "specification", but I ask the readers to bear in mind when they see this word and how its meanings have subtly changed in the world of Web APIs.

2.3.1 Documentation

Writing good documentation has long been a concern for the software engineering community [17, 26, 39, 66]. Developers have always disagreed with each other on what constitutes a good documentation. Some developers favor detailed documentation, so they may continue to maintain legacy software products or migrate them easier. This view has long been the mainstream and incorporated into best practices for development and maintenance. Others developers, under the influence of agile development, rely more on informal communication rather than documentation. Nonetheless, most of them can agree that "specifications and documentation are frequently missing, vague, or outdated" [17].

Documentations exist in very different forms. Although many researchers have traditionally emphasized the importance for documentation to describe the hierarchical architecture of the system [102], surveys have surprisingly shown that such documentation are less valued by developers. Instead, "source code and comments are the most important artifact to understand a system to be maintained. Data model and requirement description were other important artifacts" [26].

In the case of RESTful APIs, the most popular form of documentation is usually an interactive web page generated by Swagger UI [101] from an OpenAPI specification (which is often generated by Swagger from code comments). It is a web interface that lists each API endpoints, their respective input parameters, request bodies, and all possible responses. For each possible response, it lists the HTTP response code, the description, and an example of returning values. For example, Figure 2.1 shows the Swagger UI documentation generated from the Motivating Example in Listing 1.1. Furthermore, it allows developers and consumers to interactively send requests to the API endpoint and view the responses, as shown in Figure 2.2. When encountering undocumented responses, Swagger UI will also display an "undocumented" warning next to the response code, as shown in Figure 2.2b.

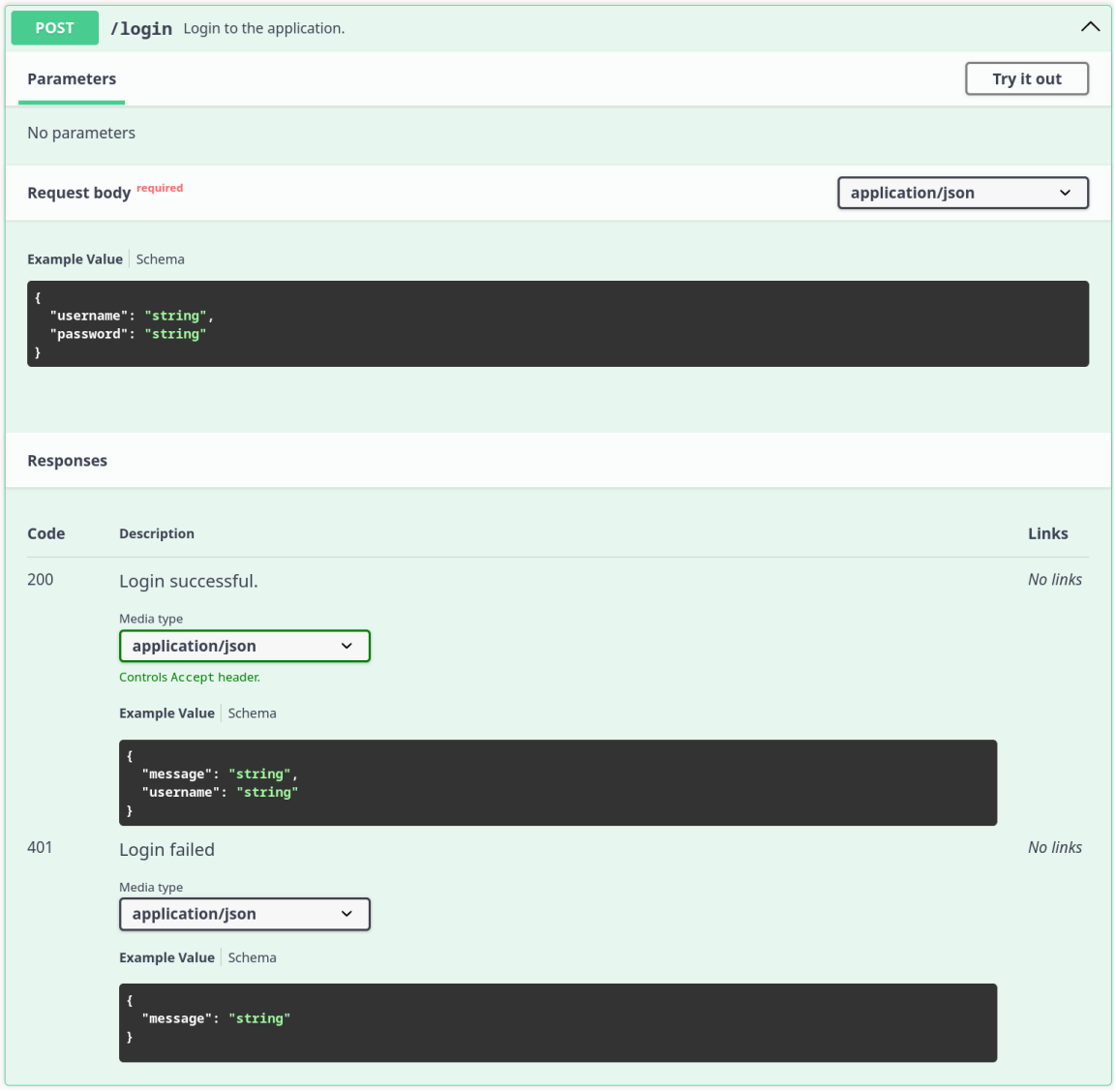
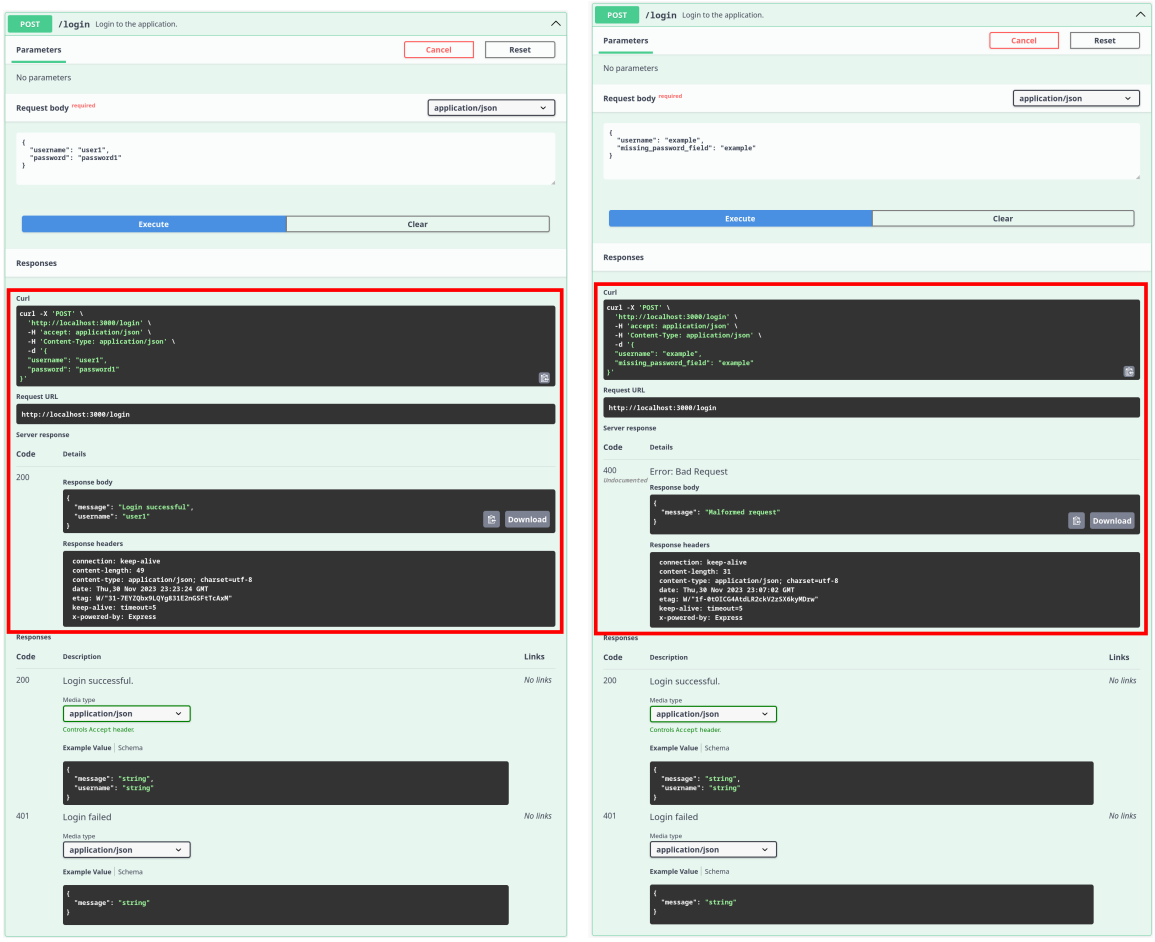


Figure 2.1: Swagger Web UI documentation generated from the Motivating Example in Listing 1.1. It documents the API endpoint, input parameters, request body schema, and all possible responses (HTTP code and response body schema).



(a) In this case, the API returns an HTTP 200 response with the expected response body.

(b) In this case, the API returns an undocumented HTTP 400 response with response body.

Figure 2.2: Swagger Web UI allow users to interact with API endpoints by sending requests and displaying responses. When encountering undocumented responses, a warning will appear (2.2b).

2.3.2 Traditional Role of Specifications in Software Engineering

The word "specification" is heavily loaded in the software engineering community and can be used to refer to many things. At its core, a **program specification** is a statement whose role is to say: (1) what purpose the program serves, and (2) how the program can be correctly used [56, p. 279]. Traditionally, specifications are written by developers or specialized software architects during the design phase of the software, before it is implemented. Software developers following the traditional software development lifecycle will often employ the waterfall model, composed of a series of steps: Software Concept, Requirement Analysis, Architectural Design, Detailed Design, Coding and Debugging, System Testing [65, p.139]. Depending on the stage of development, specifications can be written in different levels of details. That includes requirement specifications, product design specifications, or detailed design specifications, in the order of increasing details [65, p. 168]. However, this puts a lot of burden on the developers to keep the specifications up-to-date with the refined design and changing requirements. "For a rapid-development project, … updating the specification can become a full-time job" [65, p. 139].

In more recent years, the software engineering community has adopted agile or rapid development methodologies to ship software products faster. Many of these approaches attempt to combat wasted time on specifications by employing a minimal specification approach, even to the extent of using "[u]ser manual as spec" [65, pp. 323–329]. In other words, use the documentation as the specification. This approach has clearly been adopted by the modern RESTful API developers, where the lines between documentation, specification, and definition have been blurred by automated tools like Swagger [49]. These automated tools generate documentation and specifications from code comments, which are written alongside the code implementation. Thus, the meaning of "specification" has changed subtly from what it used to be in the traditional software engineering community. Although the sense of describing program behavior is preserved, it is no longer produced in a clearly separated requirement analysis and design phase [49, 110].

2.3.3 Genealogy of Formal Specifications

Another debate to be had within the software engineering community is the choice between formal and informal specifications [55, 56]. Formal specifications are written in some variant of formal notational system, which is usually a programming language like TLA+ [54] or a program logic like LTL [79]. On the other hand, informal specifications are typically written in plain natural language. Formal specifications are often considered to be more precise and less ambiguous, given that internal inconsistencies and contradictions can be mathematically detected. Moreover, they can be understood by machines programmatically, which allows for automated formal verification. On the other hand, informal specifications are written for a single type of consumer: humans. Without the formalism constraints, they may contain diagrams, overviews, and examples that are not easily expressible in formal notations but facilitate human understanding.

The concept of formal specifications has been around since the 1970s [5, 27]. However, formal specifications and formal verification still remains a foreign concept to many software developers to this day, and is rarely adopted in practice. Formal specifications had its roots in

classical mathematics and logic, whereas software developers see themselves more as engineers than mathematicians, thus the gap between the two communities [77, 86]. In fact, not just formal specifications, even informal specifications are known to be hard to write. Customers may change requirements over time or are unable to specify what they want, the problem may be ill-defined, or the intended solution may be too complex to be fully understood [65]. When specifications are formalized, the challenge is even greater. It can even be said that "[w]riting a 'correct' [formal] specification is very difficult - probably as difficult as writing a correct program." [55, p. 150]. On the extreme end, [56] argues that specifications are necessarily informal, because writing a formal specification involves the same formalization process as the implementation. It necessitates a formal semantics that must be defined *a priori*, thus requiring the writer to give a precise definition of the problem and the solution. However, such formal semantics is often foreign to the problem domain, and must accompany some representational conventions (read: informal natural language) to be understood by humans. Moreover, the requirement for formal semantics presupposes the existence of an already fully understood theory of the problem. In reality, the writer often do not have a full understanding of the problem and the solution before the program is implemented [56]. All of these factors combined have made formal specifications very hard to write, and often not worth the effort for traditional general purpose software. For example, formally verifying the stack-space bounds in C programs turns out to be a nontrivial task, requiring a complicated formal model specifying the stack consumption of C programs [12]. Microsoft provided .NET users with "Code Contracts" that allow them to formally specify the pre- and postconditions for .NET APIs, which is then statically verified [33, 34]. However, this endeavor is abandoned in production, after .NET developers discovered that despite its powerful formal capacity, "overwhelming usage is about null handling" and decided to replace it with nullable-reference types [48].

2.3.4 Formal Specifications for RESTful API

RESTful APIs happen to fall within a niche spot where all behaviors conform to the REST principles and operates over a well-defined HTTP communication channel. Moreover, the input and output are commonly encoded in JSON format, which is a well-defined data model with a formal schema and format [9, 108]. Thus, a formal semantics for the behaviors of RESTful APIs can be defined in a relatively straightforward manner. Hence, writing a formal specification for RESTful APIs is greatly simplified and becomes a feasible task. In many cases, they are automatically generated from comments in the source code, which are written in a formal format [74]. These comments are then used to generate both specifications and human-readable, many times even interactive, documentations. According to a paper in 2021 measuring the APIs provided by Alexa.com top 4000 sites, "nearly half us[e] software to generate the documentation …, which are, for most cases, also compatible with the OpenAPI specification" [71, p. 11].

OpenAPI specification (formerly known by its product name "Swagger") [68] is the most popular choice of specification format for RESTful APIs [17, 44, 50, 64]. This is also confirmed by industrial surveys in 2023, in which around 60%–80% of the respondents reported using OpenAPI as their API specifications [80, p. 52][95, p. 21].

At a very high level, the OpenAPI specification is very similar to the classical Hoare triplet introduced in subsection 2.2.1. Since a RESTful API service often has multiple endpoints uniquely defined by their URL/URI paths and HTTP methods, we can think of each API endpoint as a program. As a result, the overall specification would be a giant list of specifications for each API endpoint in the service combined. For each endpoint, the endpoint will expect some inputs, either in the form of URL query parameters, request body, or headers. These are equivalent to the preconditions in the Hoare triplet. Take the motivating example in Listing 1.1 as an example, the API endpoint expects a request body with a JSON object containing the fields `username` and `password`. The implementation of the request handler (and potentially middlewares along the path of execution) will then serve as the program. When the program halts and returns a response, the specified response body and status code will be the postconditions, albeit there will be a list of valid responses (rather than one). In the motivating example, the API endpoint is expected to return an HTTP 200 response and a JSON object with a field `message` and `username`, or an HTTP 401 response with a JSON object with field `message`.

2.3.5 Details of the OpenAPI Specification

In this thesis, we target the OpenAPI specification version 3.1.0 [68]. OpenAPI specifications are JSON objects typically represented in `json` or `yaml` files. However, in reality, they are often described as code comments alongside the implementation. The specification files are then generated by automation tools from these comments [17, 97, 100]. An example is given in the motivating example at Listing 1.1, where the specification is written in the comments above the API endpoint definition. The generated full specification file is provided as Listing A.1 in Appendix A.

These specifications are often composed of the following parts: Metadata `info` of the API's title, version, description; `Server` addresses (omitted in Listing A.1); and individual API endpoints found by their `path`. For each API endpoint, the following information should be provided:

1. The HTTP path and method (`GET`/`POST`/`DELETE`/…)
2. A human-readable `summary` and `description`
3. List and type of `parameters`, which can be passed in via URL paths, query strings, headers, or cookies
4. A `requestBody` if one is Anticipated, and its object schema.
5. A list of all possible `responses` with the following fields.
   (a) HTTP response code (e.g. 200 OK, 401 Unauthorized, 500 Server Error)
   (b) `headers` of a list of expected headers with their object schema.
   (c) A human-readable `description` of this response
   (d) The `content` of the response body, and its object schema. In OpenAPI 3.0 and newer, one can even specify multiple alternative schema with `oneOf` keyword.

Then, the object schema can either be defined in-place, or defined separately with a name and re-used in different places. To define an object schema, the following fields are usually required (taking JSON as an example, which is the most common API object schema).

1. The media type of the object, such as `application/json`
2. A `type` field specifying whether one `object` or an `array` of the specified object is returned/required
3. A `properties` field that lists all the names in the JSON object, each with the following fields
   (a) `name` of the field
   (b) `type` specifying the type of the value
   (c) `format` specifying whether a binary-encoded file is expected
   (d) A human-readable `description`
   (e) Some human-readable `example`, that could be potentially used for test generation.

2.4 Specification Conformance for RESTful APIs

Specification conformance has always been a huge problem. There have been work ever since the 1970s [27]. However, it is not until recently, with the popularization of Web Services and Software as a Service (SaaS), that specification conformance has become a topic of widespread interest. Specification conformance is an important topic of interest to both academia and the software engineering industry. For industrial programmers, it is important to write programs that conform to specifications, so their products are successfully delivered to their clients (whether external customers or internal). More importantly, in the case of web applications and services that anticipate programmatic communication between software systems, building systems correctly according to specifications become more important than ever before. Otherwise, the error within a single system will likely result in a catastrophic chain of errors that propagate along the dependency chain, especially if the API consumers are not built robustly against errors.

Research work have flourished, with solutions proposed from both academia and industries. Here, we shall briefly categorize them into two types: testing-based approaches and formal approaches. In the world of RESTful API conformance, the specification needs to be followed in two separate systems: the server-side, or the producers/providers of the API; and the client-side, or the consumers of the API. Hence, the solutions can be further categorized into client-side solutions or server-side solutions. For client-side, there already exist mature solutions to generate client SDK sending requests and handling all specified response cases given an OpenAPI specification [99]. Although the tool also generates server stubs, it is largely insufficient and at best provides a starter template that saves programmers from re-typing boilerplate codes. At server side, the complicated business logic and interaction with complicated dependencies (e.g. database operations, time or file dependent operations) make guarantees hard to come by. In this thesis, we shall primarily focus on the server-side problem, which is more complicated and more interesting.

2.4.1 Testing-based approaches

The current standard approaches in industries are all based on testing. According to recent industrial surveys, 68%–81% API developers employ testing, and the most testing types are functional and integration testing [85, 95]. However, most of the industries relied on human testers and Quality Assurance engineers, and some also uses unit tests [95]. Although there exists automated testing tools and fuzzers ([94], by the same company behind Swagger), their industrial adoption seems limited.

Inside academia, the most active methods for specification conformance is also black-box testing. In a black-box testing, the tester or testing software treats the API provider codebase as a black box and generates test cases directly from the OpenAPI specification. Then, they run the test cases against the implementation, and observe its outputs, then compare the outputs to the specification to determine if the implementation conforms to the specification. Within the software engineering community, researchers have used a variety of traditional test generation techniques. These techniques can be further broken down into two usages: generating test data, and generating test cases. For test data generation, some use example values provided in the OpenAPI specification, some uses fuzzing, some uses random inputs, and some uses custom data generators, or contextual data extracted from previous API responses. For test case generation, some use random testing that assign random values to parameters, some try to satisfy inter-parameter dependencies, or employ property-based testing. Regarding test oracles, most of the works only check for API server failures (indicated by HTML 5xx error codes), some also check for response specification conformance [45, 50, 64].

The black-box method is widely popular because no information about the server implementation is required. Most of the commercial web services only provide formal specifications at best and do not provide any information about the server implementation. More commonly, they provide human-readable documentation that are not machine-readable to avoid excessive resource usage from automated tools. For these cases, third-party clients can only depend upon trust or black-box testing approaches to gain confidence in the correct behavior of the APIs.

However, program testing as an approach to show the absence of bugs is ineffective. As quoted in the epigraph of this chapter, "[p]rogram testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness" [27].

2.4.2 Formal approaches

Let us now turn to the application of formal approaches in the field of RESTful API specification conformance. The software engineering community is primarily only interested in testing-based approaches, whereas formal approaches are mostly presented by the programming language community. However, work in this direction is generally lacking. There are some limited works that employ formal verification techniques on the client-side [83, 106]. However, the client-side code logic is a lot simpler when compared to server-side. These codes are usually a single function call (sometimes encapsulated within an SDK) that authors and sends an HTTP request, and parses its

corresponding response. In most cases, the returning response can be trivially parsed according to the specification. There even exist automated tools to automatically generate client SDKs according to OpenAPI specifications [99].

However, work on formal verification of server-side code has been a blank to the best of my knowledge. Extending the existing approaches from client-side to server-side is nontrivial, given that the server side contains a lot more complicated business logic and interacts with third-party middlewares and databases. In addition, verifying the specification conformance property requires some nontrivial addition to "standard" verification techniques, and requires the verifier to have a good understanding of the specification, how server-side code processes the input request, and how responses are stored and produced. Although Swagger CodeGen [99] is able to generate a server-side code stub, it only provides a bare starting template. In production-grade server codes, the API response controllers usually have complicated business logic, depending on the type of input and result interacting with database, sometimes even invoking different functions and returning different objects. A recent survey paper has suggested employing white-box type-based approaches that use type checkers to check for generated responses against the schema defined in specification [17]. However, we argue that nominal type-based approaches does not fully solve the problem, as it accepts incomplete implementations of a specification and may be unable to handle complicated branching, inter-procedural dependencies, and conformance to specification's refinement types. In response, we shall investigate white-box symbolic execution to provide guarantees to consumers of the API.

2.4.3 Limitations and Challenges Anticipated

When compared to testing approaches, formal verification is a lot more powerful, as it provides true guarantees of the absence of bugs. However, it also comes with many limitations in its usage and adoption, which we will need to address in this thesis. Some of them are inherent to the formal verification techniques, while some are specific to the RESTful API domain.

Interoperability and Cross-Language Operations RESTful APIs are designed to be language-agnostic, and can be consumed or produced by any programming language that can send HTTP requests. As a result, the specification is often written in a language-agnostic format, with generic types like `string`, `number`, and `object`. However, the formal verification tools are usually designed for a specific programming language, and we may need to map the generic types to the specific types of the programming language. For example, OpenAPI may require a field to be of type `integer`, but JavaScript does not have an integer type and represents all numbers as floating-point numbers. There exists past work, Doppio [105], which is a JVM (Java Virtual Machine) written in JavaScript and runnable in browsers, that aim to solve this problem; but it is out of scope for this thesis. Furthermore, the RESTful API operation may involve complex data structures like dates, times, or binary files, which are not easily representable in all programming languages.

Interaction with Nondeterministic Environment A web application is a complex system that interacts with many external components. For example, the server-side code usually interacts

with the database (via SQL queries or ORM libraries), third-party middlewares (e.g. for request authentication). Sometimes, the server-side code also interacts with the file system, the time, the system random number generator, or other external services that are often nondeterministic. To fully verify the specification conformance property, we need to model these external components and their interactions with the server-side code, which is sometimes infeasible or even impossible (if the system is closed-source).

State Space Explosion While symbolic execution is able to remain faithful to the actual running program (unlike model checking methods), it suffers from scalability problem during the analysis. The number of execution paths will increase exponentially with regard to the number of branches. Given the complicated business logic of web applications, the number of execution paths can be very large. Moreover, there often exists loops and recursion in the code, which makes static analysis harder and may lead to infinite execution paths without further information of loop invariants, e.g. the loop counter's range, which may help us establish an upper bound on the number of iterations.

No Off-the-Shelf Static Symbolic Execution Engines Unfortunately, no off-the-shelf static symbolic execution engines exist for JavaScript/TypeScript, our target languages. There are some academic research projects that attempted to build symbolic execution engines for JavaScript, such as Kudzu [90], ExpoSE [60], Cosette [89], and JaVerT/Gillian-JS [40, 61]. Unfortunately, Kudzu and Cosette is not open sourced. ExpoSE is a dynamic symbolic execution framework, which is not suitable for our use case. Lastly, JaVerT/Gillian-JS was broken at the start of my thesis [2]. Hence, we may have to build our own symbolic execution engine from first principles.

Chapter 3

Formalism

工欲善其事、必先利其器。
The craftsman, who wishes to do his work well, must first sharpen his tools.

— 孔子,《论语·卫灵公》
Confucius, The Analects (15.9) [96]

3.1 Overview

In this thesis, we develop a tool and algorithm, named RestAssured, that automatically checks the conformance of a RESTful API server implementation to its OpenAPI specification. At a high level, the tool accepts two inputs: the server-side code and the OpenAPI specification. It should tell us whether the server-side code conforms to the OpenAPI specification, and if not, give helpful information to help the programmer find the non-conformance. The tool works by analyzing the server-side code using symbolic execution, a powerful program analysis technique that can explore all possible execution paths of a program.

This chapter formalizes both of inputs and paves the way for the rest of the thesis. We first formalize the OpenAPI specification and formally define what "specification conformance" means in this context and how we check for them (section 3.2). Then, we formally define the programming language our tool targets, which we name RestScript and is a subset of JavaScript/TypeScript, and the library our tool targets, which is Express.JS (section 3.3).

3.2 RESTful API Services and OpenAPI Specification

As the name suggests, RestAssured is designed to work with RESTful API services, the predominant form of web services today. A RESTful API service is a web service that follows the principles of Representational State Transfer (REST) [36]. Like any other web service, a RESTful API service is provided by a web backend server that accepts HTTP requests from frontend clients (also known as consumers) running on heterogeneous platforms, including web browsers or mobile

apps. After processing a request, the server sends back an HTTP response to the corresponding client, as shown in Figure 1.1. The only difference is that RESTful API services follow a set of design principles to make better use of the HTTP protocol, such as using HTTP methods to represent different operations, using URIs to represent resources, and using status codes to represent the result of the operation. Requests and responses are usually some JSON object with a defined schema, which is then parsed by the recipient.

3.2.1 A Formal Definition of OpenAPI Specification

The detail of the OpenAPI specification is discussed in subsection 2.3.5. In short, it is a machine-readable document that describes the RESTful API service, including the available endpoints, the HTTP methods they support, the request and response schemas, and the status codes they return. They can be treated as a formal language describing a set of REST endpoints provided by the REST API service. For each of the endpoints defined, the server being specified is expected to accept some HTTP requests (which are inputs to the server) and produce some HTTP responses (which are outputs from the server). Let us now denote the specification formally, starting from the primitive types in OpenAPI.

Definition 3.1 **OpenAPI Type**

An **OpenAPI Type** T is a type defined in the OpenAPI specification [98]. It can be one of the six basic types: `string`, `number`, `integer`, `boolean`, `array`, or `object`. It can also be a union type, such as `string | number`.

Moreover, the type can be refined with constraints. For example, an `integer` type can be further constrained to be a multiple of 10, a `string` type can be constrained to have a minimum length of 8 characters and a maximum length of 64 characters, or another `string` type can be constrained to be a UUID or a RFC3339-formatted date, or even be matched by a custom regular expression.

In the case of an `object` type, the type can be refined with an **object schema**, or a set of field names and types for each field. We shall denote the object schema as $N \rightharpoonup T$, a partial function that maps field names N to their types T.

Definition 3.2 **Request Schema**

A **Request Schema** R is a tuple $R = (Q, H, B)$, where:

1. $Q = N \rightharpoonup T$ is a partial function that maps defined **query parameter** names N to their OpenAPI types T.
2. $H = N \rightharpoonup T$ is a partial function that maps defined **HTTP headers** names N to their OpenAPI types T.
3. $B = \mathcal{M} \rightharpoonup T$ represents the **HTTP body content**. It is a partial function that map the MIME media type $\mathcal{M}$ (e.g. `application/json` or `text/plain`) to their OpenAPI types T of the body content. These types can either be an object with defined schema, or a singular primitive type (string, number, or boolean), depending on the MIME type.

Definition 3.3 **Response Schema**

A **Response Schema** P is a tuple $P = (C, H, B)$, where:

1. C is the HTTP response status code of the response (e.g. 200, 404, 500, etc.).
2. $H = N \rightharpoonup T$ is a partial function that maps expected HTTP header names N to their OpenAPI types T.
3. $B = \mathcal{M} \rightharpoonup T$ represents the possible HTTP body content. It is a partial function that maps the MIME media type $\mathcal{M}$ (e.g. `application/json` or `text/plain`) to the OpenAPI type T of the body content. These types can either be an object with defined schema, or a singular primitive type (string, number, or boolean), depending on the MIME type.

Definition 3.4 **RESTful API Endpoint**

A **RESTful API Endpoint** $\mathcal{E}$ is a tuple $\mathcal{E} = (M, U, \mathcal{H})$, where:

1. M is the HTTP method (e.g. GET, POST, PUT, DELETE, etc.) of the endpoint.
2. U is the URI path of the endpoint, and may contain path parameters, such as `/users/{id}`.
3. $\mathcal{H} = R \times P$ is the endpoint handler. It is a set of functions that maps some request schema R to a set of possible response schemas P.

Definition 3.5 **RESTful API Specification**

A **RESTful API Specification** $\mathcal{S} = \overline{\mathcal{E}}$ is an array of RESTful API endpoints $\mathcal{E}$.

It should be noted that the HTTP method M and the URI path U together uniquely identify an endpoint. That is, $\forall \mathcal{E}_1, \mathcal{E}_2 \in \mathcal{S}$, if $\mathcal{E}_1 \neq \mathcal{E}_2$, then $M_1 \neq M_2$ or $U_1 \neq U_2$.

In addition, a real-world specification also contains some metadata, such as the title, description, or concrete examples for the request/response schemas, the API version, or the server URL. These properties are irrelevant to the conformance property and thus omitted.

With Definitions 3.1 through 3.5, we now have a notational system to express what the OpenAPI specification is able to describe and specify. Given the amount of notations we have used, we list a table of symbols in Appendix D for easy reference.

3.2.2 An Example of a Real-World OpenAPI Specification, Formalized

We will now illustrate how the formal specification corresponds to concrete HTTP requests (inputs to the server) and responses (output from the server). For example, consider the API endpoint and its specification given in the motivating example Listing 1.1. It describes a `/login` endpoint that accepts a POST request with a JSON object containing the `username` and `password` fields. If the authentication is successful, the specification states that the server should respond with HTTP code 200 and another JSON object containing the `message` field and the `username` field.

Suppose we know a pair of valid credentials `username`: `ys5` and `password`: `my-secret-password`. We can send an HTTP POST request to the `/login` endpoint with the corresponding JSON object

containing the credentials, as specified by the specification. The HTTP request is shown in the plaintext ASCII format[1] in Listing 3.1, side by side with the OpenAPI specification. Given the valid credentials, authentication will be successful. Hence, the server will respond with HTTP 200 OK and another JSON object, as specified in the specification. The actual response is similarly shown in plaintext in Listing 3.2. Both of the HTTP request and response are valid sentences in the language defined by the OpenAPI specification, as shown by the highlighted fields in both HTTP packets and the OpenAPI specification.

Observe that some fields are not specified by the OpenAPI specification in both the request and response, such as the `User-Agent` field (which is specified by RFC 7231 [35]) and the `X-Powered-By` field (which is a nonstandard common usage). The request even has a field `unused-field` in the HTTP body. The fact that these fields are not specified by the OpenAPI specification does not mean that they must not be present in the actual HTTP request or response. Instead, it signals that the server does not make use of these fields when processing the request, and the client should not expect to handle these fields when parsing the response. In other words, the OpenAPI specification only describes the fields that are relevant to the server and the client, and the server and client are free to include additional fields as needed. Hence, a request and response need only to include all the required fields to be considered conformant to the OpenAPI specification. It may also include some optional fields in the specification or even additional fields not specified by the specification.

3.2.3 Formal Properties to Check for

We formally specify the properties we check for in the server implementation in Definition 3.7. These properties are the basis for the verification algorithm we develop in the rest of this thesis. In preparation for the properties, we need to first formalize the notations for the server implementation.

Definition 3.6 **Server Implementation**

A **server implementation** $\mathcal{S}' = \overline{\mathcal{E}'}$ is an implementation of the RESTful API specification $\mathcal{S} = \overline{\mathcal{E}}$. Here, the prime symbol $\mathcal{S}'$ indicates the implementation relation to the specification $\mathcal{S}$. Similarly, the prime symbol $\mathcal{E}'$ indicates that each **endpoint implementation** $\mathcal{E}' = (M, U, \mathcal{H}')$ implements the endpoint $\mathcal{E} = (\mathcal{M}, \mathcal{U}, \mathcal{H})$.

All implementations are of the same type as their corresponding specifications, except for the **handler function implementation** $\mathcal{H}' = R' \to P'$, which implements some specified handler $\mathcal{H} = \mathcal{R} \times \mathcal{P}$.

Definition 3.7 **Verification Properties**

RESTASSURED verifies RESTful API specification conformance in the server implementation.

[1] An HTTP packet, like any other network packet, is a binary data structure of some encoded information. However, it is often more human-readable, as the information is directly represented with ASCII encoding.

They are composed of a formatted header and a body, which can contain anything from a plain string to JSON objects to binary files, which are parsed according to the `Content-Type` header field. An HTTP request always begins with `METHOD PATH`, followed by some HTTP headers in the form of `KEY: VALUE`, a blank line, and the body content. Similarly, an HTTP response always begins with `HTTP-VERSION STATUS-CODE`, followed by some HTTP headers in the form of `KEY: VALUE`, a blank line, and the body content. For more details, see RFC 7231 [35].

```http
POST /login HTTP/1.1
Host: localhost:3000
User-Agent: curl/8.7.1
accept: application/json
Content-Type: application/json
Content-Length: 116

{
    "username": "ys5",
    "password": "my-secret-password",
    "unused-field": "this field is not
    ↪ specified or used"
}
```

```yaml
/login:
  post:
    summary: Login to the application.
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              username:
                type: string
              password:
                type: string
    /* ... */
```

Listing 3.1: An example of a RESTful HTTP request to the login endpoint defined in Listing 1.1. The HTTP packet is shown in plaintext format on the left, and the relevant fragment of the OpenAPI specification is shown on the right. The corresponding fields in both the HTTP packet and the specification are highlighted to show how the specification describes the actual request.

```http
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=
    ↪ utf-8
Content-Length: 47
ETag: W/"2f-+U9igXk4nkx4svjBD957HNUsGz0"
Date: Fri, 05 Apr 2024 02:22:32 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"message":"Login successful",
    ↪ "username":"ys5"}
```

```yaml
/* ... */
responses:
  200:
  description: Login successful.
  content:
    application/json:
    schema:
      type: object
      properties:
      message:
        type: string
        description: A success message.
      username:
        type: string
        description: Username.
  /* ... */
```

Listing 3.2: An example of a RESTful HTTP response from the login endpoint defined in Listing 1.1. The HTTP packet is shown in plaintext format on the left, and the relevant fragment of the OpenAPI specification is shown on the right. The corresponding fields in both the HTTP packet and the specification are highlighted to show how the specification describes the actual response.

More specifically, it guarantees that the following properties are satisfied given a RESTful API specification $\mathcal{S} = \overline{\mathcal{E}}$ and a **server implementation** $\mathcal{S}' = \overline{\mathcal{E}'}$.

1. **Soundness of API Endpoints in Specification**: For each endpoint defined in the OpenAPI specification (which can be uniquely identified by the HTTP method and URI), there must exist some corresponding endpoint implementation with the same HTTP method and URI in the server code.

$$\forall \mathcal{E} = (M, U, \mathcal{H}) \in \mathcal{S} \mid \exists \mathcal{E}' = (M, U, \mathcal{H}') \in \mathcal{S}'$$

2. **Well-definedness of Handler Functions**: For each endpoint $\mathcal{E} = (M, U, \mathcal{H})$ defined in the OpenAPI specification, the corresponding handler function implementation $\mathcal{H}' = R \to \mathcal{P}(P)$ must implement the specified handler $\mathcal{H} = R \times P$. That is, the handler function must be well-defined for all possible request schemas R defined in the OpenAPI specification.

   In other words, for all possible request schemas R received, the handler function must not throw an error or crash and must produce some set of response schemas P'. In addition, for each possible branch of execution in the handler function implementation, the response schema produced $p' \in P'$ must be a valid response schema P defined in the OpenAPI specification.

$$\forall \mathcal{H} = R \times P \text{ and } \mathcal{H}' = R' \to P' \mid \forall r \in R, \mathcal{H}'(r) \subseteq P$$

A violation of these properties indicates a non-conformance of the server implementation to the OpenAPI specification.

Moreover, we also check for the following properties, which do not signal a violation of the conformance property. However, they may be indicators of a poorly written specification or a poorly implemented server, and give warnings to the user if they are found.

3. **Completeness of API Endpoints in Specification**: For each endpoint implemented in the server code, there must exist some corresponding endpoint in the OpenAPI specification. Otherwise, the programmer may have forgotten to write specification for the endpoint, or is hiding the endpoint from specification for "security through obscurity", a bad practice.

$$\forall \mathcal{E}' = (M, U, \mathcal{H}') \in \mathcal{S}' \mid \exists \mathcal{E} = (M, U, \mathcal{H}) \in \mathcal{S}$$

4. **Completeness of Response Schemas in Specification**: For each possible fields in an actual response P' produced by the server, it should be defined in the OpenAPI specification response schema P. Otherwise, the server may return extra fields, which may leak extra information; or the server may expect the client to handle some fields that are not present in the specification, which is a bad practice.

$$\forall \mathcal{H} = R \times P \text{ and } \mathcal{H}' = R' \to P' \mid \forall r \in R, \mathcal{H}'(r) \supseteq P$$

3.3 Our Target Language and Library

Since we assume that we have access to the source code, we plan to conduct symbolic execution on the source code itself, rather than conducting a binary analysis with the compiled binary (using tools like KLEE [11]) This technique is called whitebox analysis, which, like most program analysis techniques, is language specific. Therefore, we need to choose a specific language to target.

In this thesis, we focus on the TypeScript language, a statically typed superset of JavaScript that compiles to plain JavaScript and is widely used in web development. Moreover, we are specifically targeting the popular Express.JS library, a minimal and flexible Node.JS web application framework that allow programmers to build web applications on [30].

However, it should be noted that RESTful API servers can be written in any language and with any library. The OpenAPI specification is designed in a way that is language-agnostic, and the server implementation can be written in any language that supports HTTP. Some popular choices for backend web service development other than TypeScript/JavaScript with Express.js include Python with Flask, Ruby with Ruby on Rails, PHP with Laravel, Java with Spring, and Golang with Gin. In more recent years, serverless architectures have also gained popularity in RESTful API services, where an API gateway route requests to serverless functions that are executed on demand. Some function-as-a-service platforms include AWS Lambda, Google Cloud Functions, and Azure Functions. They allow developers to write serverless functions (handlers) in a variety of languages. However, we will not consider these platforms in this thesis, as they differ from traditional web server environments and are more heterogeneous in nature, thus requiring a different approach to analyze.

We have chosen TypeScript/JavaScript as our target language(s), primarily because of its popularity in web development and in general. As of Apr 2024, the TIOBE index [103] ranks JavaScript as the 6th most popular programming language, behind Python, C, C++, Java, and C#. Moreover, the PYPL PopularitY of Programming Language Index, which is based on the frequency of searches for tutorials on Google [13], ranks JavaScript as the 3rd most popular programming language, behind only Python and Java. Therefore, we believe that a beginner in web application development is more likely to choose JavaScript as their language of choice to develop a RESTful web service. They are also more prone to make mistakes in their code, and thus more likely to benefit from a tool that can automatically check for conformance to the written OpenAPI specification. Therefore, we believe that targeting JavaScript is a good choice for RestAssured. In addition, TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript, and it makes it easier to reason about the code and to perform static analysis. In other words, if RestAssured supports (non-strict) TypeScript, it by extension also supports JavaScript[2].

Similarly, we have chosen Express.JS [30] to be the target library for its popularity. According to NPM, the official package manager for Node.js, Express.JS is among the most downloaded packages, with over 20 million average weekly downloads and 24 million weekly downloads at the time of

[2]TypeScript has a strict mode, where it enforces stricter type checking rules than JavaScript, e.g. it disallows the type blackhole `any`.

TypeScript in strict mode is not a superset of JavaScript, and the TypeScript compiler will reject some JavaScript code that is otherwise valid. Here, when we talk about TypeScript, we refer to TypeScript in non-strict mode, where it is a strict superset of JavaScript. Therefore, if RestAssured supports (non-strict) TypeScript, it will also support JavaScript.

writing [31]. Moreover, there are 78 thousand open source packages published on NPM that depend on Express.JS, which indicates that it is widely used in the Node.js ecosystem [30]. In comparison, React, the most popular frontend library for building user interfaces, also has around 20 million average weekly downloads and 138 thousand dependent packages [32]. It should note that weekly download counts and dependent counts are only rough indicators of popularity, and there exist many confounding variables such as CI/CD adoption rate (which automatically downloads dependencies to test them) and the culture of the community (backend software projects are less likely to be open source than frontend software projects). However, we think these are strong indicators of the popularity of Express.JS in the Node.js ecosystem and justify our choices in targeting it for our analysis.

3.3.1 RestScript: A Restricted Subset of JavaScript

Both JavaScript and TypeScript are large languages with complicated and quirky semantics. Due to the limited timespan of this thesis, we need to prioritize the most important semantics first. Thus, we shall restrict our language to a meaningful subset of JavaScript/TypeScript. We name this language RestScript.

We build RestScript upon a past work in PLDI '15 [76], which derived formal syntax and semantics for a near-complete subset of ECMAScript 5.1 (also known as ES5) [28], the standard for JavaScript proposed in 2011. However, it is worth noting that the modern day JavaScript has evolved significantly since the release of the ES5 standard. Some of the most prominent changes include the `let` and `const` keywords for variable declarations, the concept of modules, import/export statements, and asynchronous programming with promises introduced in the ECMAScript 6 standard (ES6, also known as ECMAScript 2015, or ES2015); the `async` and `await` keywords for easier asynchronous programming introduced in ES8/ES2017; `Promise.prototype.finally` in ES9/ES2018; the nullish coalescing operator `??` and the optional chaining operator `?.` (which were both originally proposed and widely used in TypeScript) and `BigInt` type for arbitrary precision integers in ES11/ES2020. We will selectively support some of these features in our restricted JavaScript language, for their wide usage and importance in modern web development.

At a high level, RestScript supports the most important features of the JavaScript/TypeScript language, such as variable declarations, function declarations, expressions, control flow statements (if/else, for/while loops, and function calls), and literals. We specifically support some of the features used widely in web service programming, such as arrow functions, object literals, array literals, and object property/array element access expressions. For simplicity, we assume that all variables are declared before use, and we do not support hoisting, a JavaScript feature that allows variables to be used before they are declared. Moreover, we do not fully support the dynamic re-binding of `this`'s scope in JavaScript.

There are also a few things we do not support (for now) in our restricted JavaScript language. They include:

- Exception throwing and catching, since they are very engineering-heavy. However, they are important in web services, since uncaught exceptions will automatically result in HTTP 500

responses. They are in principle possible to support, with Java-like annotations on possible exceptions.

- `import`/`export` and ES modules (we only support `require` for now), for its engineering complexity. However, they are widely used in modern web services code. They are in principle possible to support with a slightly more complicated runtime environment.
- Asynchronous code (promises, async/await), since they are not widely used in the beginner's code and are a lot more difficult to reason about.
- Node.js and browser standard libraries, such as `console`, `fs`, `sys`, `fetch`, and `XMLHttpRequest`; since they are complicated and often have side effects.
- Dynamic code execution like `eval`, since it is rarely used in practice.
- `class`es, since they are not widely used in JavaScript.

Formal Syntax of the Restricted JavaScript/TypeScript Language

For brevity, we will only present some of the most important and interesting syntactic features of the language. The full formal syntax for the restricted JavaScript language is presented in Appendix B.

```
SourceFile ::= Statements
Statements ::= Statement
             | Statement ";" Statements
             | /* ... */

Statement ::= Block
            | VariableStatement
            | FunctionDeclaration
            | ExpressionsStatement
            | IfStatement
            | WhileStatement
            | ForStatement
            | ReturnStatement
            | /* ... */

Block ::= "{" Statements "}"

VariableStatement ::= "var" VariableDeclarations
                    | "let" VariableDeclarations
                    | "const" VariableDeclarations
VariableDeclarations ::= VariableDeclaration
                       | VariableDeclaration "," VariableDeclarations
VariableDeclaration  ::= Name
                       | Name "=" Expression
                       | /* ... */

FunctionDeclaration ::= "function" Identifier "(" ParameterList ")" Block
Parameters ::= "(" ParameterList ")"
ParameterList ::= Identifier "," ParameterList
                | Identifier
                | /* ... */

ExpressionsStatement ::= Expression
Expression := BinaryExpression
```

```
            | UnaryExpression
            | ParenthesizedExpression
            | Literal
            | Identifier
            | ArrowFunction
            | CallExpression
            | NewExpression
            | PropertyAccessExpression
            | ArrayElementAccessExpression

/* All common expressions (arithmetic, bitwise, comparison) are omitted for brevity */

Literal ::= "null"
          | "true"
          | "false"
          | "this"
          | NumericLiteral
          | StringLiteral
          | ObjectLiteral
          | ArrayLiteral

/* All common literals (numbers, strings) are omitted for brevity */

ObjectLiteral ::= "{" PropertyNameAndValueList "}"
PropertyNameAndValueList ::= PropertyNameAndValue
                           | PropertyNameAndValue "," PropertyNameAndValueList
PropertyNameAndValue ::= Identifier ":" Expression

ArrayLiteral ::= "[" ElementList "]"
ElementList ::= Expression
              | Expression "," ElementList

Identifier ::= regex("[a-zA-Z_][a-zA-Z0-9_]*")  // cannot start with a number

ArrowFunction ::= Parameters "=>" Expression

CallExpression ::= Expression "(" ArgumentList ")"
ArgumentList ::= Expression[","]

PropertyAccessExpression ::= Expression "." Identifier

ArrayElementAccessExpression ::= Expression "[" Expression "]"

IfStatement ::= "if" "(" Expression ")" Block
              | "if" "(" Expression ")" Block "else" Block

WhileStatement ::= "while" "(" Expression ")" Block

ForStatement ::= "for" "(" Expression ";" Expression ";" Expression ")" Block
              | "for" "(" VariableDeclaration ";" Expression ";" Expression ")" Block
              | "for" "(" Expression "in" Expression ")" Block
              | "for" "(" VariableDeclaration "in" Expression ")" Block

/* ... */
```

Listing 3.3: A formal definition of the RestScript language using BNF. This is a simplified version of the full formal syntax, which is presented in Appendix B.

3.3.2 Targeting A Subset of the Express.JS Library

Express.JS [30] is a popular web application framework for Node.js that we target in this work. Because it is an external library and extremely complicated (currently 538k line of JavaScript code according to GitHub API of the repository [30]), the analysis of Express.JS server code is non-trivial. Instead, like other symbolic execution-based testing tools, we build an internal model of the server code to reason about the server's behavior. Since our tool primarily targets developers of RESTful API services instead of the maintainers of Express.JS, we do not aim to model the entire Express.JS library but only a meaningful subset of its features that are commonly used in practice. In Listing 3.4, we show an example Express.JS server that demonstrates some of the features that we model.

The Syntax of Express.JS

As shown in Listing 3.4, the programmer is to first require the Express.JS library by calling the `require()` function, which returns the module object that contains the Express.JS library. Then, an Express.JS server is created by calling the default module export `express()` function, which returns an instance of the Express application `app`. The server is configured by calling methods on the `app` object, which modifies the internal state of the server. Some common methods include:

- `app.use(handler, …)`: defines a middleware (a series of handler callback functions) that will be executed for every request, regardless of the path or HTTP method.
- `app.set(key, value)`: defines a setting that affects the behavior of the server, such as the environment mode ("development" or "production") or routing behaviors.
- `app.get(key)`: retrieves the value of a setting that was previously set with `app.set()`. Note that there exists two `get` method with entirely different meanings.
- `app.get(path, handler, …)`: defines a route that will invoke the series of handler callback functions when a GET request to the given path is received.
- `app.post(path, handler, …)`: defines a route that invoke the series of handler callback functions when a POST request to the given path is received.
- `app.all(path, handler, …)`: defines a route that invoke the series of handler callback functions when a request to the given path is received, regardless of the HTTP method.
- `app.route(path)`: creates a route that can be used to define handlers for a given HTTP method. For example, `app.route("/hello").get(handler1).post(handler2)`.
- `app.listen(port, callback, …)`: starts the server listening for incoming requests on the given port, and calls the callback functions when the server is ready.

When the handler functions are called, they are passed three arguments in order:

- `req`: the request object, which contains information about the incoming request, such as the request path, query parameters, headers, and body.
- `res`: the response object, which contains methods to send the response back to the client, such as `res.send()` and `res.status()`.
- `next`: a function that, when called, passes control to the next handler in the current path, or the next matching route.

```
const express = require("express");

const app = express();

// a built-in middleware that is called for every request
// parses incoming requests with JSON payloads into a JS object req.body
app.use(express.json());

// a user-defined middleware that is called for every request
app.use((req, res, next) => {
  console.log("LOGGED");
  next(); // pass control to the next handler
});

app.get("/hello/:name", (req, res, next) => {
  res.send(`Hello, Param ${req.params.name}!`);
  // next();  // if this line is uncommented, the /hello handler will
  // also be executed and result in error ERR_HTTP_HEADERS_SENT
});

app.get("/hello", (req, res) => {
  res.send("Hello, World!");
});

// A regex route that matches /hello followed by any string and /[name]
app.post(/^\/hello(.*)\/(.*)/, (req, res) => {
  res.send(`Hello, RegEx ${req.params[1]}!`);
});

if (process.env.NODE_ENV === "development") {
  // default fallback route that returns 500 for all other paths in development
  app.use((req, res) => {
    res.status(500).send("Problem!");
  });
} else {
  // another way to write a default fallback route that returns 404 for all
  app.all("*", function (req, res) {
    res.status(404).send("Page not found");
  });
}

app.listen(3000, () => {
  console.log("Server started on port 3000");
});
```

Listing 3.4: An example Express.JS server program that uses various features in Express.JS: built-in and user-defined middlewares, routes with path parameters and regular expressions, default fallback routes, and routes that are defined based on startup conditions (in this case, an environment variable).

The Behavior of Express.JS Servers

In a typical Express.JS application, the server code is organized into a set of routes (equivalent to RESTful API endpoints), each of which is a combination of a path and an HTTP method. For each route, Express.JS accepts one or multiple callback handler functions that are called when the route is matched. Moreover, the route path in Express.JS can contain regular expressions, which makes it possible for one route to match multiple different paths, or for a default fallback route to be defined when no other route matches. There also exist catch-all routes (known as "middlewares" in Express.JS terms) that are always executed regardless of the path or HTTP method. These middlewares can be used for tasks such as logging, authentication, or error handling. For example, the code in Listing 3.4 shows a simple Express.JS server that makes use of all these features.

When a request is received by an Express.JS server, the server will go through all the routes in the order they are defined in the code, and skip to the next route if the current route does not match the request's path or method. If a route matches the request, the server will execute the handler functions associated with that route. If a handler function does not call the `next()` function, (like L17 in Listing 3.4), the server will stop executing the handlers and send the response back to the client. On the other hand, when a handler calls `next()` (like L12 in Listing 3.4), the server will continue to the next handler in the current path, or the handler for the next matching route.

Chapter 4

Algorithms

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

— Donald Knuth, in "Literate Programming" [51, p. 97]

4.1 Overview

RestAssured's approach to verifing the server-side RESTful API code for conformance can be roughly divided into two parts. It first analyzes the server-side code to build an internal model of the API service, its endpoints and their handlers. Sometimes, the server may have multiple states, depending on some environment variable or startup conditions (e.g. connection to a database, or a different mode of operation), thus resulting in different endpoints or handlers. Hence, RestAssured will execute the server code across all possible execution paths and possibly end up with multiple models of the server (section 4.3). Then, for each of the endpoints defined in each model of server, RestAssured generates symbolic requests based on the OpenAPI specification (query parameters, HTTP headers, and body contents), and symbolically executes the handlers to derive a set of possible symbolic responses, walking through all possible execution paths. For each possible symbolic response, it uses a constraint solver to check for specification violations. If any are found, it generates concrete input values corresponding to counterexamples that help programmers replicate and debug non-conformant responses (section 4.4). A high-level summary of RestAssured's core algorithm can be seen in Figure 4.1.

We begin by describing our symbolic execution engine that handles RestScript, a restricted subset of TypeScript/JavaScript we defined in subsection 3.3.1, and the Express.JS library. We first detail the symbolic execution techniques employed in building a symbolic execution engine for the restricted language from first principles and discuss the many design choices we made throughout the process (section 4.2).

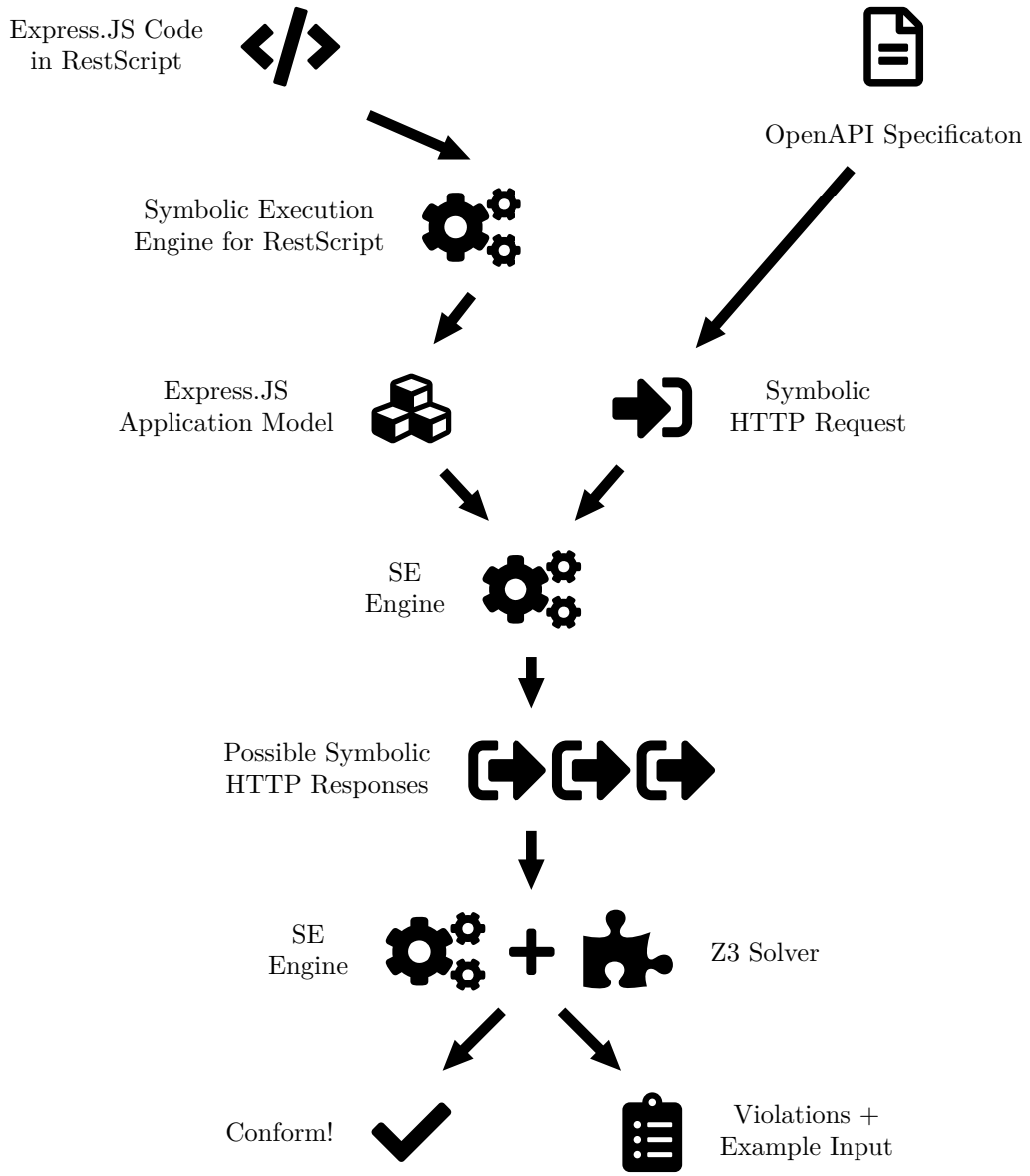


Figure 4.1: The high-level summarization of RestAssured's workflow. RestAssured verifies the conformance of a server-side RESTful API implementation to a OpenAPI specification.

The tool first takes in the source code of the Express.JS application to build an internal model of the application, its routing logic and handler functions. Then, it uses the OpenAPI specification to generate symbolic requests for each of the defined endpoints. These symbolic requests are then sent to the symbolic execution engine, which simulates the routing logic of the Express.JS application, and executes the handler functions of the defined routes. The engine produces a set of possible symbolic responses for each of the requests. These responses are then checked using the SE engine and Z3 against the specification for violations. If violation is found, RestAssured synthesizes a counterexample and reports it to the user.

4.2 Symbolic Execution for RestScript

Due to the lack of off-the-shelf static symbolic execution engines that can handle JavaScript/TypeScript or code, we built our own symbolic execution engine for a restricted subset of the language, which we define in subsection 3.3.1 and call RestScript.

An Example Program

Let us begin by drawing an example program that we will use to illustrate the symbolic execution engine. The program is intentionally complicated, using conditionals, anonymous inline arrow function declarations, variable shadowing, and function evaluations to demonstrate some features of the symbolic execution engine.

```
1  function add(a, b, c) {
2      if (a < 0) {
3          return 0;
4      } else if (b < 0) {
5          return 0;
6      }
7
8      let adder = (x, y) => {
9          return x + y;
10     };
11
12     if (c == 0) {
13         let adder = (x, y) => {
14             return x - y;
15         };
16     } else {
17         adder = (x, y) => {
18             return x * y;
19         };
20     }
21
22     const val = adder(a, b);
23     return val;
24 }
```

Listing 4.1: An example RestScript program to illustrate the symbolic execution engine. This program is intentionally overcomplicated to demonstrate some features of the RestScript symbolic execution engine. The function takes in three numeric parameters `a`, `b`, and `c`, which the engine models as symbolic values that can take on any integer value. At the end of execution, the engine will produce four different states, each representing a different path of execution.

The function takes in three numeric parameters `a`, `b`, and `c`, which may be any numerical values. The engine does not know the concrete values of these parameters, and hence models them as **symbolic values**, which can take on any integer value. Then, when the function is executed, the engine will go on to execute all statements and evaluate all expressions in the function body, recording their side effects and updating the **symbolic states** accordingly, which represent the internal state of the program running at each instance.

For example, the variable `adder` is declared at L8-L10 and assigned an arrow function that adds two numbers. A slightly more complicated example happens at L2, where an if-else statement is

used to return 0 if `a` is less than 0. Since the engine does not know the concrete value of `a`, it will consider all possible cases, and create two different states: one where `a` is less than 0 and one where `a` is greater than or equal to 0. For the first state, the engine will add $\texttt{a} < 0$ to the **path constraint**, enter the block in the true branch, which returns 0, and ends the execution of the function. Here, path constraint is a logical formula that represents the conditions that must be satisfied for the current path to be taken. Hence, the path constraint tells us what values drive the program down into the path. The engine similarly creates another state for the false branch, adding $\neg(\texttt{a} < 0)$ to the path constraint, and continues on to execute the next statement, which is the else-if statement at L4.

The symbolic execution engine is aware of the scope of the variables and need to handle variable shadowing, *i.e.* the re-declaration of a variable with the same name in an inner scope, correctly. For example, the variable `adder` is declared twice, once in the outer function scope (L8-L10) and once in the inner block scope (L13-L15). When the variable is declared in the inner block scope, it shadows the variable declared in the outer function scope. As a result, the value assigned to the variable `adder` in the inner block scope should not affect the value of the variable `adder` in the outer function scope. However, at L17-L19, the variable `adder` is reassigned. This `adder` variable refers to the same variable declared in the outer function scope (L8-L10), and the engine should update the value of the variable `adder` in the outer function scope to the new value.

At the end of execution, the engine will produce four different states, each representing a different path of execution:

1. The path where `a` is less than 0, and the function returns 0.

2. The path where `a` is greater than or equal to 0, `b` is less than 0, and the function returns 0.

3. The path where `a` is greater than or equal to 0, `b` is greater than or equal to 0, `c` is 0. The function returns the sum of `a` and `b`, the result of evaluating the arrow function at L8-L10.

4. The path where `a` is greater than or equal to 0, `b` is greater than or equal to 0, `c` is not 0. The function returns the product of `a` and `b`, the result of evaluating arrow function at L17-L19.

In the following subsections, we will discuss the inner workings of this symbolic execution engine, including how it models the symbolic environment, evaluates expressions, handles branching and multiple paths in the program, and simulates control flow indirections in the program. It is important to note that the symbolic execution engine is not faithful to the JavaScript language specification [29] but makes reasonable approximations given real-world usage patterns to make the symbolic execution more tractable. This is because the JavaScript language is extremely complex, and faithfully modeling the entire language would be time-consuming, given the time constraints of this project. These simplifications are also detailed in the following subsections. They are also potential sources of unsoundness in the symbolic execution engine, which we acknowledge and discuss the rationale behind them in section 7.2.

4.2.1 Symbolic Environment and Activation Records

In a symbolic execution engine, all states of a program execution is stored in a symbolic environment, or sometimes known as symbolic states. In our symbolic execution engine, we include the following information in the symbolic state:

- The current path constraint, which is a logical formula that represents the conditions that must be satisfied for the current path to be taken.
- The symbolic store, which is a mapping from all variables (primitives, objects, arrays) to their respective values (symbolic or concrete), and from named functions to their declarations.
- The stack of activation records, which represents the current execution context of the program, including the next statement to be executed, the current function being executed, the statement to return to, and some of the scope information for enclosures.
- Internal properties of the symbolic execution engine, such as bookkeeping information for the control flow, cached results, and sibling states (for multipath execution).

The Path Constraints

A path constraint is a logical formula that represents the conditions that must be satisfied for the current path to be taken. It is a conjunction of many logical clauses, with each clause representing a single condition.

For example, for the execution of Listing 4.1 to reach L13, it must take the false branch at L2, the false branch at L4, and the true branch at L12. Hence, the path constraint for the current state would be the conjunction of these three clauses:

- $\neg(\texttt{a} < 0)$, since the program takes the false branch at L2.
- $\neg(\texttt{b} < 0)$, since the program takes the false branch at L4.
- $\texttt{c} = 0$, since the program takes the true branch at L12.

Therefore, the conjuncted logical formula would be:

$$\neg(\texttt{a} < 0) \wedge \neg(\texttt{b} < 0) \wedge (\texttt{c} = 0)$$

The path constraint tells us what values drive the program down into the path. Since the path constraint can be expressed as a logical formula, we can use a Satisfiability Modulo Theories (SMT) solver, such as Z3 [24], to solve it. If we were to solve for the path constraint, we would get the set of values for `a`, `b`, and `c` that would lead the program to this path. In this case, the set of values would be $\texttt{a} \geq 0 \wedge \texttt{b} \geq 0 \wedge \texttt{c} = 0$. Since Z3 tells us that the path constraint is satisfiable, we know that there exists some value for `a`, `b`, and `c` that would lead the program down this path. In other words, this path is reachable. This is an important property, because some paths may be unreachable in real-world codebases, and we want to avoid spending time analyzing them and the outputs they produce.

In our design, we adopt a *lazy evaluation* strategy for the path constraints, meaning that we merely store the path constraints during execution, and only check for satisfiability after the

execution finishes. This is opposed to the *eager evaluation* strategy, where the path constraints are checked for satisfiability at every branching point, so unreachable paths can be pruned early. Although the lazy strategy may lead to more paths being explored, it reduces the number of calls to the SMT solver and is reported to be "more efficient than their eager counterparts" [4].

The Symbolic Store

The symbolic store is a symbol table that contains all the variables and identifiers in the program and their respective values (symbolic or concrete). In our symbolic execution engine, we use a map to represent the symbolic store, where the keys are the variable names and the values are special `Value` objects for the symbolic execution. A `Value` is represented as a tuple of its type and an expression. For concrete values that are produced by literals, the expression is the concrete value itself, such as a number, a string, or a boolean in JavaScript. For symbolic values, the expression is an SMT-LIB expression [6] that represents the value. For objects, the expression would be another map of all fields and their respective values.

Like most other programming language, scope is important in JavaScript. Moreover, the scoping rules of JavaScript are perhaps among the most complicated, with many quirks and exceptions. It is built on the concept of lexical scoping, where the scope of a variable is determined by its position in the source code. To quote the language standard, "usually an Environment Record is associated with some specific syntactic structure of ECMAScript code such as a `FunctionDeclaration`, a `BlockStatement`, or a Catch clause of a `TryStatement`. Each time such code is evaluated, a new Environment Record is created to record the identifier bindings that are created by that code." [29, §9.1]. However, not all identifiers are associated to the immediate environment record. For example, variables declared by a `var` statement are usually scoped at the function level and hoisted to the creation of the environment record [29, §14.3.2], whereas variables declared by `let` and `const` are scoped at the block level and only created during the execution of that statement [29, §14.3.1]. To make things more convoluted, the keyword `this` is scoped at the non-arrow-function function level, and can also be re-binded to other environment records at runtime [29, §9.1.1.3].

In our symbolic store, we simplify the scoping rules to dynamic scoping, where the scope of a variable is determined by the current execution context at runtime. The dynamic scoping rules are semantically similar to the lexical scoping rules if no variable shadowing or capturing is present, and is easier to implement. In our symbolic execution engine, we use a single stack of **execution contexts** (the rough equivalent of environment records under dynamic scoping) to represent the scope of the variables. Each execution context represents a level of scoping, and contains a map of the variables and their values. The TypeScript definition of the execution context is shown in Listing 4.2, where we use an array to simulate the stack.

At the start of the program, the stack of execution contexts contains only the base execution context, which contains the built-in functions and variables in JavaScript, such as `console` and `require`. When a new scope (e.g. a block or a function) is entered, a new execution context is pushed onto the stack. When a scope is exited, the execution context is popped from the stack. The type of scope is determined by the prefix of the scope name, which is a string that is used

```
type ExecutionContext = {
    symbolTable: Map<string, Value>;
    scopeName: string; // the prefix determines the type of scope (function, block, etc.)
};
type ExecutionStack = ExecutionContext[];
```

Listing 4.2: A simplified TypeScript definition of the execution context. The execution context contains a map of the variables and their values, and a scope name that determines the type of scope (function, block, etc.). A function-typed scope contains special symbols and also serves as an activation record.

to differentiate the different types of scopes (e.g. function, block, etc.). At the function scope, in addition to the variables declared in the function, the symbol table also stores special symbols to denote special control flow directives and `this`. When looking up a variable in the symbolic store, we start from the top of the stack (the current execution context) and search downwards. This way, we implement the dynamic scoping rules, where the current execution context takes precedence over the outer execution contexts. When declaring variables, we add the variable to the nearest appropriate execution context in the stack. For example, `let` and `const` are block-scoped, `var` is function-scoped. For `var`, we add the variable to the function-level execution context when the statement is encountered and choose not to hoist its declaration, for modern JavaScript/TypeScript code usually does not use `var` anymore, not to mention that relying on the hoisted declaration of `var` is a bad practice (for only the declaration and not the initialization is hoisted, and the variable will be left declared but uninitialized). Same goes for function declarations. For simplicity, we do not support the dynamic scoping of `this`. For `this`, we bind the `this` keyword to the current non-arrow function's `this` value, and we do not handle the dynamic binding of `this` at runtime.

Activation Records

Lastly, the **activation record** is a stack of records that represents the current execution context of the program [29, §9.4]. It should contain "whatever implementation specific state is necessary to track the execution progress of its associated code" [29, §9.4].

In our design, due to the dynamic scoping rules, we can represent both the activation record and the environment record with the same data structure, which we call the execution context and detailed above. Hence, an activation record becomes a special execution context with "function"-typed scope name. It contains special symbols for the current function being executed, the next statement to be executed, the statement to return to, and offers easier access to the current function scope. The activation record helps us to keep track of changes in the control flow, such as a function call, a function return, or a break or continue statement in a loop.

Whenever a function is called, a new activation record is pushed onto the execution stack. The new activation record contains the function to be executed, the next statement to be executed (usually the first statement of the function), and the statement to return to (usually the statement that evaluates the function call expression). When we encounter a return statement in the function block, we would skip the execution of the rest of statements, save the value to be returned as a special symbol in the symbolic store, and pop the activation record from the stack. Then, we would

continue the execution from the statement to return to and replace the function call expression with the return value.

4.2.2 Expression Evaluation and Symbolic Values

RestScript nodes can be separated into two categories: statements and expressions. **Statements** are executed to perform certain actions with side effects, such as variable declarations, function declarations, or change the control flow in the cases of if-else statements and loops. On the other hand, **Expressions**—such as an arithmetic expression, a function call, or a variable reference—are evaluated to produce some values. Naturally, during a statement execution, we update the symbolic store and the path constraint to reflect the changes in the program state. A statement may contain expressions, which we evaluate to produce a `Value` object. A **`Value` object** is a tuple of its type and an expression, and is used by the symbolic execution engine to represent both concrete and symbolic values.

For simple expressions like arithmetic expressions on concrete values, we evaluate them using the built-in JavaScript arithmetic operators. For example, consider the following expression in the RestScript Abstract Syntax Tree (AST), which is a subset of the TypeScript AST:

```
BinaryExpression(NumericLiteral(1), PlusToken, NumericLiteral(2))
```

It can be evaluated concretely using the JavaScript code `(1 + 2)` to produce the concrete value

```
Value(Number, 3).
```

This value is then returned as the result of the expression evaluation.

However, if any operand is a symbolic value, the resulting value need to also be symbolic. We evaluate them by constructing an SMT-LIB expression that represents the operation and the operands. These SMT expressions will eventually be solved by Z3 [24], our SMT solver, to search all possible values that satisfy the path constraint. For example, let us change the numeric literal `1` in the previous symbolic expression to be a symbolic value with expression $3 \cdot x$. Recall that symbolic expressions are represented as SMT-LIB expressions. Hence, `(* 3 x)` represents the product of 3 and a symbol `x`, which can take on any integer value, assuming the path constraint already contains the symbol declaration `(declare-const x Int)`. Then, the expression becomes:

```
BinaryExpression(SymbolicValue<Number>("(* 3 x)"), PlusToken, NumericLiteral(2))
```

This expression would instead be evaluated to the following symbolic value:

```
SymbolicValue<Number>("(+ (* 3 x) 2)")
```

Here, `SymbolicValue<T>(expr)` is a syntactic sugar that expands to:

```
Value(Symbol, { type: T, expr: expr }).
```

Sometimes, expression evaluation may involve branching. For example, if an expression divides by a symbolic value, the symbolic execution engine has to consider both the case where the divisor is zero and the case where it is not zero. Similarly, if the expression is a `CallExpression` that calls a function with multiple possible return values, the symbolic execution engine has to consider all possible return values. In these cases, the symbolic execution engine clones the current state with different path constraints and evaluates the expression separately in each state.

Modeling Object Typed Values

Other than primitive values, we also handle values of the Object types, such as objects and arrays in JavaScript, and function values. For object values, we store each field and its value as a key-value pair in the symbol. For example, the object `{a: 1, b: 2}` would be stored as the following:

```
Value(Object, {a: Value(Number, 1), b: Value(Number, 2)})
```

Concrete array values are modeled similarly to object values, except we map each index as a key to its value. For example, the array `['a', 'b']` would be stored as the following:

```
Value(Array, {0: Value(String, 'a'), 1: Value(String, 'b')})
```

Symbolic array values are very complicated to fully model, because we need to handle some symbolic array length and have symbolic values for each of the elements, not knowing the exact number of elements in the array. In our symbolic execution engine, we simplify them to a list of the same values, with a symbolic length. As a result, we can model an array of numbers as:

```
SymbolicValue<Array>({value: SymbolicValue("x"), length: SymbolicValue("n")})
```

with the following path constraints:

```
(declare-const x Real), (declare-const n Int), (assert (>= n 0))
```

where `x` is a symbolic value representing the value of the array elements, which is assumed to be the same across all elements, and `n` is a symbolic value representing the length of the array. We are able to make this simplification because the REST specification is not granular enough to make specific assumptions about the contents of each array element. Instead, it only requires the array to be a list of values of some refined types.

Lastly, function values are stored as a reference to the function body in the AST. When a function is called, the engine creates a new activation record for the function, and goes to the first statement of the function body.

4.2.3 Multipath Execution

Another problem we encounter in the example Listing 4.1 is the presence of branching, which creates multiple paths in the program execution, depending on the boolean value that some conditions evaluate to. In the example, the program has three branching points: the if-else statement at L2, the if-else statement at L4, and the if-else statement at L12. If the input values of `a`, `b`, and `c` are

concrete, we can determine which path the program will take. For example, if $\texttt{a} = -1$, $\texttt{b} = 1$, and $\texttt{c} = 0$, the program will take the first path, and directly return 0.

However, since we do not know the concrete values of the inputs, we model them as symbolic values, which can take on any numerical values. As a result, we do not know what the conditions $\texttt{a} < 0$, $\texttt{b} < 0$, and $\texttt{c} = 0$ evaluate to, and hence do not know which path the program will take. As a result, we have to consider all possible paths and execute them separately. When we enter a branching statement, we effectively branch out into states, one for each possible branch, which we call **sibling states**. The conditions that will hold true for each state are added to the path constraint of the state. Then, we continue on to execute the next statement in each state. This technique is called **Multipath Execution** in the symbolic execution literature [15].

In general, branching could happen during the execution of an if-else statement, a switch-case statement, or a loop. RestScript does not explicitly support the switch-case statement, but it can be trivially rewritten as a series of if-else statements. Loops are usually more complicated, especially when the loop counter variable is symbolic, and is a major challenge in symbolic execution [4]. As a result, we only support bounded loops, which can be effectively rewritten as a `forEach` statement on some array.

Choices

For if-else statements, we model branching by maintaining a set of sibling states. When a branch is encountered, we first determine if the condition of the branch can be evaluated to a concrete value. If it can, we evaluate the condition and take the appropriate branch. If it cannot, we are left with a symbolic value representing the result of the condition. Depending on the truth value of the condition, the program will take one of the two branches. Hence, we clone the current state to create a sibling state for the false branch, and treat the current state as taking the true branch. If an else block exists, we mark it as the next statement to be executed for the cloned state. If there is no else block, we mark the statement after the if-else statement as the next statement to be executed for the cloned state. We then update the path constraint in both states with the condition and its logical negation, respectively. Then, we continue the execution of the current state into the true block with the new path constraint.

When the current state finishes execution in the current activation record (for example, if a return statement is encountered), we check if there are any sibling states. If there exist any, we switch to resume execution of that sibling state, until it finishes execution.

Loops and Recursion

Loops and recursion are another source of branching in a program execution, and often considered as a major challenge in symbolic execution [4]. Each iteration of a loop creates a new path in the program execution, since the program needs to evaluate the loop condition and decide whether to continue the loop or exit. The same holds for recursions. As a result, if the loop condition or the recursion termination condition is symbolic, the program will often end up with an exponential number of paths. To make it worse, if the symbolic execution engine does not have enough

information on the loop condition or the recursion termination condition, it may run into an infinite loop or an infinite recursion; consider the following example:

```
let i = 0;
while (Math.random() > 0) {
    // do something
    i++;
    console.log(i);
}
```

Since we have no information what `Math.random()` will return, the loop must consider both cases where it is greater than 0 and less than or equal to 0 at each iteration, which will result in an infinite number of paths.

Even for bounded loops that are guaranteed to terminate, the number of paths can still grow exponentially with the number of iterations and becomes infeasible to explore. For example, consider the following loop:

```
let n = ...; // a numerical symbolic input
for (let i = 0; i < n; i++) {
    // do something
    console.log(i);
}
```

In this case, we know that the program will terminate after n iterations, because the loop counter `i` is incremented by 1 at each iteration, the loop condition is `i < n`. Moreover, `n` as a `number` has a maximum value of about 1.8×10^{308} in JavaScript [29, §21.1.2.7]. Not knowing the concrete value of `n`, we have to consider all possible values that `n` can take, creating 1.8×10^{308} possible paths, which is infeasible to explore.

To simplify the handling of loops, we only support bounded, well-structured loops. If the condition can evaluate to a concrete value, the loop is unrolled and executed as a series of statements. If the condition is symbolic, the loop must be iterations over an array and can be rewritten as a `forEach` statement. We do not support recursions as of now, due to their lack of usage in real-world industrial codebases. The simplification over arrays are made possible because we model arrays in our symbolic execution engine as a list of the same values, with a symbolic length (see subsection 4.2.2). Hence, when we encounter a loop that iterates over an array, we can simply perform the loop body once on the value of array elements, and keep the symbolic length the same. This way, we are able to avoid the exponential explosion of paths that would have occurred if we had to consider all possible values for each of the array elements, while still retaining the relevant information to reason about the OpenAPI specification conformance property.

4.2.4 Other Control Flow Indirections

Other than the control flow statements listed above, there also exist other control flow indirections that do not create multiple paths, but change the control flow of the program in the current path. These include function calls, early returns, and exceptions.

When a function is called, the program jumps to the function definition and executes the function body. As a result, we store the current statement in the activation record and push a new activation

record onto the stack. Along with the activation record, we also create a new scope in the symbolic store to store the local variables of the function. The new scope should also be populated with the parameters of the function, which are passed in as arguments to the function call expression. The function is then executed as a series of statements, and the return value is stored in the symbolic store. When the function finishes execution, the activation record is popped from the stack, and the program jumps back to the caller with the return value. However, not all functions finish execution of the entire function body. Some functions may terminate early with a `return` statement, or throw an exception. In our symbolic execution engine, we also have a special symbol `__returned__` to determine if a function has returned early. If the flag is set to true, the rest of the statements in the function body are skipped without execution, and the execution returns to the caller at the end of the block.

Other than `return` statements in function calls, `break` and `continue` statements in loops can also terminate the current control flow early. We handle these cases similarly to that of `return`, by creating a special symbol to determine if the rest of the loop body is skipped. At the end of the loop body, we use the symbol to determine whether the loop should be exited or continued to the next iteration.

Unfortunately, we do not currently handle exceptions in our symbolic execution engine, but we plan to add support for them in the future. In prior works, Microsoft has added Structured Exception Handling (SEH) to C to handle exceptions, by adding special function prologue and epilogue instructions to the function runtime [53, 78]. We plan to follow a similar approach by model exception throwing and handling as special function prologue and epilogue logic. Since we are already modeling early return as skipping the rest of the function body, the addition of function prologues and epilogues should be relatively straightforward.

4.2.5 Handling of Library Codes and Dependencies

Handling library code and dependencies is also a known challenge in symbolic execution [4]. In a real-world program, it is common to interact with external environments. This includes system environments, such as the file system, the network, or the operating system; and application environments, such as external libraries and dependencies to speed up development and reuse code. However, for a symbolic execution engine, the complexity of these external environments can be overwhelming, as they are often large, complex, and oftentimes beyond the scope of the program under analysis. Moreover, interactions with system environments may produce side effects, such as I/O operations, network requests, or system calls, that are difficult to model and reason about in a symbolic execution engine. Therefore, many symbolic execution engines choose to draw a boundary between the code under analysis and the external environment. For example, CUTE [91] and KLEE [11] use a combination of concrete and symbolic execution to handle system calls and external libraries. They execute the user code symbolically, but execute the environment code with concrete values and directly use that information to reason about the user code. However, this approach cannot produce all possible outputs of the environment code, and may cause some possible paths to be missed during symbolic execution. Some other symbolic execution engines require the user

to provide some annotations or models for the environment code, or synthesize them automatically [4, pp. 16–17]. However, this approach may require a lot of effort from the user, and may not be feasible for large, complex, or proprietary libraries.

In this thesis, we choose to treat most external library code as a black box and only reason about the user code. Whenever we encounter a function call to a library function, we model it as a Z3 uninterpreted function and try to extract available information from type annotations if possible. However, this means that we are losing all information that is previously gained on the parameters, and the return value would be mapped as a completely new symbolic value that we know nothing about. In the future, we plan to give user the option to provide some annotations on how the library function maps its input to output, or to instruct the symbolic execution engine to skip some simple library functions that provide simple utilities, such as re-formatting a string.

However, the web framework library is an exception. REST servers written in Express.JS, the target web framework library in our analysis, are heavily integrated with the library. Instead of calling the library functions, all handler functions are callbacks *to be called* by the Express.JS library. Furthermore, the complexity of the Express.JS library and its extensive interactions with incoming network requests make it infeasible to analyze the library code as-is. Moreover, the motivation of RestAssured and this thesis is to help API developers to write correct RESTful API servers, not to verify the correctness of the Express.JS library. We believe it is reasonable to turn verification effort to the user code, which is more likely to contain bugs, compared to the widely-used and well-tested Express.JS library. Therefore, we assume that the Express.JS library does not contain bugs, and instead build an internal model of the Express.JS library. In this model, we capture the behavior of some Express.JS's built-in methods, how they interact with the request and response objects, and how they modify the state of the server. Since the model is internal and heavily integrated into how the RestAssured tool works, we will discuss it in detail in the following sections.

4.3 Symbolically Modeling the Express.JS Application

Now that we have built a symbolic execution engine for RestScript, our restricted subset of TypeScript/JavaScript, we can use it to analyze the server-side code of a RESTful API service. RestAssured uses symbolic analysis and takes a two-pass approach in analyzing the server-side code. It first builds an internal model of the RESTful API service, including its endpoints and their handlers, by symbolically executing the server-side code. Then, it generates symbolic requests based on the OpenAPI specification and symbolically executes the handlers to derive a set of possible symbolic responses, walking through all possible execution paths. In this section, we focus on the first pass.

In the first pass, RestAssured identifies all REST endpoints, including their HTTP methods, paths, parameters, and handler functions. In addition, it also parses through the definition for all middleware functions, which are functions that are executed regardless of paths or methods, usually before the handler functions, and can modify the request and response objects, or terminate the request-response cycle early. In other words, we build a model of the defined Express.JS application from the source code.

4.3.1 An Internal Model for Express.JS Server Application

We model the Express.JS server application as part of the symbolic environment in the symbolic execution engine. The following information is stored in the symbolic environment to represent an Express.JS application:

- `definedRoutes: Route[]`. A list of all the defined routes in the server, where each route contains the path, the method, the handler function, and the middleware functions. Each of the `Route` objects is defined as follows:
  - `catchAll: boolean`. A boolean flag indicating whether the route is a catch-all route defined by the `app.use()` method or `app.all()` method. If it is a catch-all route, it will match all paths and methods.
  - `path: string`. The URL path of the route.
  - `method: string`. The HTTP method of the route, e.g. `GET`, `POST`.
  - `handler: Function[]`. A series of handler functions of the route, which are executed when the route is matched.
- `expressSettings: Map<string, string>`. A map of all the settings defined in the server via the `app.set()` method.
- `expressRunning: boolean`. A boolean flag that indicates whether the Express.JS server is running or not. This is used to model the server startup and shutdown.
- `expressPort: number`. The port number that the Express.JS server is running on.

One caveat of this design is that we assume there to be only one Express.JS application per project. This is a reasonable assumption, for while one can create multiple Express.JS applications in one project and have them listen on different ports, it is not a common practice. Furthermore, supporting multiple instances of Express.JS applications may complicate the design as the two applications may share resources with each other and create side effects and interdependencies. To enforce this assumption, we add a boolean flag `expressApplicationCreated` to the symbolic environment, which is initialized to `false`. When the Express.JS application is created, the flag is set to `true`. If the flag is already `true`, the symbolic execution engine raises an error. In other words, our analysis engine will raise an error if the single Express.JS application assumption is violated.

4.3.2 Analyze Source Code for Express.JS Application Definitions

To parse the AST for Express.JS application definitions, we have enhanced our symbolic execution engine with a set of special symbols to represent the Express.JS library functions and objects. These special symbols are used to recognize the Express.JS library functions and objects during symbolic execution of the RestScript code. When they are encountered, the symbolic execution engine will intercept the execution and update the internal model of the Express.JS application in the symbolic environment. Then, some special symbol representing the return value will be returned instead, to continue model the execution and invocation of the Express.JS library functions.

To build the internal model, we will symbolically execute the source code itself, which is essentially defines a set of callback functions for the Express.JS application. The algorithm builds upon the symbolic execution engine, and augments it with special symbols that are created and handled as follows:

1. When the code imports the Express.JS library via the `require("express")` function call expression, the symbolic execution engine will recognize the expression and assign a special symbol `expressjs_namespace` to the return value of the function. This value is usually stored in a constant variable, such as `const express = require("express")`.

2. When the code calls the module's default export `expressjs_namespace()` function, the symbolic execution engine will recognize the expression and assign a special symbol `expressjs_app` to the return value of the function. This value is usually stored in a constant variable, such as `const app = express()`.

   In addition, the symbolic execution engine will update the `expressApplicationCreated` flag in the symbolic environment to `true`. If the flag is already `true`, the symbolic execution engine will raise an error, as we assume there to be only one Express.JS application per project.

3. When the code calls one of the methods of the `expressjs_app` object, e.g. `app.use(…)`, the expression is usually broken down into two expressions: a `PropertyAccessExpression` that accesses the method name, and a `CallExpression` that invokes the method code with parameters.

   During evaluation of the `PropertyAccessExpression`, the symbolic execution engine will recognize the method name and return a set of special symbols representing each method, such as `expressjs_app_get`, `expressjs_app_post`, `expressjs_app_use`, etc. for the `app.get`, `app.post`, and `app.use` methods, respectively.

4. During evaluation of the `CallExpression`, the symbolic execution engine will recognize the special symbol representing the method name and update the internal model of the Express.JS application in the symbolic environment accordingly.

   - For the `app.use(handler, …)` method, the symbolic execution engine will append a new `Route` object to the `definedRoutes` list in the symbolic environment, which has the `catchAll` flag set to `true`, and the `handler` set to the handler function(s).
   - For the `app.set(key, value)` method, the symbolic execution engine will update the `expressSettings` map in the symbolic environment with the key-value pair of the setting name and setting value.
   - For the `app.get()` method, the symbolic execution engine will first need to determine from the number and type of parameters whether the method is being used to define a route or to retrieve a setting.

     If the method is being used to retrieve a setting, *i.e.* `app.get(key)`, the symbolic execution engine retrieves the setting from the `expressSettings` map in the symbolic

environment and returns the value of the setting. If the key is not found in the `expressSettings` map, the symbolic execution engine will raise an error. It shall be noted that the Express.JS engine also comes with a list of predefined setting names and default values, such as `"env"`, `"etag"`, `"x-powered-by"` [30].

If the method is being used to define a route, *i.e.* `app.get(path, handler, …)`, the symbolic execution engine will append a new `Route` object to the `definedRoutes` list in the symbolic environment, which has the `catchAll` flag set to `false`, the `method` set to `"GET"`, and the `path` and `handler` set to the path and handler function(s), respectively. The engine will also return the `expressjs_app` special symbol to support method chaining.

Note that, Express.JS allows the path to be a regular expression, such as `^\/user\/.+$`, or a templated path, such as `/user/:id`. Hence, for simplicity in design, the symbolic execution engine handles all paths as regular expressions. All regular string paths are converted to regular expressions by escaping special characters and adding the start and end of string tokens, so that they match exactly the same paths as the string paths. For path parameters, the parameter name is replaced with a regular expression that matches any character except for the path separator `/` and the query string separator `?`. For example, the path `/user/:id` is converted to the regular expression `^\/user\/[^?\/]+$`.

- For other HTTP methods like `app.post`, `app.put`, `app.delete` with parameters `(path, handler, …)`, the symbolic execution engine will follow the same procedure as the `app.get(path, handler, …)` method, but with the `method` set to `"POST"`, `"PUT"`, `"DELETE"`, etc., respectively.
- For the `app.all(path, handler, …)` method, the symbolic execution engine will follow the same procedure as the `app.get(path, handler, …)` method, but with the `catchAll` flag set to `true`.
- For the `app.route(path)` method, the symbolic execution engine will return the special symbol `expressjs_app_route` with the `path` name. When the `get()`, `post()`, or other HTTP methods are called on the `expressjs_app_route` object, the symbolic execution engine will follow the same procedure as above, but with the `path` set to the `path` name of the `expressjs_app_route` object.
- For the `app.listen(port, …)` method, the symbolic execution engine will update the `expressPort` value in the symbolic environment with the port number, and the `expressRunning` flag to `true`.

Multiple Models due to Multipath Execution

By now, we have established a model for the server code, composed of all of its routes and handlers. However, we may also end up with more than one model, especially if the server code contains branches. For example, the server may start up in different modes, depending on certain environment variables, and offer different paths, or handle requests differently. An example is shown in L30 in

Listing 3.4, where the server defines different fallback handlers based on the environment variable `NODE_ENV`. In that case, we would end up with two different models of the server, one for each branch of the if-else statement. The models would have the corresponding branching condition as its path constraint, and a different set of defined routes and handlers. This is a unique advantage of symbolic execution, as it is guaranteed to handle all scenarios of multiple path occurring. In contrast, most testing approaches assume the server to be a single non-changing entity, and ignores the variation in server initialization conditions.

4.4 Examining and Executing the Symbolic Model

In the second pass of RestAssured, we analyze the model(s) of the Express.JS application and determine whether they conform to the OpenAPI specification, following the properties in Definition 3.7. If there exists more than one model, we perform the following procedure for each model, preserving and building upon the path constraints from the earlier execution.

We first check the simple properties, such as soundness and completeness for specified endpoints. To put it more plainly, we check whether all specified endpoints are implemented and whether all implemented endpoints are specified. These properties are checked by examining the defined routes in the model and comparing them to the API endpoint definitions in the OpenAPI specification. The analysis does not need to invoke the symbolic execution engine. Then, we check more complex properties, such as the validity of the response status codes, the presence of required response body fields, and the correctness of response body types. To do so, we must symbolically execute handler functions for defined routes, and check for specification violations in the responses. For each defined route in the model, we generate symbolic requests based on the OpenAPI specification, and symbolically execute the handler functions to derive a set of possible symbolic responses. Then, we check for specification violations, such as invalid status codes, missing response body fields, or incorrect response body types. If the expected type contains refinement predicates, we invoke a constraint solver, Z3, to check if the predicates can hold for all possible values in the symbolic response.

4.4.1 Examining Router and Endpoints Definitions

Given a model of all defined routes, we are now able to check the first property in Definition 3.7, the soundness of specified endpoints, or the existence of the specified routes in the server implementation. For each of the API endpoint definitions in the OpenAPI specification, we check if there exists some non-`catchAll` route in the model that matches the path and method of the endpoint definition.

In routing the request, we simulate the routing logic of the Express.JS server, as described in section 3.3.2. The incoming request goes through all the defined routes in the order they are defined, and skips over routes that do not match the path or method of the request. Whenever we find a route with a matching path, we mark the API endpoint definition as fulfilled, and continue to the next endpoint definition. One particularity of the Express.JS routing logic is that it allows path to be stated as a regular expression. So during matching, we have to check if the path matches

the regular expression, instead of a simple string comparison. The same goes for templated paths, where the path may contain parameters that are extracted and passed to the handler function, such as `/user/id` (in OpenAPI definitions) or `/user/:id` (in Express.JS definitions). In the first pass, we have already converted all string paths to regular expressions, so we can directly run a regular expression match on the path of the request. In addition, observe that the path parameters in the OpenAPI definition (`{param_name}`) can be matched by our converted regular expression `[^?\/]+`. For example, the path `/user/:id` in the Express.JS definition is converted to the regular expression `^\/user\/[^?\/]+$`, which can match the path `/user/{id}` in the OpenAPI definition. Hence, we can conveniently run the regex match against full paths with parameters, without the need to parse and put in placeholder values.

Note, however, that the process here is only a preliminary check and may result in false negatives (but not first positives). It is perfectly possible that the handler for the matched route is unreachable in reality. For example, a `catchAll` middleware may terminate a request-response cycle early, or a matched handler may never produce an HTTP response due to a bug in the code. As a result, we will be able to immediately report to the user that certain HTTP paths are not fulfilled by any defined routes, but we may not be able to guarantee otherwise.

In the same process, we are also able to check the third property in Definition 3.7, the completeness of specified endpoints, or the non-existence of any implemented but unspecified routes. To do so, we create a dictionary `visited` for each entry in the defined routes. All values are initialized to `false`, except for the `catchAll` routes, which are initialized to `true`, for they will be trivially matched by any request. Then, during the routing process, we set the corresponding value in the `visited` dictionary to `true` when a route is matched. When all the specified routes have been checked, we go through the `visited` dictionary and report any routes that are still `false` as implemented but unspecified routes. While these routes do not violate specification conformity *per se*, they indicate sources of potential undocumented behavior or extraneous interfaces that should probably not be exposed. If some of these routes are intended for internal use, they should be guarded with some form of authentication or access control mechanism. Trying to hide them in the specification is a form of "security through obscurity" [1], a bad security practice.

4.4.2 Generating Symbolic Requests

Now, let us move on to the second property in Definition 3.7, the well-definedness of the handler functions, or the validity of the response produced by the handler functions. In other words, for all valid requests, the handler function should produce a response that is well-defined with regard to the specified response schema. To do so, we must invoke and symbolically execute the handler functions of the defined routes, and check for specification violations in the responses. However, to symbolically execute the handler functions, we need to first determine and symbolize the inputs to these handler functions, which include the request parameters, the request body, and the request headers. Fortunately, the OpenAPI specification provides a clear definition of the request for each endpoint. Hence, we generate symbolic requests from the OpenAPI specification, and use them as inputs to the handler functions.

The Symbolic Request Object

First, let us formalize how a request is being symbolized for our symbolic execution engine. A symbolic request contains the following components:

- `method: string`. The HTTP method of the request, e.g. `GET`, `POST`.
- `path: string`. The URL path of the request. A real-world path may also contain path parameters and query parameters that are part of the request. However, we choose to symbolize them separately for ease of analysis.
- `params: Map<string, Value<Type>>`. A map of the path parameters of the request. The keys are the parameter names, and the values are values of some `Type`.
- `query: Map<string, Value<Type>>`. A map of query parameters of the request. The keys are the parameter names, and the values are values of some `Type`.
- `headers: Map<string, Value<Type>>`. A map of custom HTTP headers used in the request. The keys are the header names, and the values are values of some `Type`.
- `body`. The request body of the request. The body may be a JSON object, a form data, or a file, depending on the content type in the specification. As a result, the type of this object is a union type of the possible types of the request body.

  The body is usually a JSON object (`application/json`), in which case the type would be: `{ type: "json"; value: { [key: string]: Value<String> } }`.
  The keys are field names of the JSON object, and the values are values of type string.

  The body can also be of type form data (`application/x-www-form-urlencoded`), in which case the type of the body would be similar to that of the JSON object: `{ type: "form"; value: { [key: string]: Value<String> } }`.

  Lastly, the body can be of MIME type `text/plain`, in which case the type would be: `{ type: "text"; value: Value<String> }`.

  Note that, in our current design, we only support three possible types of body schemas, JSON, form data, and plain text. We do not currently support files or vendor-specific content types, such as `application/vnd.github.v3.diff`, as specified in RFC 6838 [41].

Filling in a Symbolic Request

We then parse the OpenAPI specification and fill in the symbolic request object with symbolic values. For each defined endpoint in the specification, we generate a symbolic request object, and fill in the `method` and `path` fields with the method and path of the endpoint. Then, we parse the `parameters` field in the OpenAPI specification (at both the path and the method level), and fill in the path `params`, `query` parameters, and `headers` fields of the symbolic request object respectively, depending on the parameter type. For these parameters, we would keep their respective names, and insert a symbolic value of the declared type as their values. If no types are declared in the specification, we would default to type `string`. These symbols may then be interpreted as sets of

possible values during the symbolic execution of the handler functions. For request body, we first determine the MIME type of the body from the OpenAPI specification, and then fill in the `body` field similarly with defined field names and symbolic values.

One peculiarity of the OpenAPI specification is that a field may be an array of a given schema. Since the array is of the same type, we assume that all elements are of the same type of data. Thus, for simplicity of analysis, we model the array of objects in the same way as a singular object, with an extra symbolic field to represent the length of the array (see subsection 4.2.2).

4.4.3 Symbolically Executing the Handlers

After we have a symbolic request, we are to simulate its execution in the model of Express.JS application. Like in subsection 4.4.1, we simulate the routing logic of the Express.JS server, and go through each of the defined routes to derive the symbolic response object.

The Symbolic Response Object

The symbolic response object is similar to the symbolic request object, but is stored as a part of the symbolic environment. Because each request may go through multiple handler functions, and each request can only produce one response, it is the best design choice to store the response object in the symbolic environment, and update it as the request goes through the handler functions. The symbolic response object contains the following components:

- `expressResponseSent: boolean`. A boolean flag that indicates whether the response has been sent. If the response has been sent, the handler function should not send another response.
- `expressResponseStatusCode: number`. The status code of the response.
- `expressResponseBody: SymbolicValue<Type>`. The body of the response. The type of the body is usually a computed symbolic expression based on the symbols in the request. The `Type` is dependent on the handlers that produce the response body, which may be an `Object` or a `String`. In some cases, it may also be a concrete value, if the handler functions always produce some concrete value not dependent on the symbols in the request.
- `expressResponseHeaders: Map<string, SymbolicValue<Type>>`. A map of custom HTTP headers used in the response. The keys are the header names, and the values are similarly computed expressions dependent on the input symbols. Here, the `Type` is most commonly `string`, but depending on the handler functions, it may also be `number` or `boolean`.

Invocation of Handler Functions

For any given symbolic request, we will go through each of the defined routes in the order they are defined, skipping over routes with unmatching paths or methods, and symbolically execute the handler functions of the matched routes. There are a few special cases that we handle directly without invoking the symbolic execution engine, such as the built-in middleware of Express.JS like `express.json()` that parses the request body. When we encounter `express.json()`, we will treat

the `body` field of the symbolic request object as a parsed JSON object instead of a plain string, and continue the execution. For user-defined middlewares and handler functions, we will invoke the symbolic execution engine to simulate the execution of these functions, as if they are called by the Express.JS application, and update the internal model of the Express.JS application with the response.

To model the handler functions' interaction with the Express.JS library, we similarly augment the symbolic execution engine with a set of special symbols to represent the Express.JS library functions and objects, and intercept the execution of these functions, like we did in subsection 4.3.2. The algorithm works as follows:

1. When a handler or a middleware function is invoked, three parameters are usually passed in: `(req, res, next)`. The function may choose to only use `req` and `res`, or may choose to call `next()` to pass the control to the next middleware or handler function. Regardless, the symbolic execution engine models them as special symbols `expressjs_callback_req`, `expressjs_callback_res`, and `expressjs_callback_next`, and assigns them to the corresponding parameter variables in the function scope of the symbolic store.

2. Then, the symbolic execution engine executes the handler function as a normal function, until it recognizes one of the special symbols and intercepts execution.

3. If one of the expressions tries to access the request object `req`, the symbolic execution engine will intercept the expression and return the corresponding value from the symbolic request object.

   For example, if the program tries to access the request body (`req.body` field), the symbolic execution engine will recognize the expression and return either a plain string of the body or a JavaScript object representing the parsed JSON object, if the middleware `express.json()` has been called. If the program tries to access the path parameters (`req.params` field) or query parameters (`req.query` field), the symbolic execution engine will return a map of the parameters from the symbolic request.

   Alternatively, the program may also try to read a header from the request, such as `req.get("Content-Type")`, in which case the symbolic execution engine will return the corresponding value from the headers map in the symbolic request.

   There also exist some other fields, such as `req.cookies` that access the cookies in the request and `req.ip` which gets the IP address of the requester, which we do not currently support.

4. If one of the expressions tries to modify the response object `res`, the symbolic execution engine will intercept the expression and update the corresponding field in the symbolic response object, similar to how it handles the request object.

   If the program wants to send a response directly using the `res.send()` method, the symbolic execution engine will update the `expressResponseBody` field of the symbolic response object with the parameter passed to the `res.send()` method. It will also update the `Content-Type`

header in the `expressResponseHeaders` map, recognizing the type of the parameter passed to the `res.send()` method. If the `expressResponseStatusCode` field has not been set, it will be set to `200` by default. The symbolic execution engine will also update the `expressResponseSent` flag to `true`, to indicate that the response has been sent. Any further attempts to modify the response will result in an error. The similar is done for the `res.json()` method, which is a shorthand for `res.send()` with the `Content-Type` header specifically set to `application/json`.

If the program wants to set a non-200 HTTP status code (using the `res.status()` method), the symbolic execution engine will update the `expressResponseStatusCode` field of the symbolic response object with the parameter passed to the `res.status()` method. It will return the response symbol `expressjs_callback_res` to support method chaining. Note that the `res.status()` method does not send the response and can only be used before the response is sent out with `res.send()` or `res.json()`. There is also the `res.sendStatus()` method, which is a shorthand that sets the status code and then immediately sends the response.

There also exists some other methods that sends files (`res.sendFile()`), sets cookies (`res.cookie()`), or redirects the request (`res.redirect()`), which we do not currently support.

5. In Express.JS, the request may be matched by more than one handlers, and the control may be passed to the next handler by calling the `next()` function. When the handler function calls the `next()` function, the symbolic execution engine will recognize the special symbol and attempts to find the next middleware or handler function that matches the request to execute. When these functions are found, the same procedure will be applied recursively, until the list is exhausted or when the handler function ends without calling `next()`. When the callee returns, it returns to the next statement after the `next()` call in the caller, and the symbolic execution engine continues the execution of the caller.

6. If the handler function ends without calling `next()`, the symbolic execution engine will terminate the handling of the request, and deem the symbolic response produced as the final response.

Due to the possible existence of branching and multiple execution paths, we may end up with a set of possible symbolic responses and their corresponding path constraints. We will now turn to the last step of our algorithm, in which we check these responses for specification violations.

4.4.4 Checking Responses for Specification Violations

To check if an individual response is well-formed, we need to compare it to the list of valid responses in the OpenAPI specification. In other words, for each response, there must exist some defined response in the OpenAPI specification that:

- shares the same HTTP status code,
- contains all the defined HTTP headers of the correct types,

- and has a response body which contains all the required fields with the correct types.
- If the type contains refinement predicates, the predicates should hold true for all possible values of the field.

The verification for any given symbolic response is carried out as follows:

1. We first check the satisfiability of the path constraints to determine if the execution path producing the response is reachable. If the path constraint is unsatisfiable, the path leading to the response is considered unreachable. We will then discard the symbolic response and continue to the next response.

2. We then find the response with the same status code in the OpenAPI specification. If no such response is found, we report an error, as the produced response is not specified in the specification, resulting in a violation of the OpenAPI specification.

3. If we have located one such response (note that there can only be one response with the same status code in the OpenAPI specification), we then check the headers of the response. For each header (key) in the specification, the symbolic response must also contain the same header. If the header is missing in the symbolic response, we report an error, as the produced response is missing a required header, signaling a violation of the OpenAPI specification. If the header is present, we check if the type of the header in the symbolic response matches the type of the header in the OpenAPI specification. If the types do not match, we report an error, signaling a violation of the OpenAPI specification.

4. Lastly, we check the response body of the response. The response body must first be of the same type (a JSON object or a plain string) as specified. If it is a JSON object, it should have all the specified (required) fields. In addition, each field must have the correct type, as specified in the OpenAPI specification. If the type contains a refinement predicate, we invoke Z3 to check if the predicate holds for all possible values of the field. If any of the fields are missing or have the wrong type, we report an error, signaling a violation of the OpenAPI specification. If the specified response body is a plain string, the symbolic response is always correct, because the response body is always a string (either serialized JSON or plain text).

5. If all the checks pass, we consider the response to be well-formed, and continue to the next response.

The types in OpenAPI specification are more expressive than nominal types, and requires a technique more powerful than nominal type checking. The symbolic execution engine can check for existence of fields and their nominal types, by comparing the fields in the symbolic response with the expected primitive types in the OpenAPI specification. However, the OpenAPI specification can also have additional predicated attached to the types. For example, it may require a `number` to be within a `minimum` and `maximum` range, or require a `string` to be of `format: uuid` or even a regex pattern. When these predicates are present, the symbolic execution engine must invoke a constraint solver to check if the symbolic values satisfy the predicates. Hence, we translate the requirements

into SMT formulas, negate them, and disjunct them. We then conjunct the path constraints with the disjunction of the negated SMT formulas, and send the resulting formula to Z3 [24] to check for satisfiability. To put it more formally, for predicates $P_1, P_2, \ldots, P_n$ on a field f, we check the satisfiability of the following formula:

$$\text{Path Constraints} \wedge \left(\bigvee_{i=1}^{n} \neg P_i(f) \right)$$

In this way, if the symbolic value of the field can violate any of the predicates, the formula will be satisfiable, and we report an error to the user. The formula will only be unsatisfiable if for all possible concrete values represented by the symbolic value, the field can satisfy all the predicates. Hence, the response is considered well-formed if the formula is unsatisfiable.

Note that here we only choose to report this type of specification violation as a warning, as the violation may not always be a bug in the code, but are instead caused by an underconstrained input in the specification or lack of information in the code. For example, an endpoint may read a user's UUID from the database, and the OpenAPI specification may require the UUID to be a valid UUID string. However, having no knowledge of the database schema and the content stored, the symbolic execution engine cannot guarantee that the UUID will always be a valid UUID string. As a result, these violations may be false positives without user-added annotations. For now, we report them as warnings, so as to not distract users from more serious bugs that are guaranteed to be present.

4.5 Summary

In this chapter, we have presented the design of RestAssured, a symbolic execution-based tool for verifying the conformance of Express.JS applications to OpenAPI specifications. We have built a symbolic execution engine for RestScript, a subset of JavaScript that models most of the common language features used in a typical Express.JS application. We have also extended the symbolic execution engine with special symbols to represent the Express.JS library functions and objects, and to model the internal states of the Express.JS application. We have described the general workflow of RestAssured, which is summarized in the flowchart Figure 4.1.

Chapter 5

Implementation

And programming computers was so fascinating. You create your own little universe, and then it does what you tell it to do.
— Vint Cerf, in an AARP interview [23]

5.1 Overview

RestAssured is composed of several components, including a parser for TypeScript/JavaScript, a parser for OpenAPI Specification, a symbolic execution engine, an internal model simulating the Express.JS library logic, and a conformance checker that communicates with the Z3 constraint solver. Here, we briefly divide the components into three categories: input preprocessing, symbolic executor, and conformance checker. Note that these are not the only components in RestAssured. I have also written various small utility programs and tools to help with the development and evaluation of RestAssured, such as a TypeScript compiler transformer program to traverse and pretty print the AST of input programs, a JavaScript program to go into all benchmarks and extract the OpenAPI specification from JSDoc comments, a shell script to run all benchmarks and extract the relevant results, … For brevity, we will only discuss the main components in this chapter.

Most of these components are implemented in the TypeScript language, which allows us integration with the TypeScript Compiler API and ability to easily model RestScript execution semantics. In the following sections, we will briefly describe how these components implement the algorithms and techniques discussed in the previous chapter, and how they interact with other existing libraries and tools.

5.2 Input Preprocessing

RestAssured takes in two inputs, an implementation code written in RestScript and an OpenAPI Specification file. Hence, the entrypoint of RestAssured is the parser for both inputs.

5.2.1 Preprocessing and Parsing RestScript Codes

RestScript is a subset of TypeScript. Hence, we are able to use any TypeScript/JavaScript-compliant tools to preprocess and parse the RestScript code.

Firstly, we preprocess the input RestScript code to inline all modules and dependencies into a single file, so we do not need to properly support modules (either CommonJS or ES modules) in our symbolic execution engine. This is done by the `rollup` bundler [88] with the configuration shown in Listing 5.1.

```
import commonjs from "@rollup/plugin-commonjs";
import { nodeResolve } from "@rollup/plugin-node-resolve";
import autoExternal from "rollup-plugin-auto-external";
import { babel } from "@rollup/plugin-babel";

export default {
  input: "app.js",
  output: {
    dir: "output",
    format: "cjs",
  },
  external: ["express"], // do not include expressjs, which we model internally
  plugins: [
    // comment next line to include external dependencies for analysis
    autoExternal(), // ignore external dependencies in the bundle
    commonjs(), // cjs modules require()'d
    nodeResolve(), // es modules import'ed
    // Transpile with Babel to ES5
    babel({
      babelHelpers: "bundled",
      presets: ["@babel/preset-env"],
    }),
  ],
};
```

Listing 5.1: Rollup configuration to inline all internal modules

We configure `rollup` to inline both CommonJS and ES modules, but only do so for internal modules (i.e., modules that are part of the implementation repository). We mark `express` explicitly as an external module to exclude it from the bundle, as we choose to model it in the symbolic execution engine. In addition, we also choose to ignore all external modules (*i.e.* dependencies that are installed by `npm` or `yarn`) in the bundle. While including those modules will allow us to conduct a more comprehensive analysis, it will also significantly increase the complexity of the code under analysis. Library code is often written in a general way to support various use cases. However, the generality leads to code complexity and create a large number of execution paths for the symbolic execution engine to explore, which may not be necessary for the analysis of the specific server under test. As a result, we think it is more beneficial to focus on the internal modules written by the developer and model calls to external modules as an uninterpreted function in the symbolic execution engine. In this way, we also maintain the soundness of the analysis, as an uninterpreted function can be used to represent any function with unknown behavior.

Last but not least, we transpile the code to ES5 using the `babel` plugin [3], because RestScript more closely resembles ES5 syntax. By rewriting many modern JavaScript features into ES5, they have a larger chance of being supported by our symbolic execution engine. Ideally, we could also write a `babel` preset that directly transpiles modern JavaScript/TypeScript to RestScript syntax, but we leave it as a future work. However, note that if `babel` is run on TypeScript code (with `"@babel/plugin-transform-typescript"` plugin), it will strip out the type annotations and interfaces, leading to loss of information for the analysis. Therefore, we only enable `babel` if the input code is in JavaScript and contains ES6+ features.

After preprocessing, we are to now parse the RestScript code into an abstract syntax tree (AST). Since RestScript is a proper subset of TypeScript, we can use the TypeScript Compiler API to do so. Specifically, we will use `fs.readFileSync()` to read the preprocessed file and use the `ts.createSourceFile()` function to parse the code into an AST. The TypeScript Compiler API provides us many ways to traverse the AST, which we can use to print out and visualize the AST nodes. For example, we can parse the motivating example in Listing 1.1 into an AST shown in Figure 5.1. Even for a simple 15-line program, the AST could consist of 103 nodes of 24 different kinds. In fact, the TypeScript Compiler API provides a total of 363 different `SyntaxKind`s for the AST nodes [67]. As shown, the AST of a real-world programming language is often a complex data structure that takes much effort to understand and manipulate. We will discuss how this AST is then manipulated and executed in the symbolic execution engine in section 5.3.

5.2.2 Generating and Parsing OpenAPI Specifications

Other than the implementation code, RestAssured also takes in an OpenAPI Specification file. For projects that come with a `swagger.json` or `openapi.yaml` file, we can directly take the file as input. However, many projects do not have a separate swagger file. Instead, the swagger definitions are found in the source code, often in the form of JSDoc comments, as in the motivating example in Listing 1.1. In this case, we need to parse the JSDoc comments to extract the OpenAPI Specification. We have written a small JavaScript program to automatically invoke `swagger-jsdoc` [97] to do so for each project we analyze.

Given an OpenAPI specification, either in the form of a `yaml` or `json` file, we use a third-party library called `openapi-ts-json-schema` [14] to translate it into a TypeScript object. In this way, we can easily access and manipulate them in the symbolic execution engine. For example, the OpenAPI Specification in Listing 1.1 can be parsed into a TypeScript object as shown in Listing 5.2. The TypeScript object is then able to be read in by the symbolic execution engine to generate symbolic requests and check symbolic responses for conformance properties.

Other than `openapi-ts-json-schema`, we have also experimented with some of the more popular libraries, such as `swagger-codegen` [99] and `openapi-typescript` [81]. However, both of these libraries generate TypeScript interfaces for the OpenAPI Specification types. While it is useful for type-checking and code completion for developers, it is not as useful for our symbolic execution engine; because when TypeScript codes are compiled to JavaScript, the type information (which interfaces are a part of) is erased. Therefore, the symbolic execution engine will not be able to

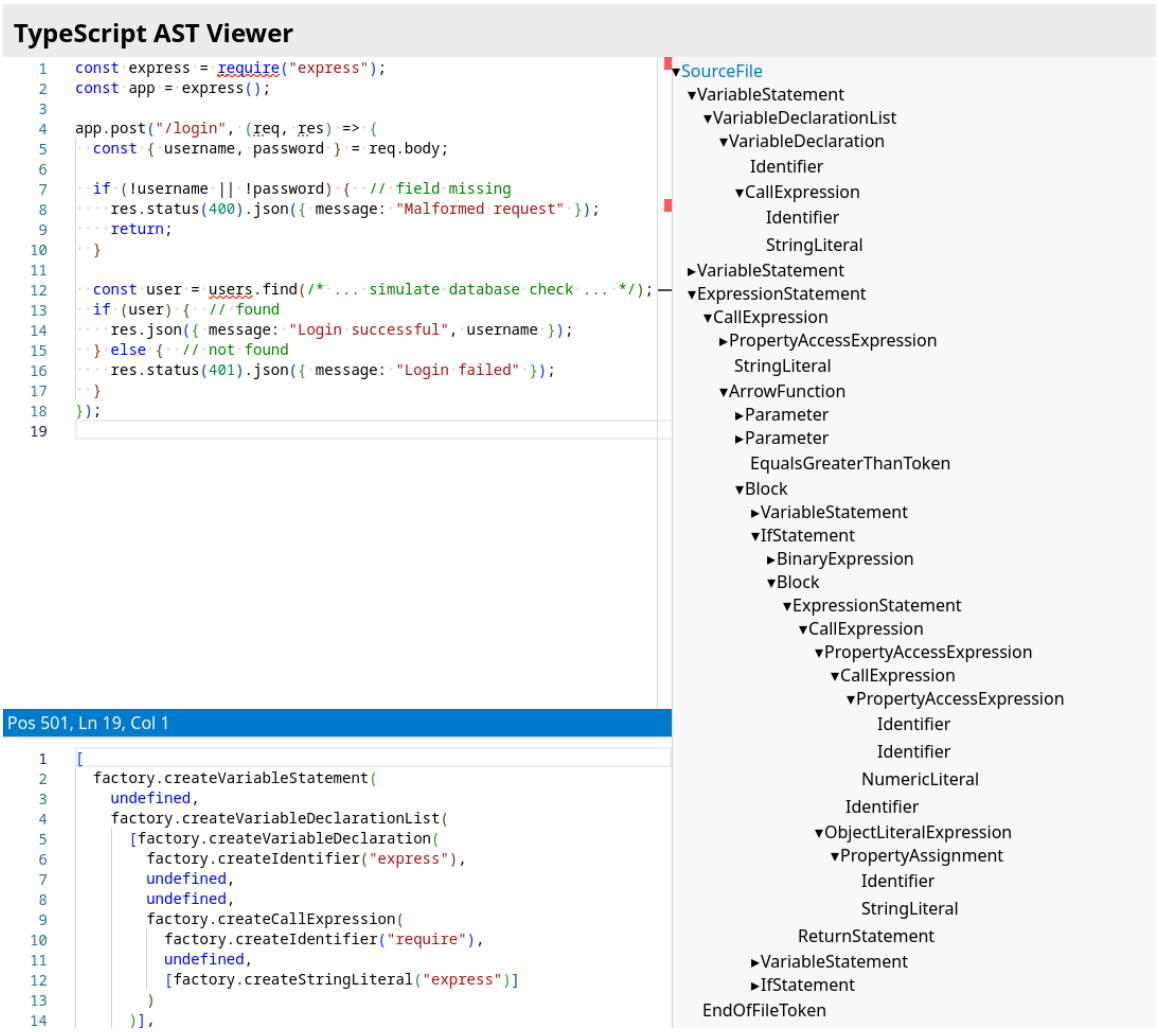


Figure 5.1: Abstract Syntax Tree (AST) for the motivating example in Listing 1.1. The AST is visualized by the online tool TypeScript AST Viewer [92], which visits all nodes recursively via the TypeScript Compiler API `node.forEachChild(node => …)`.

The source code is shown on the upper left, and the corresponding AST is shown on the right. The lower left box shows the factory code to generate the AST nodes using the TypeScript Compiler API. The AST is partially collapsed to fit the screen.

```ts
export default {
  post: {
    summary: "Login to the application.",
    requestBody: {
      required: true,
      content: {
        "application/json": {
          schema: {
            type: "object",
            properties: {
              username: { type: "string" },
              password: { type: "string" },
            },
          },
        },
      },
    },
    responses: {
      "200": {
        description: "Login successful.",
        content: {
          "application/json": {
            schema: {
              type: "object",
              properties: {
                message: {
                  type: "string",
                  description: "A success message.",
                },
                username: {
                  type: "string",
                  description: "Username.",
                },
              },
            },
          },
        },
      },
      "401": {
        description: "Login failed",
        content: {
          "application/json": {
            schema: {
              type: "object",
              properties: {
                message: {
                  type: "string",
                  description: "An error message.",
                },
              },
            },
  }}}}},
} as const;
```

Listing 5.2: The Parsed TypeScript object representing the OpenAPI Specification in Listing 1.1. The object is generated by the `openapi-ts-json-schema` library. Output is partially collapsed to fit on page.

access these type information at runtime, when it wants to compare the actual response with the expected types. Hence, we chose the relatively lesser-known library `openapi-ts-json-schema` to generate a TypeScript object that can be accessed at runtime.

5.3 Symbolic Execution Engine

As discussed in section 2.4.3, no existing symbolic execution engine fits our need. Hence, we have to implement our own symbolic execution engine from first principles. The symbolic execution engine is implemented in TypeScript, which allows us to easily manipulate the TypeScript AST and model the semantics of the RestScript language.

The core principles have already been detailed in section 4.2, and we will now discuss how we implement these principles in the symbolic execution engine. This section will primarily be divided into two parts: the data structure used and the control flow of the symbolic execution engine.

5.3.1 Data Structure

All internal data are organized in the `SymbolicState` class. An instance of `SymbolicState` represents a single symbolic execution path. It consists the symbolic store, the path constraints, the AST node to be executed, and internal express states.

The path constraints are an array of SMT-LIB formulae [6] that represent the constraints that must be satisfied for the path to be feasible. For example, a path constraint could be

```
[(declare-const x Int), (assert (>= x 0)), (assert (< x 10))]
```

The symbolic store (sometimes also referred to as symbolic tables) is a map from variable names to their `Values`. A `Value` can be either a concrete value (like a number or a string) or a symbolic value (some SMT formula using declared symbols). The symbolic store is implemented as a stack of maps, where a new map is pushed onto the stack whenever a new scope is entered. The stack begins with a `built_in` scope, which contains the built-in functions in RestScript. Each following stack has an id to identify the scope, in the format of `type_pos_x`, such as `block_pos_0`. Here the `pos` is the position of the AST node in the source file, provided to us by the TypeScript Compiler API. This allows us to model the scoping rules of RestScript, where variables declared inside a scope is not accessible outside it.

The internal express states are special fields as detailed in subsection 4.3.1.

5.3.2 Control flow

The TypeScript Compiler API provides us many ways to traverse the AST. The most common way is to create a transformer function using a visitor pattern and the `ts.visitNode()` function. However, we choose to walk through the AST ourselves, so we can model the control flow in the symbolic execution engine. We will recursively execute the AST, until we reach a control flow indirection, such as a function call or a conditional statement.

When a condition is detected, the execution path branches. We create a deep copy of the current `SymbolicState`, and store it in a doubly linked list with the current state. When copied, both the new state and the current state would have the path constraints updated with the branch condition and the activation record updated with the AST node to resume from. This allows us to take one path and then resume execution from the other path.

For function calls and loops, we will need to create new activation records (AR) to store the state of the function call. However, we do not fully model the ARs as the JavaScript runtime does. Instead, we model ARs as a special scope in the symbolic store with special types like `function` and `loop`. They store the AST node to return to after exiting the current AR, the parameters passed to the function, the value returned, and special control flow symbols to determine if the function has been `return`ed or if the loop should `continue` or `break`.

However, even though we have stored return addresses in the AR, we cannot directly `jump` to the return address like in C. This is because JavaScript, the language our symbolic execution engine runs on, does not allow programmers to directly manipulate the program counter easily. Therefore, we have to rely on the JavaScript runtime to resume execution from the correct AR. This is done by preserving the call stack of the symbolic execution engine. When we call a function, we push a new AR onto the stack and instructs the symbolic execution engine to execute the function body. When the function body finishes execution, the symbolic execution engine has to return from the function call evaluation function to the previous stack and resume execution from there. While this is straightforward for a function without branching, it becomes more complicated when there are multiple paths in the function body. In the latter case, we have to finish all pending paths in the current AR before we can return to the previous AR. Therefore, we prevent the need for the symbolic execution engine to jump to a specific AST node in the AR, and instead models execution by preserving the engine call stack.

5.4 Z3 Interface

Our symbolic execution engine uses the Z3 solver [24] to solve for SMT formulae in both path constraints and symbolic values in the produced REST responses. While Z3 does have a JavaScript binding, it is not as feature-complete as the official Python or C API. For example, it does not support the string and regular expression theories [63], which are essential for conformance analysis against refined predicates in the OpenAPI types. Moreover, the JavaScript binding runs in WebAssembly and has some weird out of memory bugs for even the simplest SMT formula that we have not been able to resolve.

As a result, we have turned to use the Python API for Z3, which requires us to set up a REST API service ourselves to communicate between the symbolic execution engine in JavaScript and the Z3 solver in Python. We have written a small RESTful server using Flask [75] to receive the SMT formulae from the symbolic execution engine, solve them with Z3, and return the results back to the symbolic execution engine. The symbolic execution engine will send requests with `fetch()` Node API to the Flask server and wait for responses to parse them and update the symbolic store with the results.

Chapter 6

Evaluation

Beware of bugs in the above code; I have only proved it correct, not tried it.

— Donald Knuth, in a memo to Stanford students [52]

Let us now evaluate the prototype implementation of RestAssured we have built empirically. We have gathered a set of benchmarks, consisting of both synthetic and real-world programs/specifications, to run RestAssured on. We shall use these results to answer the following research questions (RQs):

1. Is RestAssured reasonably accurate with little false negatives, *i.e.* verified programs will conform to specification? (section 6.2)
2. Does RestAssured run with reasonable speed? (section 6.3)
3. Does RestAssured produce actionable output? (section 6.4)

6.1 Gathering Benchmarks

We use a set of synthetic benchmarks inspired by real-world programs and bugs to evaluate our symbolic execution engine. We established our ground-truth positives by searching for keywords like "OpenAPI specification error" and "wrong Swagger specification" on GitHub. However, not many projects have explicit bug reports or issues related to OpenAPI specification errors. As a result, we also searched directly for any projects with keywords like "Express.JS AND Swagger" to find real-world programs that use the Express.JS framework to build API endpoints with OpenAPI specification. We then use these programs to understand the common types of mistakes and reproduce them in the RestScript language. We have also handcrafted some programs, so the benchmarks cover all cases of specification violations in Definition 3.7. These benchmarks are listed and described in Table 6.1. They are numbered from `0A` to `5D`, where the digit represents the type of violation being tested (except for 0 denoting miscellaneous programs) and the letter represents the specific benchmark within that type.

No.	Description	Expected Output
0A	Hello World program	Conform
0B	Branched Hello World program	Conform
0C	The motivating login example (Listing 1.1)	Error: Violation
0D	Branched Express.JS server definition	Conform for one server model and Error: Violation for the other
1A	Specified endpoint not implemented	Error: Violation
1B	All specified endpoints implemented	Conform
1C	Implementation contains unspecified endpoint	Warning: Unspecified
2A	Handler returns unspecified HTTP Code	Error: Violation
2B	Handler returns only specified HTTP Code	Conform
2C	Specified HTTP Code not returned	Warning: Overspecified
3A	Handler returns unspecified body MIME type	Error: Violation
3B	Handler returns only specified body MIME type	Conform
4A	Handler returns missing required field	Error: Violation
4B	Handler returns all required fields	Conform
4C	Handler returns extra unspecified field	Warning: Unspecified
5A	Handler returns field with wrong primitive type	Error: Violation
5B	Handler returns field with correct primitive type	Conform
5C	Handler returns field with unspecified refined type	Warning: Possible Violation
5D	Handler returns field with specified refined type	Conform

Table 6.1: The 19 synthetic benchmarks we have handcrafted to test RestAssured on.

No.	Endpoints	Branches	Conformant Branches	Time (sec)	Verdict	Expected
0A	1	1	1	2.31	Conform	✓
0B	1	2	2	2.79	Conform	✓
0C	1	3	2	2.90	Violate	✓
0D-1	1	1	1		Conform	✓
0D-2	1	1	0	2.42	Violate	✓
1A	2	1	1	2.60	Violate	✓
1B	2	2	2	2.34	Conform	✓
1C	1	1	1	2.21	Conform (Warn)	✓
2A	1	3	2	2.29	Violate	✓
2B	1	3	3	2.52	Conform (Warn)	✓
2C	1	2	2	2.33	Conform	✓
3A	1	3	1	2.46	Violate	✓
3B	1	3	3	2.54	Conform	✓
4A	1	3	2	2.41	Violate	✓
4B	1	3	3	2.49	Conform	✓
4C	1	3	3	2.35	Conform (Warn)	✓
5A	1	3	2	2.69	Violate	✓
5B	1	3	3	2.33	Conform	✓
5C	1	3	3	2.59	Conform (Warn)	✓
5D	1	3	3	2.54	Conform	✓

Table 6.2: The results of running RestAssured on the synthetic benchmarks. Note that for benchmark 0D, two different server models are produced due to the branched server definition. For benchmark 1A, although all produced branches are conformant, the program fails to implement a specified endpoint and thus is still non-conformant with regard to the specification.

6.2 Is RestAssured accurate?

The first research question we ask is whether RestAssured is accurate. In other words, **if a program passes verification, does it conform to the specification (no false negative)**? And also, if a program fails verification, does it necessarily violate the specification (no false positive)?

We have run RestAssured on the 19 synthetic benchmarks and recorded the results in Table 6.2. The results show that RestAssured is able to accurately detect the 7 types of violations of the OpenAPI specification in the synthetic benchmarks, and give warnings for the 3 types of situations where the program is conformant to the specification but the two disagree. Note, however, that for benchmark `5C`, although the produced output may be non-conformant with regard to the refined type of the field, we do not report it as a violation but as a warning of a possible violation. We intentionally downgrade the severity of this type of violation to reduce the number of false positives and to not overwhelm the user with too many error messages. As a result, RestAssured will gain a lower false negative rate at the cost of a higher false positive rate. We will discuss this choice and the rationale behind it in section 6.4.

Another interesting observation is that even for the simplest program (`0A`) that produces a fixed HTTP 200 OK response with body "Hello World", an interprocedural analysis would be required, since we need to simulate Express.JS's behavior to invoke the handler function and how it constructs the response from possible requests. For the same reason, we have chosen to build an internal model of the Express.JS framework in our symbolic execution engine, which allows us to conduct the interprocedural analysis and reason about the behavior of the program more effectively. Naturally, this simplification renders the analysis not totally sound, but we believe it is a reasonable trade-off to make the analysis more practical and usable. There are also some other simplifications, which we discuss in section 7.2.

Over all, although RestAssured is not sound for practical reasons, we believe it is accurate enough to be useful for developers to verify their programs against the OpenAPI specification. From the empirical evaluation results, it is able to detect all violations of the OpenAPI specification. For benchmark `5C`, we downgrade the error message to a warning, since RestAssured can only reach the conclusion of a "Possible Violation" without further information from the user. We will discuss whether this benchmark falls into the false positive category, and how user can provide more information to RestAssured to make the analysis more precise in section 6.4.

6.3 Does RestAssured run with reasonable speed?

Another important question to answer is: **does RestAssured run with reasonable speed?** Static analysis techniques, especially symbolic execution, is known to be computationally expensive, due to the exponential explosion of path numbers in a program [4].

From our implementation of RestAssured, we can estimate a rough upper bound for the time complexity of the algorithm:

$$O(n \cdot m \cdot b \cdot s \cdot t)$$

```
[INFO] Reading API spec: ../examples/0c-login/swagger.json
[INFO] Reading file: ../examples/0c-login/server.js
[INFO] Symbolic execution complete. We now have a symbolic model for the Express.JS
    ↪ application.
[INFO] Found 1 server models in all branches of the program.

[INFO] Generating symbolic requests for the server...
[INFO] Injecting symbolic request: POST /login

[INFO] Server received request: POST /login
[INFO] Executing middleware: express.json()
[INFO] Executing handler: POST /login (function-anonymous-pos-1387-1851)

[INFO] Symbolical execution complete. 3 branches found. We will now check them for
    ↪ conformance with specifications.

[INFO] checking response state (HTTP 400) conformance with Path Constraints:
(declare-const username String)
(declare-const password String)
(assert (or (= username "") (= password "")))
[INFO] SAT:
(define-fun username () String
  "")
(define-fun password () String
  "")
[ERROR] response status code 400 not specified in swagger API spec. VIOLATE
[INFO] expected (valid) codes:
[ 200, 401 ]
[INFO] State 1: VIOLATE.

[INFO] checking response state (HTTP 200) conformance with Path Constraints:
...
[INFO] State 2: CONFORM.

[INFO] checking response state (HTTP 401) conformance with Path Constraints:
...
[INFO] State 3: CONFORM.

[INFO] ANALYSIS for endpoint POST /login:
   Out of 3 states, 3 are SATisfiable/reachable. In which 2 CONFORMs to specification.

[INFO] Symbolical execution complete

[INFO] FINISHED. VERDICT FOR THE SERVER: VIOLATE
  Out of 1 total paths and 1 endpoints specified, 0 are conforming to the API spec.
  The symbolic execution produced 3 states, of which 3 are SATisfiable/reachable and 2 are
    ↪ conforming to the API spec.
[INFO] MESSAGES:
[POST /login] VIOLATE: Response status code 400 not specified in swagger API spec.
```

Listing 6.1: Sample output of running RESTASSURED on Benchmark `0C`, the motivating example Listing 1.1. Some parts of the output have been omitted for brevity. For full output, see section C.1. Observe that RESTASSURED is able to detect the violation of the OpenAPI specification and point out that the response status code 400 (produced in `State 1`) is not specified in the API spec. RESTASSURED is also able to detect that in two other branches (states), the program produces the correct response status codes and the correct response body.

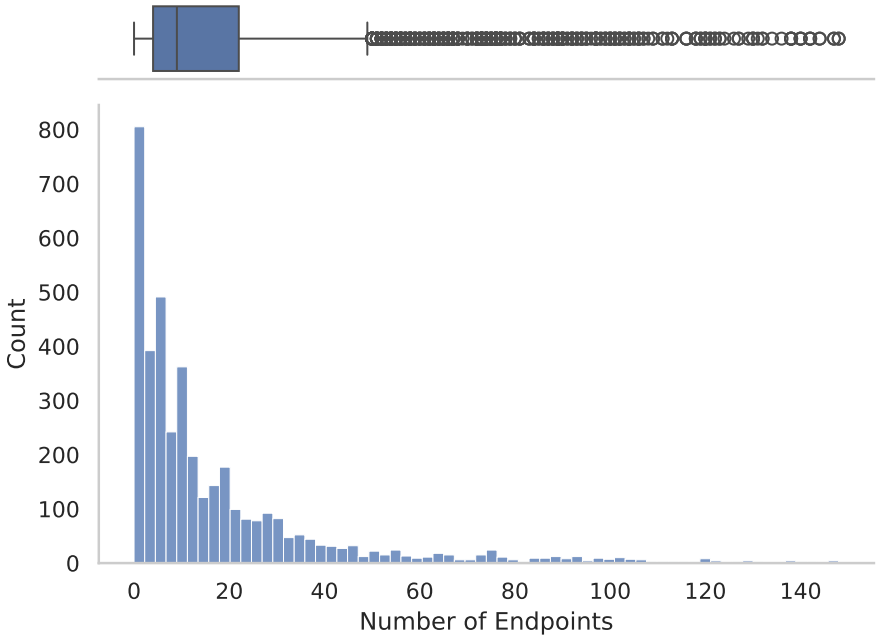


Figure 6.1: The distribution of the number of endpoints in 4,110 public OpenAPI specifications in the GitHub directory `APIs-guru/Openapi-Directory`.

Out of 4,110 programs, 3,938 programs (97.01%) have 150 or fewer endpoints. The 123 programs with more than 150 endpoints are considered outliers and thus not shown. The maximum number of endpoints is 22,361; which skews the average number of endpoints to 37.52, with a standard deviation of 401.29.

The box plot shows the median (10), first quartile (4), and third quartile (24) of the number of endpoints, and considers any data points outside the whiskers $(0 - 38)$ as outliers. The histogram shows the frequency of the number of endpoints for each program.

where n is the number of different defined server models, m is the number of endpoints in each server, b is the number of branches/leaf states created during symbolic execution, s is the number of program statements in each branch, and t is the time complexity of the Z3 solver given the path constraints.

Although the formula looks daunting with many factors, it is important to note that many of them are usually small in practice. For example, most server programs tend to only define a single server. Even when there are multiple servers, the number is usually small, accounting for debug/testing purposes and initialization failures. The number of endpoints is also usually small. We surveyed 4,110 public API specifications in the GitHub directory `APIs-guru/Openapi-Directory` [84] and found that the median number of endpoints is 10, and 75% of specifications have 24 or fewer endpoints. However, there does exist some extreme outliers, with the maximum number of endpoints being 22,361. The distribution for all specifications with less than 150 endpoints (97.01% of all 4,110 programs) is plotted in Figure 6.1. The results are a strong indicator that the number

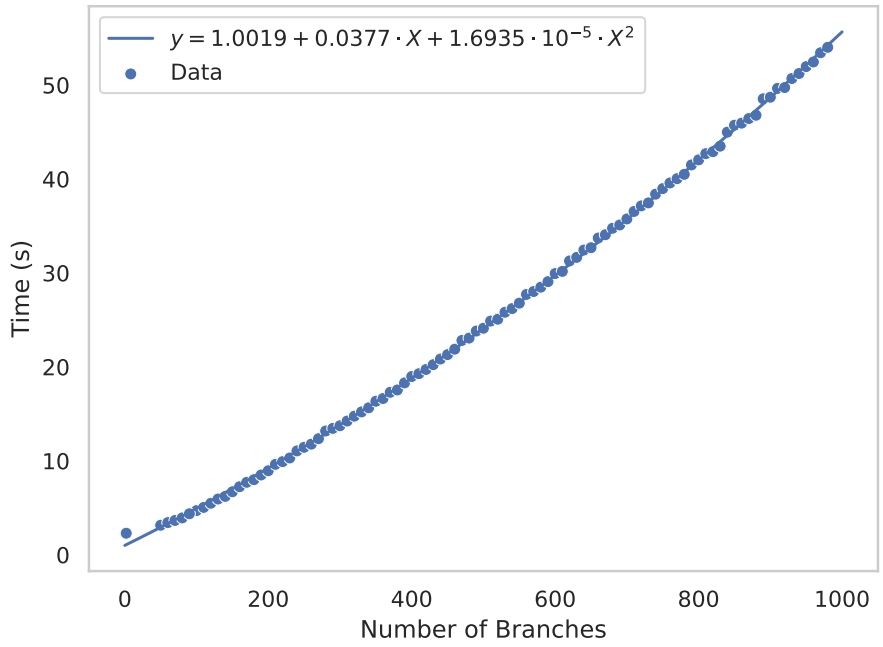


Figure 6.2: The time taken to run RestAssured on generated programs with different branch numbers from 1 up to 980. For branch numbers greater than 990, we hit the Node.JS maximum call stack size limit and the program crashes. The data is fitted with a polynomial curve of degree 2, which is the best fit according to cross-validation. In all cases, RestAssured finishes the analysis within a minute.

of endpoints will not be a major factor in the time complexity of RestAssured.

This leaves us with the number of branches and the number of statements in each branch as the main factors that affect the performance of RestAssured. It is commonly believed that the number of branches in a program grows exponentially with the length of the program, because each conditional statement can split the program into two branches [4]. However, the claim is only true if the program produces a full binary tree of execution paths, so the number of leave states is 2^d where d is the depth of the tree. As a result, we do not model our O-analysis in terms of the length of the program, but rather the number of branches times the number of statements in each branch, which will produce a more accurate estimate of the time complexity of RestAssured.

We also create a set of synthetics programs with arbitrary number of branches (see section C.2), start from a simple program with a single branch up to 1000 branches, with a step size of 10. We then run RestAssured on these programs and measure the time taken to analyze the program. We are able to run RestAssured on programs with up to 980 branches, after which we hit the Node.JS maximum call stack size limit and the program crashes. The results are plotted in Figure 6.2. For the results, we used a linear regression model to fit polynomial curves of degree 1 to 10, and performed cross-validation to select the best model, which is of degree 2. Assuming the number of server models and endpoints are kept constant, the curve of best fit shows that the time taken to

run RestAssured scales quadratically with the number of branches in the program. It is worth noting that the quadratic term is very tiny (of 10^{-5} order of magnitude), making the overall time complexity of RestAssured almost linear with the number of branches. The result is consistent with our theoretical analysis. As the number of branches increases, each state also produces more path constraints, which may slightly increase the time it takes for the Z3 solver to determine the satisfiability of the constraints.

Overall, we show that RestAssured runs with reasonable speed, and can finish all synthetic benchmarks within 3 seconds (in Table 6.2). Although the number of branches may exponentially explode with the length of the program, we believe that the number of branches in a program is usually smaller in practice. We empirically show that the time taken to run RestAssured scales quadratically with the number of branches in the program, with a very small quadratic term. For generated programs with up to 980 branches, RestAssured is able to finish the analysis within a minute. We did, however, discover that RestAssured crashes when the number of branches exceeds 990, due to the Node.JS call stack size limit. As a result, we believe that the time complexity of RestAssured is reasonable.

6.4 Does RestAssured produce actionable output?

The final research question we ask is: **does RestAssured produce actionable output?** In other words, is the output of RestAssured usable and understandable by real people? We will try to answer the question by analyzing the output of RestAssured on the synthetic benchmarks, for both true positives and potential false positives.

6.4.1 Output for True Positives

For confirmed positive violations, the output of RestAssured is clear and actionable. For example, in Listing 6.1, we show the output of running RestAssured on the motivating example (Listing 1.1). For the final verdict, RestAssured produces the following output:

```
Out of 1 total paths and 1 endpoints specified, 0 are conforming to the API spec.
The symbolic execution produced 3 states, of which 3 are SATisfiable/reachable and 2 are
    ↪ conforming to the API spec.
[INFO] MESSAGES:
[POST /login] VIOLATE: Response status code 400 not specified in swagger API spec.
```

The output clearly states the reason for violation. In this case, the response status code 400 is not specified in the OpenAPI specification. As a result, if the user wants to modify the specification, they can easily add the unspecified status code to the specification.

Furthermore, in the detailed log, RestAssured is able to pinpoint the exact state where the violation occurs, and use Z3 to synthesize the input values that lead to the violation.

```
(define-fun username () String
  "")
(define-fun password () String
  "")
```

As a result, if the user wants to modify the program instead of the specification, they can use these concrete input values to reproduce the violation in their own environment and debug the program.

For a more complicated reason of violation, RESTASSURED is also able to provide a detailed explanation of the violation. For example, for benchmark `4A` and `5A`, RESTASSURED produces the following verdict messages respectively:

```
[POST /login] VIOLATE: Expected field username in response body, not found.
```

```
[POST /login] VIOLATE: Expected field username to be type number, but got Symbol<string>.
```

Similar to the sample output, RESTASSURED is able to provide a clear and actionable message for the user to fix the violation, either in the program or in the specification.

6.4.2 Case Study: Benchmark `5C`

Benchmark `5C`: unsatisfied refinement predicate is a more complicated case, where the violation is not as clear-cut as the others. The source code for this benchmark is shown in section C.3, which is an adaptation of the motivating example (Listing 1.1). In this benchmark, the specification requires the returned `username` field to be a string with a minimum length of 3 characters and a maximum length of 10 characters.

During symbolic execution of the program, RESTASSURED is able to detect that the program produces a response with a `username` field that is exactly the same as the input `username` field in the HTTP body. However, since RESTASSURED does not have enough information to determine the exact length of the input `username` field, it will report the following message:

```
[INFO] checking refinement predicate for field username
[WARN] response body field username may not satisfy refinement predicate. POSSIBLY VIOLATE.
[INFO] model:
(define-fun username () String
  "A")
(define-fun |str_pos_645_Hello,_| () String
  "Hello, ")
(define-fun str_pos_645_! () String
  "!")
(define-fun password () String
  "my_super_secret_password")
```

From the output, if the input `username` field is set to the string "A", the output `username` will also violate the refinement predicate. Similarly, if the response contains some value from an external source, such as the database or a system call, RESTASSURED will not be able to have enough information to construct the exact set of values that may be returned.

As a result, RESTASSURED may produce a false positive for these cases, if the user will only send a username between the length of 3 and 10 characters. We believe that this is a reasonable trade-off, because we favor security properties and lean towards producing less false negatives, which will give the user a false sense of security. In this case, the user can provide more information to RESTASSURED to make the analysis more precise. For example, for this specific benchmark, the user will be able to constrain the input `username` field to be a string with a length between 3 and 10 characters. As a result, RESTASSURED will be able to add the following path constraint to the Z3 solver:

```
(assert (>= (str.len username) 3))
(assert (<= (str.len username) 10))
```

Thus, it will determine that the returned `username` field will always satisfy the refinement predicate. If, however, the `username` field is produced from a database query, the user may need to specify through inline comments that, *e.g.*, the `username` returned from a database query statement is always between 3 and 10 characters long. However, in most cases, given the static nature of RestAssured's analysis, it will not be able to have enough information to determine the exact set of values that may be returned. So, we choose to downgrade the severity of the error message to a warning, to reduce the number of false positives and to not overwhelm the user with too many error messages.

Overall, we believe that the output of RestAssured is actionable and understandable. It may produce false positives if refinement predicates are present in the OpenAPI types, but we believe that this is a reasonable trade-off to favor security properties and to reduce the number of false negatives. Moreover, the user is able to provide annotations to RestAssured to reduce the number of false positives. Currently, these errors are reported as warnings, and the user can choose to ignore them or to provide more information to RestAssured to make the analysis more precise.

Chapter 7

Conclusion and Discussions

> *Even perfect program verification can only establish that a program meets its specification. [...] much of the essence of building a program is in fact the debugging of the specification.*
>
> — Fred Brooks, in "No Silver Bullets" [10, p. 16]

7.1 Summary

In this thesis, we explore formally verification techniques to verify a RESTful API service's conformance to its specification properties. I have presented RestAssured, a static white-box analysis algorithm that formally verifies a RESTful API service implemented with Express.JS in JavaScript/TypeScript against its OpenAPI specification. I also propose RestScript, a restricted subset of JavaScript/TypeScript that is widely used for implementing RESTful APIs and is amenable to static symbolic execution. This thesis describes a specific implementation of the RestAssured algorithm built for RestScript programs.

RestAssured takes in two inputs, the OpenAPI specification and an Express.JS server implementation, and verifies that the implementation conforms to the specification. It uses symbolic execution to create models of the implementation's routing and request handling logic, exploring all possible execution paths if the program creates a set of different server definitions, `e.g.` for testing and production environments. RestAssured then generates symbolic requests using the OpenAPI specification and simulates the server's behavior in invoking middlewares and route handlers. For each possible response produced, RestAssured checks that it conforms to the specification. When a violation is found, or when the implementation disagrees with the specification, RestAssured reports the discrepancy to the user, generating concrete counterexamples to aid debugging.

Using empirical evaluation on a set of synthetic benchmark programs inspired by real-world errors, we demonstrate that RestAssured is effective and accurate in finding discrepancies between the OpenAPI specification and the implementation. It runs with reasonable performance, finishing the verification process in a minute for programs up to 980 branches. Furthermore, its output is usually informative and detailed enough to help developers pinpoint the root cause of the discrepancy,

or provide concrete counterexamples to help reproduce the nonconformance behavior. Last but not least, as a static analysis tool, RestAssured is able to find discrepancies that are not easily found by traditional testing methods. For example, in benchmark `OD`, RestAssured is able to identify that the implementation produces two different server definitions for the testing and production environments and analyze them separately, which is not possible with traditional testing methods. RestAssured is also able to identify in benchmark `1C` that the implementation contains endpoints that are not specified, which cannot be easily identified if the testing only uses the specification as the test oracle.

At the time of proposal, to the best of author's knowledge, RestAssured is the first tool that uses purely static analysis techniques to verify server-side RESTful API code against their OpenAPI specification. We have, however, at the time of writing, found a recent work published a month ago that similarly uses symbolic execution to statically generate REST API specification from the implementation [47]. The work, Respector, uses symbolic execution to generate OpenAPI specification from the implementation. The developer can then compare the generated specification with the original, handwritten OpenAPI specification to find discrepancies. RestAssured, on the other hand, verifies the implementation program directly against the developer-written OpenAPI specification, which solves a verification problem instead of a synthesis problem. We believe the RestAssured workflow is more useful for developers who want to have the choice to either modify the implementation or the specification to make them agree. We think that the two works are complementary and can be used together to ensure the correctness of future RESTful API services.

Looking forward, we believe that the future of RESTful API development will benefit from the use of formal verification techniques, in the same way that other important distributed systems, such as smart blockchain contracts [72], have benefited. Formal verification methods can help developers catch bugs early in the development process, well before they deploy their software in production. Furthermore, it provides a principled way to reason about, even prove, the correctness of their software; which existing testing techniques cannot provide. We do, however, acknowledge issues like usability and performance that need to be addressed before formal verification techniques can be widely adopted in practice. It is our hope that one day, formal verification techniques of nontrivial properties will be as easy to use as linters or type checkers, and that they will be as widely adopted as these tools are today. RestAssured is one attempt to make this vision a reality, by employing static symbolic execution to verify RESTful API services against their OpenAPI specification, a property that no one else has attempted to formally verify before. We are glad to discover that we are not alone in this endeavor and that other researchers [17, 47] are also exploring similar ideas. We hope that RestAssured will inspire future work in the direction, and that the techniques we have developed will be useful in other domains as well.

7.2 Limitations and Future Work

That being said, we also acknowledge the limitations of RestAssured, consisting primarily in our implementation of the RestScript symbolic execution engine.

To begin with, RestScript only captures a subset of the important JavaScript language features,

as discussed in subsection 3.3.1. An important limitation of RestScript is that it does not support exception throwing and handling, which are used very commonly in server-side Express.JS code, to ensure the robustness of the server. This could be a future direction to extend our work, where we discuss potential solutions in subsection 4.2.4, using the same way Microsoft has added Structured Exception Handling (SEH) to C [53, 78]. In addition, we do not support some ES6 features, such as ES modules and `class`es. While we currently offer a workaround to transpile ES6 code to ES5 using Babel [3], we believe that it would be more elegant to support these features directly in our symbolic execution engine.

In our implementation of the symbolic execution engine for RestScript, we made some assumptions to simplify the implementation at the cost of soundness. For example, we interpreted the program as dynamically scoped instead of lexically scoped, which leads to unfaithful modeling of the program's behavior if variable shadowing and closures are used. We model arrays as an array of the exact same content for all elements, which is good enough for the granularity of OpenAPI specification, but makes the analysis not faithful to the actual program, especially if the program manipulates arbitrary elements in the array. We also do not fully support unstructured loops and recursion.

In addition, RestScript chooses to only model a well-behaved subset of JavaScript, which is not always the case in real-world programs. For example, we do not support dynamic re-binding of `this`, which is commonly used in event handling callback functions. Although they are more commonly seen in frontend JavaScript code, they can also be used in backend code. We do not model the full semantics of JavaScript's object prototype system, which allows objects to change their prototypes, or inherited methods and properties, at runtime. We also do not support automatic type casting, which effectively treats all equal operators (`==`) as strict equal operators (`===`). These behaviors are all common sources of frustration and bugs in JavaScript code [7], and we believe that it would be beneficial to support them in the future.

Moreover, in RestAssured's analysis, we make some simplifying assumptions that may not hold true in reality. For example, we assume that all handlers are side effect free, meaning the execution of them will not change the global state of the server program and hence the order of execution does not matter. As a result, RestAssured analyzes each handler in isolation, and is thus unable to model the interactions between different handlers and time dependencies. This is a reasonable assumption for some RESTful methods, such as `GET`, which are supposed to be idempotent. However, for other methods, such as `POST` or `PUT`, this assumption may not hold true. For example, we may imagine a server that provides an admin interface to change its configuration on the fly. Then, the same configuration would be read in by some other handlers and change their behaviors. In our analysis, we treat the two events as independent and reason about all possible values of the configuration during analysis of the other handlers. However, it might be beneficial to recognize these dependencies and interactions in our analysis.

Similarly, RestAssured adopts a very conservative model for interactions with external environments, such as databases, file systems, or network. Currently, we model all of these interactions as uninterpreted functions in Z3, which can take any value. This effectively causes us to lose all information about the output of these interactions, even though we have full information

over what types of values the input can take. A potential future direction is to allow the user to provide inline annotations to specify the behavior of these uninterpreted functions, thus making the analysis more precise. For example, supposing in benchmark `5C` (see subsection 6.4.2), the `username` field is read from a database, the user may want to specify that the `username` field is a string and must have a length between 3 and 10 characters. These annotations will provide more information to the symbolic execution engine, and thus reduce the number of false positives in the analysis.

In conclusion, we make many simplifying assumptions and approximations in the implementation of RestAssured, at the cost of the analysis's soundness. However, we believe that although our analysis is not sound, it is still "soundy", in the sense that it is "mostly sound, with specific, well-identified unsound choices" [59] which we have discussed above. These limitations make it possible for RestAssured to be implemented in the time frame of this thesis, while still providing useful and informative results to an important subset of JavaScript and Express.JS programs. We believe that these limitations are not fundamental to the RestAssured algorithm, and can be addressed in future work to make the analysis more precise and more faithful to the actual program behavior.

7.3 Discussions

Last but not least, I would like to share and discuss some broader insights and lessons that I have learned along the course of this project.

The OpenAPI specification is not expressive enough.

The OpenAPI specification, although named as a specification, is not at all expressive enough to capture all semantics of a RESTful API service. In fact, it is much closer to a schema documentation than a specification. It is not as powerful or expressive enough to capture all the constraints that we want to check. For example, an API designer may want to specify that a login endpoint only returns 200 if the user is authenticated, and 401 otherwise; or for all endpoints to return a 403 if an authentication token is not provided in the HTTP header. Currently, these constraints are not expressible in the OpenAPI specification, and thus cannot be checked by RestAssured. In practice, API designers and developers can only specify these desired semantic properties in plaintext documentation, which provides no principled way to verify them.

Our analysis is only as good as the specification we are given, for "even perfect program verification can only establish that a program meets its specification" [10]. Currently, our analysis technique of symbolic execution overpowers the specification and is limited by the expressiveness of the OpenAPI specification. For example, in benchmark `0B`, where the program produces the output "Hello ${name}" if a `name` is provided in the query string, our analysis is able to tell that the program output will always be of the form

```
Symbol<(str.++ (str.++ "Hello, " name) "!")>
```

given the following path constraints

```
(declare-const name String)
(assert (> (str.len name) 0))
```

However, the OpenAPI specification is only powerful enough to specify that the output is of type `string`, or at the very best some regular expression "`^Hello \w+!$`". It does not allow us to specify that the output is dependent on the input field `name` and must contain part of the input. We hope that future versions of the OpenAPI specification will be more expressive, allowing developers to specify fine-grained constraints on the behavior of their RESTful API services. These advances could prompt the development of more powerful verification tools to verify these constraints automatically.

Do we really need symbolic execution to verify conformity?

Given that the OpenAPI specification is not as expressive, the question naturally arises: do we really need symbolic execution to verify the conformity of the implementation to the current OpenAPI specification? For now, we have chosen to employ symbolic execution as a starting point because it is a powerful and general technique. However, can we achieve the same verification results using more lightweight methods, such as some form of dependent or refinement type checking methods? We do not currently have a definitive answer to this question, but we believe that it is an interesting avenue for future research, given the complicated inter-procedural control flow in Express.JS programs.

We could have used an off-the-shelf symbolic execution engine, or could we?

Looking from hindsight, a majority of the project's complexity comes from building a symbolic execution engine from scratch. We would have saved a lot of time and effort if we had chosen to use an off-the-shelf symbolic execution engine like KLEE [11] or S2E [15]. However, none of these popular engines support JavaScript; and as discussed in section 2.4.3, at the time of writing, all existing symbolic execution engines developed for JavaScript are either not purely static or not available. It seems that building a symbolic execution engine from scratch is most straightforward.

Alternatively, we could have chosen to rewrite the JavaScript/TypeScript code into a more symbolic-execution-friendly language, such as LLVM IR or C code, and then run KLEE or S2E on the rewritten code. This would save us some work in building the symbolic execution engine, but rewriting JavaScript to LLVM IR or C in a semantic-preserving way is similarly a nontrivial task. In addition, we could also choose a more static analysis friendly target language, such as Java, which is also a popular language for back-end RESTful API services. For example, Respector performs symbolic execution on RESTful servers implemented in Java using Spring Boot or Jersey [47].

However, we believe that the effort is not wasted. Building a symbolic execution engine from first principles has given me a deep understanding of the techniques and challenges involved in symbolic execution, and why static symbolic execution engines for JavaScript are not widely available. The many quirks of JavaScript, such as its dynamic typing, object prototypes (which can be changed at runtime), and its various scope and closure rules, make it a challenging language to analyze statically. JavaScript is perhaps a prime example for why formal methods at its current state is hard to be widely adopted in practice. To quote an unnamed researcher specializing in the verification of JavaScript programs, "I have a love-hate relationship with JavaScript", and I can now understand why. RestAssured is a small step towards making formal methods available and usable for JavaScript developers, and we hope that it will inspire future work in this direction.

Bibliography

[1] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, New York, 2001. ISBN 978-0-471-38922-4.

[2] Sacha Ayoun. Gillian-JS is broken · Issue #113 · GillianPlatform/Gillian. GitHub, August 2022. URL https://github.com/GillianPlatform/Gillian/issues/113.

[3] babel. Babel/babel. Babel, May 2024. URL https://github.com/babel/babel.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL https://dl.acm.org/doi/10.1145/3182657.

[5] R. Balzer, N. Goldman, and D. Wile. Informality in Program Specifications. *IEEE Transactions on Software Engineering*, SE-4(2):94–103, March 1978. ISSN 1939-3520. doi: 10.1109/TSE.1978.231480. URL https://ieeexplore.ieee.org/abstract/document/1702503.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[7] Gary Bernhardt. Wat. CodeMash, 2012. URL https://www.destroyallsoftware.com/talks/wat. Last accessed: 2024-04-11.

[8] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, September 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.284. URL http://ieeexplore.ieee.org/document/5233505/.

[9] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. Request for Comments RFC 8259, Internet Engineering Task Force, December 2017. URL https://datatracker.ietf.org/doc/rfc8259.

[10] Frederick P. Brooks Jr. No Silver Bullet—Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987. ISSN 1558-0814. doi: 10.1109/MC.1987.1663532. URL https://ieeexplore.ieee.org/document/1663532.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, December 2008. USENIX Association.

[12] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 270–281, New York, NY, USA, June 2014. Association for Computing Machinery. doi: 10.1145/2594291.2594301. URL https://dl.acm.org/doi/10.1145/2594291.2594301.

[13] Pierre Carbonnelle. PYPL PopularitY of Programming Language index, April 2024. URL https://pypl.github.io/PYPL.html. Last accessed: 2024-04-05.

[14] Andrea Carraro. Toomuchdesign/openapi-ts-json-schema, January 2024. URL https://github.com/toomuchdesign/openapi-ts-json-schema.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1):2:1–2:49, February 2012. ISSN 0734-2071. doi: 10.1145/2110356.2110358. URL https://dl.acm.org/doi/10.1145/2110356.2110358.

[16] Edmund M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer. doi: 10.1007/BFb0058022.

[17] Michael Coblenz, Wentao Guo, Kamatchi Voozhian, and Jeffrey S Foster. A Qualitative Study of REST API Design and Specification Practices. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, Washington, DC, USA, October 2023. IEEE. URL https://www.cs.tufts.edu/~jfoster/papers/vlhcc23.pdf.

[18] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018. URL https://inria.hal.science/hal-01960203/document.

[19] coq. Coq/coq. Coq, March 2024. URL https://github.com/coq/coq.

[20] CppCon. CppCon 2017: Bjarne Stroustrup "Learning and Teaching Modern C++", September 2017. URL https://www.youtube.com/watch?v=fX2W3nNjJIo. Last accessed: 2024-02-26.

[21] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, pages 233–247, Berlin, Heidelberg, 2012. Springer. doi: 10.1007/978-3-642-33826-7_16.

[22] Dan Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note, World Wide Web Consortium, May 2000. URL https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[23] David Frank. The Man Behind the Internet: A chat with worldwide pioneer Vinton Cerf, November 2016. URL https://www.aarp.org/home-family/personal-technology/info-2016/vint-cerf-internet-pioneer-video.html. Last accessed: 2023-11-19.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer. doi: 10.1007/978-3-540-78800-3_24.

[25] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. doi: 10.1007/978-3-319-21401-6_26.

[26] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, SIGDOC '05, pages 68–75, New York, NY, USA, September 2005. Association for Computing Machinery. doi: 10.1145/1085313.1085331. URL https://dl.acm.org/doi/10.1145/1085313.1085331.

[27] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10): 859–866, October 1972. ISSN 0001-0782. doi: 10.1145/355604.361591. URL https://dl.acm.org/doi/10.1145/355604.361591.

[28] Ecma International. ECMA-262 Edition 5.1, the ECMAScript® Language Specification, June 2011. URL https://262.ecma-international.org/5.1/.

[29] Ecma International. ECMA-262 14th Edition, ECMAScript® 2023 language specification, June 2023. URL https://262.ecma-international.org/14.0/.

[30] expressjs. Expressjs/express. OpenJS Foundation, October 2023. URL https://github.com/expressjs/express.

[31] expressjs. Express. npm, March 2024. URL https://www.npmjs.com/package/express. Last accessed: 2024-04-06.

[32] facebook. React. npm, March 2024. URL https://www.npmjs.com/package/react. Last accessed: 2024-04-06.

[33] Manuel Fähndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, Lecture Notes in Computer Science, pages 10–30, Berlin, Heidelberg, 2011. Springer. doi: 10.1007/978-3-642-18070-5_2.

[34] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, March 2010. Association for Computing Machinery. doi: 10.1145/1774088.1774531. URL https://dl.acm.org/doi/10.1145/1774088.1774531.

[35] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Request for Comments RFC 7231, Internet Engineering Task Force, June 2014. URL https://datatracker.ietf.org/doc/rfc7231.

[36] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL https://www.proquest.com/docview/304591392/abstract/2B46E63399B74C63PQ/1.

[37] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, May 2002. Association for Computing Machinery. doi: 10.1145/512529.512558. URL https://dl.acm.org/doi/10.1145/512529.512558.

[38] Robert W Floyd. Assigning Meanings to Programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19. American Mathematical Society, 1967.

[39] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, pages 26–33, New York, NY, USA, November 2002. Association for Computing Machinery. doi: 10.1145/585058.585065. URL https://dl.acm.org/doi/10.1145/585058.585065.

[40] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL):66:1–66:31, January 2019. doi: 10.1145/3290379. URL https://dl.acm.org/doi/10.1145/3290379.

[41] Ned Freed, John C. Klensin, and Tony Hansen. Media Type Specifications and Registration Procedures. Request for Comments RFC 6838, Internet Engineering Task Force, January 2013. URL https://datatracker.ietf.org/doc/rfc6838.

[42] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation - PLDI '91*, pages 268–277, Toronto, Ontario, Canada, 1991. ACM Press. doi: 10.1145/113445.113468. URL http://portal.acm.org/citation.cfm?doid=113445.113468.

[43] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for RESTful APIs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 122–136, New York, NY, USA, June 2022. Association for Computing Machinery. doi: 10.1145/3519939.3523450. URL https://dl.acm.org/doi/10.1145/3519939.3523450.

[44] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving Semantics-Aware Fuzzers from Web API Schemas, December 2021. URL http://arxiv.org/abs/2112.10328. An extensive and more detailed preprint for [45].

[45] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 345–346, May 2022. doi: 10.1145/3510454.3528637. URL https://ieeexplore.ieee.org/document/9793781.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://dl.acm.org/doi/10.1145/363235.363259.

[47] Ruikai Huang, Manish Motwani, Idel Martinez, and Alessandro Orso. Generating REST API Specifications through Static Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, New York, NY, USA, April 2024. Association for Computing Machinery. doi: 10.1145/3597503.3639137. URL https://dl.acm.org/doi/10.1145/3597503.3639137.

[48] James Kibirige. Are Code Contracts going to be supported in .NET Core going forwards? · Issue #6361 · dotnet/docs, July 2018. URL https://github.com/dotnet/docs/issues/6361.

[49] Janet Wagner. Understanding the Differences Between API Documentation, Specifications, and Definitions, n.d. URL https://swagger.io/resources/articles/difference-between-api-documentation-specification/. Last accessed: 2023-11-30.

[50] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, October 2020. doi: 10.1109/ICST46399.2020.00023.

[51] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, January 1984. ISSN 0010-4620. doi: 10.1093/comjnl/27.2.97. URL https://doi.org/10.1093/comjnl/27.2.97.

[52] Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Classroom Notes Stanford University, March 1977. URL https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf.

[53] Vishal Kochhar. How a C++ compiler implements exception handling, April 2002. URL https://www.codeproject.com/Articles/2126/How-a-C-compiler-implements-exception-handling.

[54] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002. URL https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/.

[55] Axel Van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, Limerick Ireland, May 2000. ACM. doi: 10.1145/336512.336546.

[56] Baudouin Le Charlier and Pierre Flener. Specifications are necessarily informal or: Some more myths of formal methods. *Journal of Systems and Software*, 40(3):275–296, March 1998. ISSN 01641212. doi: 10.1016/S0164-1212(98)00172-1. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121298001721.

[57] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer. doi: 10.1007/978-3-642-17511-4_20.

[58] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys*, 48(3):33:1–33:41, December 2015. ISSN 0360-0300. doi: 10.1145/2831270. URL https://dl.acm.org/doi/10.1145/2831270.

[59] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58 (2):44–46, January 2015. ISSN 0001-0782, 1557-7317. doi: 10.1145/2644805. URL https://dl.acm.org/doi/10.1145/2644805.

[60] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 196–199, New York, NY, USA, July 2017. Association for Computing Machinery. doi: 10.1145/3092282.3092295. URL https://dl.acm.org/doi/10.1145/3092282.3092295.

[61] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 827–850, Cham, 2021. Springer International Publishing. doi: 10.1007/978-3-030-81688-9_38.

[62] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *2010 Eighth IEEE European Conference on Web Services*, pages 107–114, December 2010. doi: 10.1109/ECOWS.2010.9. URL https://ieeexplore.ieee.org/abstract/document/5693251.

[63] MargeKh. Z3-solver Z3.String() is not a function angular implementation · Issue #6957 · Z3Prover/z3, October 2023. URL https://github.com/Z3Prover/z3/issues/6957.

[64] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of RESTful APIs: Promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 408–420, Singapore Singapore, November 2022. ACM. doi: 10.1145/3540250.3549144.

[65] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. Pearson Education, 1996. ISBN 978-1-55615-900-8.

[66] S.G. McLellan, A.W. Roesler, J.T. Tempest, and C.I. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, May 1998. ISSN 1937-4194. doi: 10.1109/52.676963. URL https://ieeexplore.ieee.org/abstract/document/676963.

[67] Microsoft. Microsoft/TypeScript, October 2023. URL https://github.com/microsoft/TypeScript.

[68] Darrel Miller, Jeremy Whitlock, Marsh Gardiner, Mike Ralphson, Ron Ratovsky, and Uri Sarid. OpenAPI Specification v3.1.0. Technical report, OpenAPI Initiative, February 2021. URL https://spec.openapis.org/oas/v3.1.0. Last accessed: 2023-12-01.

[69] F.L. Morris and C.B. Jones. An Early Program Proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, April 1984. ISSN 1058-6180. doi: 10.1109/MAHC.1984.10017. URL http://ieeexplore.ieee.org/document/4640518/.

[70] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6 (4):310–316, July 1966. ISSN 1572-9125. doi: 10.1007/BF01966091. URL https://doi.org/10.1007/BF01966091.

[71] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing*, 14(4):957–970, July 2021. ISSN 1939-1374. doi: 10.1109/TSC.2018.2847344. URL https://ieeexplore.ieee.org/document/8385157/.

[72] nhsz. Formal verification of smart contracts, December 2023. URL https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/. Last accessed: 2024-04-03.

[73] Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, editors. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002. doi: 10.1007/3-540-45949-9. URL http://link.springer.com/10.1007/3-540-45949-9.

[74] OpenAPI Initiative. API Endpoints, 2023. URL https://learn.openapis.org/specification/paths.html. Last accessed: 2023-10-30.

[75] Pallets. Pallets/flask. Pallets, April 2024. URL https://github.com/pallets/flask.

[76] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356, New York, NY, USA, June 2015. Association for Computing Machinery. doi: 10.1145/2737924.2737991. URL https://dl.acm.org/doi/10.1145/2737924.2737991.

[77] David Lorge Parnas. Really Rethinking 'Formal Methods'. *Computer*, 43(1):28–34, January 2010. ISSN 1558-0814. doi: 10.1109/MC.2010.22. URL https://ieeexplore.ieee.org/document/5398780.

[78] Matt Pietrek. A Crash Course on the Depths of Win32 Structured Exception Handling. *Microsoft Systems Journal*, (January 1997), 1997. ISSN 0889-9932. URL https://web.archive.org/web/20170305044131/http://www.microsoft.com/msj/0197/exception/exception.aspx.

[79] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pages 46–57, October 1977. doi: 10.1109/SFCS.1977.32. URL https://ieeexplore.ieee.org/abstract/document/4567924.

[80] Postman, Inc. 2023 State of the API Report. Technical report, Postman, Inc., 2023. URL https://voyager.postman.com/pdf/2023-state-of-the-api-report-postman.pdf. Last accessed: 2023-11-29.

[81] Drew Powers. Drwpow/openapi-typescript, April 2024. URL https://github.com/drwpow/openapi-typescript.

[82] pplonski86. Why Don't People Use Formal Methods? | Hacker News, January 2019. URL https://news.ycombinator.com/item?id=18965274. Last accessed: 2024-04-27.

[83] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935, June 2012. doi: 10.1109/ICSE.2012.6227127.

[84] Mike Ralphson. APIs-guru/openapi-directory: 🌐 Wikipedia for Web APIs. Directory of REST API definitions in OpenAPI 2.0/3.x format, March 2024. URL https://github.com/APIs-guru/openapi-directory.

[85] RapidAPI. State of APIs 2022 | Rapid Developer Survey Results, 2022. URL https://stateofapis.com/. Last accessed: 2023-11-29.

[86] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: Meeting developers where they are, October 2020. URL http://arxiv.org/abs/2010.16345.

[87] Leonard Richardson and Sam Ruby. RESTful Web Services. "O'Reilly Media, Inc.", December 2008. ISBN 978-0-596-55460-6.

[88] rollup. Rollup/rollup. Rollup, April 2024. URL https://github.com/rollup/rollup.

[89] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP '18, pages 1–14, New York, NY, USA, September 2018. Association for Computing Machinery. doi: 10.1145/3236950.3236956. URL https://dl.acm.org/doi/10.1145/3236950.3236956.

[90] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In 2010 IEEE Symposium on Security and Privacy, pages 513–528, May 2010. doi: 10.1109/SP.2010.38. URL https://ieeexplore.ieee.org/abstract/document/5504700.

[91] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, 30(5):263–272, September 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081750. URL https://dl.acm.org/doi/10.1145/1095430.1081750.

[92] David Sherret. Dsherret/ts-ast-viewer, April 2024. URL https://github.com/dsherret/ts-ast-viewer.

[93] Ye Shu. Deletes expired identity token from local storage · Issue #199 · WilliamsStudentsOnline/wso-react, April 2022. URL https://github.com/WilliamsStudentsOnline/wso-react/issues/199.

[94] SmartBear Software. The All-In-One Automated API Testing Platform | ReadyAPI, 2023. URL https://smartbear.com/product/ready-api/. Last accessed: 2023-12-08.

[95] SmartBear Software. State of Software Quality | API. Technical report, SmartBear Software, 2023. URL https://assets.smartbear.com/m/5276eb5353c29969/original/State-of-Quality-API-2023-Report.pdf. Last accessed: 2023-11-29.

[96] Confucius' Students. Confucian Analects. In James Legge, editor, The Chinese Classics, volume 1, pages 137–354. At the author's, Hong Kong, 1861. URL https://archive.org/details/chineseclassics01mencgoog/page/296/.

[97] Surnet GmbH. Surnet/swagger-jsdoc. Surnet, November 2023. URL https://github.com/Surnet/swagger-jsdoc. Last accessed: 2023-11-30.

[98] Swagger. Data Types. URL https://swagger.io/docs/specification/data-models/data-types/. Last accessed: 2024-04-29.

[99] Swagger. Swagger-api/swagger-codegen. Swagger, December 2023. URL https://github.com/swagger-api/swagger-codegen. Last accessed: 2023-12-08.

[100] Swagger. Swagger-api/swagger-core. Swagger, November 2023. URL https://github.com/swagger-api/swagger-core. Last accessed: 2023-11-30.

[101] Swagger. Swagger-api/swagger-ui. Swagger, November 2023. URL https://github.com/swagger-api/swagger-ui. Last accessed: 2023-11-30.

[102] Scott R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing - Volume 2, CASCON '93, pages 1083–1090, Toronto, Ontario, Canada, October 1993. IBM Press. URL https://dl.acm.org/doi/10.5555/962367.962412.

[103] TIOBE Software BV. TIOBE Index. TIOBE, March 2024. URL https://www.tiobe.com/tiobe-index/. Last accessed: 2024-04-05.

[104] A. Turing. Checking a large routine. In Report of a Conference on High Speed Automatic Calculating Machines, pages 67–69, Cambridge, June 1949. University Math Lab. ISBN 978-0-262-23136-7.

[105] John Vilk and Emery D. Berger. DOPPIO: Breaking the Browser Language Barrier. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 508–518, Edinburgh United Kingdom, June 2014. ACM. doi: 10.1145/2594291.2594293. URL https://dl.acm.org/doi/10.1145/2594291.2594293.

[106] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. Statically Checking Web API Requests in JavaScript. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 244–254, May 2017. doi: 10.1109/ICSE.2017.30.

[107] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. ACM Computing Surveys, 41(4):19:1–19:36, October 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592436. URL https://dl.acm.org/doi/10.1145/1592434.1592436.

[108] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. JSON Schema: A Media Type for Describing JSON Documents. Internet Draft draft-bhutton-json-schema-01, Internet Engineering Task Force, June 2022. URL https://datatracker.ietf.org/doc/draft-bhutton-json-schema-01.

[109] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, pages 214–227, New York, NY, USA, January 1999. Association for Computing Machinery. doi: 10.1145/292540.292560. URL https://dl.acm.org/doi/10.1145/292540.292560.

[110] Jinqiu Yang, Erik Wittern, Annie T. T. Ying, Julian Dolby, and Lin Tan. Towards extracting web API specifications from documentation. In Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, pages 454–464, New York, NY, USA, May 2018. Association for Computing Machinery. doi: 10.1145/3196398.3196411. URL https://dl.acm.org/doi/10.1145/3196398.3196411.

Appendices

Appendix A

Example OpenAPI Specification

You can find an OpenAPI Specification in Listing A.1. This specification is generated from the function comments in the motivating example (Listing 1.1) with [97].

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Login API",
    "version": "1.0.0"
  },
  "paths": {
    "/login": {
      "post": {
        "summary": "Login to the application.",
        "requestBody": {
          "required": true,
          "content": {
            "application/json": {
              "schema": {
                "type": "object",
                "properties": {
                  "username": {
                    "type": "string"
                  },
                  "password": {
                    "type": "string"
                  }
                }
              }
            }
          }
        },
        "responses": {
          "200": {
            "description": "Login successful.",
            "content": {
              "application/json": {
                "schema": {
                  "type": "object",
```

```
                  "properties": {
                    "message": {
                      "type": "string",
                      "description": "A success message."
                    },
                    "username": {
                      "type": "string",
                      "description": "Username."
                    }
                  }
                }
              }
            }
          },
          "401": {
            "description": "Login failed",
            "content": {
              "application/json": {
                "schema": {
                  "type": "object",
                  "properties": {
                    "message": {
                      "type": "string",
                      "description": "An error message."
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  },
  "components": {},
  "tags": []
}
```

Listing A.1: Generated OpenAPI Specification from Motivating Example Listing 1.1. For explanation of the fields, see subsection 2.3.5

Appendix B

Formal Syntax for RestScript

```
SourceFile ::= Statements
Statements ::= Statement
             | Statement ";"
             | Statement ";" Statements
             | Statement "\n" Statements  // semicolon is not required at EOL

Statement ::= Block
            | VariableStatement
            | FunctionDeclaration
            | ExpressionsStatement
            | IfStatement
            | WhileStatement
            | ForStatement
            | ReturnStatement
            | BreakStatement
            | ContinueStatement

Block ::= "{" Statements "}"

VariableStatement ::= "var" VariableDeclarations
                    | "let" VariableDeclarations
                    | "const" VariableDeclarations
VariableDeclarations ::= VariableDeclaration
                       | VariableDeclaration "," VariableDeclarations
VariableDeclaration  ::= Name
                       | Name "=" Expression
                       | Name ":" TypeName
                       | Name ":" TypeName "=" Expression
TypeName ::= "string"
           | "number"
           | "boolean"
           | "object"
           | "any"
           | "void"
           | "null"
           | "undefined"
           | TypeName "[]"
           | <... other fancy TypeScript types ...>  // e.g. union types, not supported

```

```
40
41 FunctionDeclaration ::= "function" Identifier "(" ParameterList ")" Block
42 Parameters ::= "(" ParameterList ")"
43 ParameterList ::= Identifier "," ParameterList
44                  | Identifier ":" TypeName "," ParameterList
45                  | Identifier
46                  | Identifier ":" TypeName
47
48 ExpressionsStatement ::= Expression
49 Expression := BinaryExpression
50             | UnaryExpression
51             | ParenthesizedExpression
52             | Literal
53             | Identifier
54             | ArrowFunction
55             | CallExpression
56             | NewExpression
57             | PropertyAccessExpression
58             | ArrayElementAccessExpression
59
60 BinaryExpression ::= Expression "=" Expression
61                     // syntactic sugar assignment expressions (e.g. +=) are ignored
62
63                     // arithmetic expressions
64                     | Expression "+" Expression
65                     | Expression "-" Expression
66                     | Expression "*" Expression
67                     | Expression "/" Expression
68                     | Expression "%" Expression
69                     | Expression "**" Expression
70                     | Expression "<<" Expression
71                     | Expression ">>" Expression
72                     | Expression ">>>" Expression
73
74                     // bitwise expressions
75                     | Expression "&" Expression
76                     | Expression "|" Expression
77                     | Expression "&&" Expression
78                     | Expression "||" Expression
79                     | Expression "??" Expression
80
81                     // comparison expressions
82                     | Expression "==" Expression
83                     | Expression "!=" Expression
84                     | Expression "===" Expression
85                     | Expression "!==" Expression
86                     | Expression ">" Expression
87                     | Expression ">=" Expression
88                     | Expression "<" Expression
89                     | Expression "<=" Expression
90
91                     // special expressions
92                     | Expression "in" Expression
93                     | Expression "instanceof" Expression
94
95 UnaryExpression ::= "!" Expression
96                    | "-" Expression
97                    | "+" Expression
```

```
                  | "~" Expression
                  | "++" Expression
                  | "--" Expression
                  | Expression "++"
                  | Expression "--"
                  | "typeof" Expression
                  | "void" Expression
                  | "delete" Expression

ParenthesizedExpression ::= "(" Expression ")"

Literal ::= "null"
          | "true"
          | "false"
          | "this"
          | NumericLiteral
          | StringLiteral
          | ObjectLiteral
          | ArrayLiteral

NumericLiteral ::= regex("[0-9]+")
StringLiteral ::= regex("\"([^\\\"]|\\\"|\\n)*\"")  // other escaped characters can
                                                    // be added just like "\n"

ObjectLiteral ::= "{" PropertyNameAndValueList "}"
PropertyNameAndValueList ::= PropertyNameAndValue
                           | PropertyNameAndValue "," PropertyNameAndValueList
PropertyNameAndValue ::= Identifier ":" Expression

ArrayLiteral ::= "[" ElementList "]"
ElementList ::= Expression
              | Expression "," ElementList

Identifier ::= regex("[a-zA-Z_][a-zA-Z0-9_]*")  // cannot start with a number

ArrowFunction ::= Parameters "=>" Expression

CallExpression ::= Expression "(" ArgumentList ")"
ArgumentList ::= Expression[","]

NewExpression ::= "new" Expression

PropertyAccessExpression ::= Expression "." Identifier

ArrayElementAccessExpression ::= Expression "[" Expression "]"

IfStatement ::= "if" "(" Expression ")" Block
              | "if" "(" Expression ")" Block "else" Block

WhileStatement ::= "while" "(" Expression ")" Block

ForStatement ::= "for" "(" Expression ";" Expression ";" Expression ")" Block
              | "for" "(" VariableDeclaration ";" Expression ";" Expression ")" Block
              | "for" "(" Expression "in" Expression ")" Block
              | "for" "(" VariableDeclaration "in" Expression ")" Block

ReturnStatement ::= "return" Expression
                  | "return"
```

```
156
157 BreakStatement ::= "break"  // break to label not supported
158
159 ContinueStatement ::= "continue"  // continue to label not supported
```

Listing B.1: A formal definition of RestScript using BNF

Appendix C

Select Synthetic Benchmarks for Evaluating RestAssured

C.1 Full Sample Program Output of RestAssured

This is the full output produced by running RestAssured on the motivating example in subsection 1.1.1 and the OpenAPI specification in Appendix A.

```
yarn run v1.22.19
$ npm run build

> se-engine@1.0.0 build
> tsc -p tsconfig.json

$ node --enable-source-maps build/index.js ../examples/0c-login
[INFO] Reading API spec: ../examples/0c-login/swagger.json
[openapi-ts-json-schema] ✓ JSON schema models generated at ./src/generated/
[INFO] API spec types generated
[INFO] Reading file: ../examples/0c-login/server.js
[INFO] Symbolically executing the program
[INFO] Symbolic execution complete. We now have a symbolic model for the Express.JS
    ↪ application.
[INFO] Found 1 server models in all branches of the program.

[INFO] Executing server instance 0

[INFO] Generating symbolic requests for the server...
[INFO] Injecting symbolic request with path: /login
[INFO] Injecting symbolic request with method: post

[INFO] Server received request: POST /login
[INFO] Executing middleware: express.json()
[INFO] Found matching route: /login
[INFO] Executing handler: function-anonymous-pos-1387-1851

[INFO] EOF reached.

[INFO] Multiple paths found due to branching. Continue execution.
```

```
30
31 [INFO] Multiple paths found due to branching. Continue execution.
32 [INFO] Symbolical execution complete. 3 states found. We will now check them for
       ↪ conformance with specifications.
33
34
35 [INFO] checking response state conformance with Path Constraints (HTTP 400):
36 (declare-const username String)
37 (declare-const password String)
38 (assert (or (= username "") (= password "")))
39 [INFO] SAT:
40 (define-fun username () String
41   "")
42 (define-fun password () String
43   "")
44 [ERROR] response status code 400 not specified in swagger API spec. VIOLATE
45 [INFO] expected (valid) codes:
46 [ 200, 401 ]
47 [INFO] actual response:
48 [INFO] express response: 400
49 [INFO] express response header:
50 Map(1) { 'Content-Type' => 'application/json' }
51 [INFO] express response body:
52 Value {
53   type: 'object',
54   name: undefined,
55   expression: Expression {
56     value: {
57       message: Value {
58         type: 'string',
59         name: undefined,
60         expression: Expression { value: 'Malformed request' },
61         parent: undefined
62       }
63     }
64   },
65   parent: undefined
66 }
67 [INFO] State 1: VIOLATE.
68
69 [INFO] checking response state conformance with Path Constraints (HTTP 200):
70 (declare-const username String)
71 (declare-const password String)
72 (assert (not (or (= username "") (= password ""))))
73 (declare-fun users.find-pos-1610-1692 () String)
74 (assert (> (str.len users.find-pos-1610-1692) 0))
75 [INFO] SAT:
76 (define-fun password () String
77   "C")
78 (define-fun username () String
79   "B")
80 (define-fun users.find-pos-1610-1692 () String
81   "A")
82 [INFO] response matches with code "200" and symbolic object body with the following fields:
83 {
84   message: Value {
85     type: 'string',
86     name: undefined,
```

```
    expression: Expression { value: 'Login successful' },
    parent: undefined
  },
  username: Value {
    type: 'symbol',
    name: 'username',
    expression: Expression { value: { type: 'string', smt: 'username' } },
    parent: undefined
  }
}
[INFO] State 2: CONFORM.

[INFO] checking response state conformance with Path Constraints (HTTP 401):
(declare-const username String)
(declare-const password String)
(assert (not (or (= username "") (= password ""))))
(declare-fun users.find-pos-1610-1692 () String)
(assert (not (> (str.len users.find-pos-1610-1692) 0)))
[INFO] SAT:
(define-fun password () String
  "B")
(define-fun username () String
  "A")
(define-fun users.find-pos-1610-1692 () String
  "")
[INFO] response matches with code "401" and symbolic object body with the following fields:
{
  message: Value {
    type: 'string',
    name: undefined,
    expression: Expression { value: 'Login failed' },
    parent: undefined
  }
}
[INFO] State 3: CONFORM.

[INFO] ANALYSIS for endpoint POST /login:
   Out of 3 states, 3 are SATisfiable/reachable. In which 2 CONFORMs to specification.

[INFO] Symbolical execution complete

[INFO] FINISHED. VERDICT FOR THE SERVER: VIOLATE
  Out of 1 total paths and 1 endpoints specified, 0 are conforming to the API spec.
  The symbolic execution produced 3 states, of which 3 are SATisfiable/reachable and 2 are
    ↪ conforming to the API spec.
[INFO] MESSAGES:
[POST /login] VIOLATE: Response status code 400 not specified in swagger API spec.

Done in 2.90s.
```

Listing C.1: Full program output of running RESTASSURED on the motivating example

C.2 Program to Generate Arbitrary Number of Branches

To simulate running RestAssured on programs with different numbers of branches, we created a simple script that is able to take a template, and create a program with arbitrary number of branches. The output of the program is an Express.JS program that looks like the following. To add more branches, the script will repeat the code block from L38 to L42 and increment the number in the condition, up to the specified number of branches minus one, given the existence of the `else` branch.

```
1  const express = require("express");
2
3  const app = express();
4  const port = 3000;
5
6  app.use(express.json());
7
8  /**
9   * @openapi
10  * /:
11  *   post:
12  *     summary: Responds with "Hello World."
13  *     requestBody:
14  *       required: false
15  *       content:
16  *         application/json:
17  *          schema:
18  *           type: object
19  *           properties:
20  *            number:
21  *              type: number
22  *     responses:
23  *       200:
24  *         description: Successful response with the same text
25  *         content:
26  *           text/plain:
27  *             example: 0
28  */
29 app.post("/", (req, res) => {
30   if (req.body.number < 1) {
31     res.send(req.body.number);
32     return;
33   }
34   else if (req.body.number < 2) {
35     res.send(req.body.number);
36     return;
37   }
38 // EXAMPLE: the next branch to be added is:
39 //   else if (req.body.number < 3) {
40 //     res.send(req.body.number);
41 //     return;
42 //   }
43   else {
44     res.send(req.body.number);
45     return;
46   }
47 });
```

```

app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```

Listing C.2: Express.JS program with arbitrary number of branches

C.3 Benchmark 5C: Unsatisfied Refinement Predicate

The source code for the benchmark 5C is shown in Listing C.3.

```
const express = require("express");

const app = express();
const port = 3000;

app.use(express.json());

/**
 * @openapi
 * /login:
 *   post:
 *     summary: Login to the application.
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               username:
 *                 type: string
 *               password:
 *                 type: string
 *     responses:
 *       200:
 *         description: Login successful.
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 message:
 *                   type: string
 *                   description: A success message.
 *                 username:
 *                   type: string
 *                   description: Username.
 *                   minLength: 3
 *                   maxLength: 10
 *       400:
 *         description: Malformed request
 *         content:
 *           application/json:
 *             schema:
 *               type: object
```

```
 *               properties:
 *                 message:
 *                   type: string
 *                   description: An error message.
 *       401:
 *         description: Login failed
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 message:
 *                   type: string
 *                   description: An error message.
 */
app.post("/login", (req, res) => {
  const { username, password } = req.body;

  if (!username || !password) {
    res.status(400).json({ message: "Malformed request" });
    return;
  } else {
    if (password === "my_super_secret_password") {
      res.json({ message: "Login successful", username });
    } else {
      res.status(401).json({ message: "Login failed" });
    }
  }
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Listing C.3: Source code for benchmark `5C`

Appendix D

Symbols Used in This Thesis

Symbol	Description
N	Field Name, e.g. `username`, `password`, `message`, ….
T	An OpenAPI Type. See Definition 3.1.
$Q = N \rightharpoonup T$	Query Parameters, an array of name-type pairs.
$H = N \rightharpoonup T$	HTTP Headers, an array of name-type pairs.
$\mathcal{M}$	MIME media type, e.g. `application/json`, `text/plain`, ….
$B = \mathcal{M} \rightharpoonup T$	HTTP Body, either an array of name-type pairs or a single typed value.
$R = (Q, H, B)$	Request Schema. See Definition 3.2.
C	HTTP Response Status Code, e.g. `200`, `400`, `401`, ….
$P = (C, H, B)$	Response Schema. See Definition 3.3.
$\mathcal{H} : R \times P$	Endpoint Handler Function type definition.
M	HTTP Method, e.g. `GET`, `POST`, ….
U	URI Path, e.g. `/login`, may contain path parameters.
$\mathcal{E} = (M, U, \mathcal{H})$	RESTful API Endpoint. See Definition 3.4.
$\mathcal{S} = \overline{\mathcal{E}}$	RESTful API Specification. See Definition 3.5.

Table D.1: Table of symbols for formalizing OpenAPI specification

Symbol	Description
$R' = (Q, H, B)$	HTTP Request space accepted by the server.
$P' = (C, H, B)$	HTTP Response space produced by the server.
$\mathcal{H}' : R' \to P'$	Handler Function Implementation.
$\mathcal{E}' = (M, U, \mathcal{H}')$	RESTful API Endpoint Implementation.
$\mathcal{S}' = \overline{\mathcal{E}'}$	RESTful API Server Implementation. See Definition 3.6.

Table D.2: Table of symbols for formalizing RESTful server implementation