Mälardalen University Doctoral Dissertation 412

Exploring API Behaviours by Example Generation

Stefan Karlsson

Mälardalen University

Mälardalen University Press Dissertations
No. 412

EXPLORING API BEHAVIOURS BY EXAMPLE GENERATION

Stefan Karlsson

2024

School of Innovation, Design and Engineering

ISBN 978-91-7485-654-5
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 412

EXPLORING API BEHAVIOURS BY EXAMPLE GENERATION

Stefan Karlsson

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
tisdagen den 10 september 2024, 13.15 i Zeta, Mälardalens universitet, Västerås.

Fakultetsopponent: Associate Professor Mike
Papadakis, University of Luxembourg, Luxembourg

Akademin för innovation, design och teknik

Abstract

Understanding the behaviour of complex software-intensive systems is a hard task. For developers of such systems, understanding the actual behaviours is critical in order to successfully create, extend, and maintain them.

The goal of the work in this thesis is to support explorations of the behaviour of software systems through their APIs. We fulfil this goal by generating examples of behaviours the system exhibits. An example is expressed as a sequence of API operations---with parameters, if required---that conforms to a specific behaviour.

Examples of behaviours, such as sequences of operations performed on the system, have been shown to be a good way to further the understanding of software systems for both end users and developers. However, manually creating examples requires effort. In addition, manually created examples only contain what a human can imagine---which might miss important cases, such as unintended behaviours.

The main proposed approach in this thesis is to support users in exploring the behaviour of their software system by automatically generating examples of actual behaviour. By only interacting with the system by the exposed API, we assess the behaviours as exposed to an end user of the API. The input to the approach is a set of API operations and schema of operation parameters. Sequences of operations are generated containing these provided operations. The observed responses from executing the generated sequences are used to assess if the API show an example of a sought behaviour. Found examples go through a shrinking process---trying to find a more minimal sequence showing the same behaviour---and are then reported to the user of the approach.

The approach is capable of both generating examples of faults in the system and of generating examples of general behaviours. We show evidence of this through multiple evaluations. We have evaluated the fault-finding capabilities by generating examples producing fault-indicating error codes and showing how the configuration of generators affects the interaction with the system. In addition, we evaluate the capability of the approach to generate relevant examples, both in the general API case and in the specific case of REST APIs. By conducting multiple focus group sessions, we conclude that the examples of behaviours produced by the approach indeed aid industry practitioners. The generated examples are deemed relevant for use cases such as testing, documenting, and understanding the behaviour of the system.

ISBN 978-91-7485-654-5
ISSN 1651-4238

Software does not run in a magic fairy aether powered by the fevered dreams of CS PhDs.

- Mike Acton, Data-Oriented Design Principle

Abstract

Understanding the behaviour of complex software-intensive systems is a hard task. For developers of such systems, understanding the actual behaviours is critical in order to successfully create, extend, and maintain them.

The goal of the work in this thesis is to support explorations of the behaviour of software systems through their APIs. We fulfil this goal by generating examples of behaviours the system exhibits. An example is expressed as a sequence of API operations—with parameters, if required—that conforms to a specific behaviour.

Examples of behaviours, such as sequences of operations performed on the system, have been shown to be a good way to further the understanding of software systems for both end users and developers. However, manually creating examples requires effort. In addition, manually created examples only contain what a human can imagine—which might miss important cases, such as unintended behaviours.

The main proposed approach in this thesis is to support users in exploring the behaviour of their software system by automatically generating examples of actual behaviour. By only interacting with the system by the exposed API, we assess the behaviours as exposed to an end user of the API. The input to the approach is a set of API operations and schema of operation parameters. Sequences of operations are generated containing these provided operations. The observed responses from executing the generated sequences are used to assess if the API show an example of a sought behaviour. Found examples go through a shrinking process—trying to find a more minimal sequence showing the same behaviour—and are then reported to the user of the approach.

The approach is capable of both generating examples of faults in the system and of generating examples of general behaviours. We show evidence of this through multiple evaluations. We have evaluated the fault-finding capabilities by generating examples producing fault-indicating error codes and showing how the configuration of generators affects the interaction with the system. In addition, we evaluate the capability of the approach to generate

relevant examples, both in the general API case and in the specific case of REST APIs. By conducting multiple focus group sessions, we conclude that the examples of behaviours produced by the approach indeed aid industry practitioners. The generated examples are deemed relevant for use cases such as testing, documenting, and understanding the behaviour of the system.

Sammanfattning

Att bygga förståelse för beteendet av komplexa mjukvaruintensivasystem är en svår uppgift. För utvecklare av dessa system är en förståelse för det faktiska beteendet kritiskt för att framgångsrikt skapa, utöka och underhålla dem.

Målet för arbetet i denna avhandling är att stödja utforskandet av beteendet hos mjukvarusystem genom deras APIer. Vi uppfyller detta mål genom att generera exempel av beteenden som systemet uppvisar. Ett exempel uttrycks som en sekvens av API operationer–med parametrar, om de krävs–som överensstämmer med ett specifikt beteende.

Exempel av beteenden, så som sekvenser av operationer utförda på systemet, har visat sig vara ett bra sätt att ytterligare bygga förståelse av ett mjukvarusystem, både för slutanvändare och utvecklare. Dock så krävs en insats för att manuellt skapa exempel. Dessutom, manuellt skapade exempel innehåller bara vad en människa kan föreställa sig, vilket kan missa viktiga fall, så som oavsiktliga beteenden.

Det huvudsakliga föreslagna angreppsättet i denna avhandling är att stödja användare att utforska beteendet av deras mjukvarusystem genom att automatiskt generera exempel av faktiskt beteende. Genom att endast interagera med systemet genom det exponerade API:et, utvärderar vi beteendet så som det är exponerat för en slutanvändare av API:et. Indatat till den föreslagna metoden är ett set av API operationer och ett schema över operationernas parametrar. Sekvenser av operationer genereras, innehållandes de angivna operationerna. De observerade svaren av att exekvera de genererade sekvenserna används för att utvärdera om API:et visar exempel på ett sökt beteende. Konstaterade exempel genomgår en krympandeprocess, ett försök att hitta en minimal sekvens som visar samma beteende, och rapporteras sedan till användaren av metoden.

Metoden är kapabel att både generera exempel av fel i systemet och att generera exempel av generella beteenden. Vi belägger detta genom flera utvärderingar. Vi har utvärderat förmågan att hitta fel genom att generera exempel som producerar bugg-indikerande felkoder och visar hur

konfigurationen av generatorer påverkar interaktionen med systemet. Utöver detta, utvärderar vi metodens förmåga att generera *relevanta* exempel, både i det generella API fallet och i det specifika fallet av REST API:er. Genom att genomföra flera fokusgruppssessioner, slutleder vi att beteendeexemplen som produceras av metoden hjälper industriutövare. De genererade exemplen bedöms som relevanta i användningsområden så som testning, dokumentering och för att förstå beteendet av systemet.

Acknowledgments

Thank you to everyone who has supported and inspired me, both emotionally and intellectually.

Thanks to the many times forgotten, open source contributors who put their time and effort into making software serving as a foundation for much research.

Thanks to Daniel, Adnan, and Robbert, my supervisors, for giving me a lot of freedom while teaching me the ways of science.

Finally, a big thanks to John Hughes for agreeing to collaborate with an unknown Ph.D. student. As a practitioner I have been a long-time fan of Johns' work such as co-inventing Property-based testing, all the amazing industry applications of Property-based testing, and his impact on functional programming. Working with John was one of the highlights of this thesis work, and his generosity with his wealth of experience and knowledge is inspiring.

Stefan Karlsson
Sala, 2023

List of Publications

Papers included in thesis[1]

Paper A: S. Karlsson, A. Čaušević, and D. Sundmark. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020. [56].

Paper B: S. Karlsson, A. Čaušević, and D. Sundmark. Automatic Property-based Testing of GraphQL APIs. In *International Conference on Automation of Software Test (AST)*. IEEE/ACM, 2021. [57].

Paper C: S. Karlsson, J. Hughes, R. Jongeling, A. Čaušević, and D. Sundmark. Exploring API Behaviours Through Generated Examples. In *Software Quality Journal (SQJ)*. Springer, 2024. [55].

Paper D: S. Karlsson, R. Jongeling, A. Čaušević, and D. Sundmark. Exploring Behaviours of RESTful APIs in an Industrial Setting. *In submission*. ArXiv pre-print, 2023. [93].

[1]The included papers have been reformatted to comply with the thesis layout.

Publications not included in thesis

Paper X: S. Karlsson. Exploratory Test Agents for Stateful Software Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019. [54].

Paper Y: S. Karlsson, A. Čaušević, D. Sundmark, and M. Larsson. Model-based Automated Testing of Mobile Applications: An Industrial Case Study. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021. [58].

Licentiate Thesis: S. Karlsson. Towards Augmented Exploratory Testing. Mälardalen University, 2021. [59].

Contents

Part I

Thesis

Chapter 1

Introduction

Software-intensive systems are everywhere in today's society. We have become dependent on software for our infrastructure, such as the production and supply of electricity, food, and water, our social and work interactions, and much of our entertainment. Thus, understanding the behaviours of software-intensive systems is critical. This understanding is both in the positive—does the provider of the software introduce the correct sought behaviour?—and in the negative—are any unwanted behaviours included in the software?

Experience from industry informs us that developing, maintaining, and extending complex software-intensive systems has many challenges. One such challenge is to understand the exposed behaviour of the system. When developers create or change complex software systems, it is essential—but hard—to understand the exposed behaviours of the system.

A software system might be capable of more behaviours than are exposed to an end user. The source of observable information to understand the actual behaviours exposed to an end user of the software is the systems *Application Programming Interface* (API). By only analysing information available to the end user of the software—not relying on any internal knowledge such as source code, i.e., only using black-box information—a developer can get an understanding of how the system behaves from the perspective of an end user.

Developing software-intensive systems is an iterative activity. Both in the small—such as when a developer is making small changes to add a new feature—and in the large—maintaining a system for decades through multiple versions. The faster the feedback cycle of understanding the behaviour of the software, the faster a developer can produce new functionality and correct faults. In addition, a short feedback loop enables a user to *interactively* understand the system while it is being developed, not having to wait for an end user to find any problems.

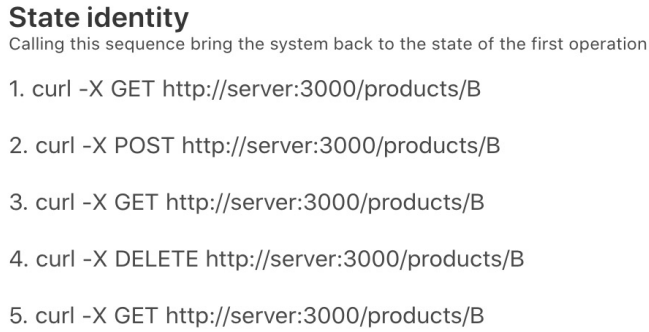

Figure 1.1: A generated example presented as curl invocations.

Based on this observed problem—the challenge of understanding the behaviour of complex software systems throughout its life cycle—, the overarching research goal in this thesis is thus; *to support developers, testers, and end users in exploring the behaviour of a system through its API.* Doing so, would allow the user of the approach to get feedback that furthers the understanding of the actual behaviour exposed by the produced system—before the behaviours are exposed to end users of the software. We want to provide the user of the approach with fast feedback—an interactive approach is preferred. Therefore, we have always been mindful of the time taken while applying the approach.

In order to enable users of the approach to explore the behaviour of the system, this thesis proposes an approach to automatically generate examples of a system's *actual* behaviour, i.e., what it *does*—in contrast to the systems specified behaviour, i.e., what it *should* do. In addition, to be able to generate examples of *actual* behaviours, while not requiring the user to provide any formal specification of behaviours, we provide a black-box approach which only requires the exposed operations of the system. An example of a behaviour consists of a sequence of operations. When searching for examples of behaviours, we generate candidate sequences which *might* show an example of the sought behaviour. When the sequence is executed, the observed responses are assessed to judge if the sequence matches a general behaviour or not. Found examples of behaviours—sequences of operations—can be presented in many different ways. Figure 1.1 shows a generated example of the general "State identity" behaviour. Since the target API in this case was a REST API, the example is presented in the syntax of a common to the domain tool curl. This example shows a sequence of operations that; first queries the state (the identity state), performs a create operation, queries the state again (observing if

it has changed from the identity state), deletes the created entity, and finally, queries the final state. This example is judged to conform to the behaviour of "State identity" as the state in 1 and 3 was different, while the state in 1 and 5 was the same.

Our example generation approach builds on a random-based test generation approach, specifically, Property-based testing [30]. We use Property-based testing to not only show results as "pass/fail" outcomes, rather, we can generate and shrink examples of sequences of API operations to show general system behaviours. In this thesis, we provide example generation approaches for faults that leverage test generation for RESTful- and GraphQL-based APIs. These are then used as a foundation in our proposed approaches that generate examples of behaviours. These approaches have been evaluated on industry-grade APIs, as well as on benchmark APIs. Our results show that the proposed example generation approach can generate examples of faults (crashes), non-conformance to specifications, and *relevant*—according to industry practitioners—examples of actual behaviours. These examples aid users—developers, testers, and consumers of the API—to better understand their systems.

This doctoral thesis is structured in two parts. Part I is an overview of the thesis and is outlined as follows; in Chapter 2 the motivation for this work is introduced and the background information needed to understand the context of the contributions. This is followed by Chapter 3, where the research goals and the methodology used to achieve those goals are described. In Chapter 4, we introduce the related work relevant to this thesis. Then, in Chapter 5, we provide an overview of the contributions of the thesis and the included papers. Finally, in Chapter 6, we conclude Part I with a discussion of the work in the thesis and open future work. Part II contains the collection of the included papers of the thesis.

Chapter 2

Background and Motivation

To achieve the goal of this thesis—supporting the user of the approach in understanding their software systems—, we need the following concepts in place; we need some method to generate sequences of operations, translate them into executable form to be executed on a running system, and assess the outcome in the context of potentially shown behaviours. Further, to not put an unnecessary burden on the user and to display examples of *actual* system behaviour, we want the user to only provide the minimal required information to execute operations. Finally, to increase the reach of the approach, to different kinds of systems and to different instances of the same kind, we do not want the approach to rely on any white-box information, such as source code. We realise these concepts in proposing an example generation approach.

Understanding the behaviour of software-intensive systems is important both for the developer of the system and the end user of the system. We know from the literature that one good way of increasing the understanding of a system by several categories of users is by providing them with examples of the system's behaviour [85, 86, 39, 76, 82, 90, 81]. Examples to increase the understanding of a software system can be expressed in different ways for different receivers, such as source code examples with a sequence of system operation invocations, or as natural language. Examples, such as code examples, can provide a deep understanding of an Application Programming Interface (API) [76], how an API can be used [85], and to better predict the result of API operations [39]. Based on these findings of prior work, we know that examples are a key component in conveying understanding.

Examples are useful in advancing understanding. However, manually producing examples requires effort. This effort might deter developers from producing examples. For example, the Java Development Kit 5 only contained code examples for 2% of the API [60]. In addition, human written examples

can only account for what humans can imagine, but automatically generated examples, based on interacting with the software system, can provide examples of scenarios no human thought of—which could be examples of unwanted, but possible, behaviour. Thus, generating examples can reduce the burden of manually producing examples and provide surprising unexpected results.

Generated examples can confirm behaviours we expect, or surprise us by giving examples of unwanted behaviours. However, judging what is correct or incorrect behaviour—commonly known as the "Oracle problem" [23] in software testing—is a complex problem. But generating examples of actual system behaviour—providing the user of the approach examples of "what is"—does not require an automatic judgment of correctness. This is left for the user to decide. In order to let the user judge the example behaviours, it is important to keep the user in the loop.

Allowing the user to interactively explore the behaviour of the system enables a feedback loop where a user can, for example, make a change to the software, explore if the change provides the sought behaviour by generating examples, and if necessary, make any additional changes. When this interactive cycle is complete, the user would have a set of generated examples that (i) show the actual sought behaviour, (ii) show no examples of unwanted behaviours, (iii) provide a source for testing scenarios (both manual and automatic), and (iv) provide a source of examples to include in user documentation.

In order to better understand a software system, we want to understand its *actual* behaviour. Requirements or other formal specifications of the system is a specification of the desired behaviour, i.e., what the system *should do*. The source code is a source of truth of the actual implemented behaviour, but—as made evident by the fact that software has faults—it is not an easy task for humans to reason about the code of complex systems. In addition, interactions with other software systems, such as services, libraries, or databases, make this even harder to understand. A common way used by software developers is to use a debugger, to analyse a predefined scenario or imagined possible scenarios. Generating examples by interacting on the system level—interacting with the systems API—can provide users with an additional source of understanding in the form of generated sequences of API operations, showing how the system actually behaves.

To evaluate our example generation approach, we need some concrete instances of software systems with APIs to interact via. In this work, our primary evaluation target is contemporary Web-based APIs. This includes systems exposing APIs via a RESTfull [37] approach and via GraphQL. The choice of the type of API to use for evaluation is based on our industry context.

There are many example generation approaches proposed [94, 65, 26, 77,

49, 67, 60, 78, 27, 39]. The primary method used by these approaches is to rely on white-box information, such as source code. Gerdes et al. provide an exception and propose a black-box approach [39]. However, the approach proposed by Gerdes et al. requires a formal specification of the behaviour of the system. As stated, this is something we aimed to avoid.

Given our primary evaluation context, contemporary Web-APIs, in order to generate examples, we need means to generate and assess API operations for this kind of API. An available approach to generate operations and their parameters and assess the outcome of their invocation is test generation. For Web-APIs, test generation approaches are in their infancy. When the work in this thesis started, there existed one approach to performing REST-API test generation. This approach, EvoMaster [7], was at the time a white-box search-based approach. Thus, there was a lack of a black-box method for REST-API test generation. In addition, the evaluation of other approaches than search-based test generation was lacking. A black-box test generation approach is foundational to our goal of generating relevant examples of behaviours.

2.1 Web APIs

The *hyper-text transfer protocol* (HTTP) is in the fabric of the Web. When we visit a web-page, the web-browser will issue an HTTP-request to the server of which the page you want to visit belongs to. The server will reply with some data in the format of *hyper-text markup language* (HTML), which the web-browser will render. This way of transferring information is the backbone of the internet. In addition, HTTP as a means of transferring data between computers has created an opportunity to create new types of *application programming interfaces* (APIs).

Two methods of exposing APIs over HTTP is with *Representational state transfer* (REST), and GraphQL. Since its introduction by Fielding [37], REST is in common use. For example, the web-page "APIS.GURU"[1] lists thousands of available REST APIs. GraphQL is a more recent development, introduced in 2015 by Facebook[2]. GraphQL gives the client of the API more control of how data is queried.

Web-APIs are ubiquitous in contemporary systems. They are obviously a part of systems that expose functionality over the Web, such as an e-commerce site, or a video streaming application, but also increasingly common in indus-

[1] `https://apis.guru/`
[2] `https://graphql.org/`

trial systems. In industrial systems—such as a process-control system or an embedded system—Web-based APIs are now used as part of the digitization of process-control related information and the Internet of Things.

Web-APIs are not only used to integrate with 3rd-party service, but also in communication between internal system components. In a micro-service architecture, services can interact via Web-APIs [38]. In such systems, there can be hundreds of interacting services. This fact also means that developers of such micro-services are both the *provider* of Web-APIs and a *consumer* of other services APIs—making generated examples useful for both consumption for developers using services and to be able to provide to users of their developed services.

One possible reason for the success of REST-APIs is their ease of use. Commonly the data sent between the client and the server is encoded in the human readable format of *JavaScript Object Notation* (JSON). This fact makes it easy for developers to inspect and reason about the inputs and outputs of an API—even if documentation is lacking.

Given how common Web-APIs now are, assessing the quality of these APIs is critical. There are several aspects to consider. As with all software, we want our Web-APIs to be free from faults—such as crashes, returning an incorrect result, etc.—but as developers both provide and consume Web-APIs, understanding the behavior of the APIs is as important.

2.1.1 REST

REST has become the industry standard way of interacting with internet-based services. This trend has also spread into industries that are not internet-based, for example, automation systems. REST is a set of architectural guidelines outlined by Fielding [37] and not a protocol, i.e., it is up to each implementation how *RESTful* (conforming to the recommendations of REST) it is. However, as Fielding himself expressed, not many APIs claiming to be REST APIs are [1]:

> *"I am getting frustrated by the number of people calling any HTTP-based interface a REST API. Today's example is the SocialSite REST API. That is RPC. It screams RPC. There is so much coupling on display that it should be given an X rating.*
>
> *What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a*

REST API. Period. Is there some broken manual somewhere that needs to be fixed?"

Typically, when an API is stated to be a REST API it is most likely a Remote Procedure Call (RPC) API using JSON over HTTP [3]. The *JavaScript Object Notation*[3] (JSON) format is the conventional way of formatting entities both for the input and the output of contemporary REST APIs. For practical reasons, in this thesis, we will use the contemporary meaning of REST, not its actual meaning as defined by Fielding.

To be able to specify a REST API, the OpenAPI[4], formerly known as Swagger, specification has been proposed. It is currently the industry standard to describe modern REST APIs. Thus, leveraging an OpenAPI specification to generate tests has been of increasing interest in research, which we will expand upon later in Chapter 4.

The challenges in automatic test generation for a REST API, compared to other APIs, are; (i) to reach the SUT, valid HTTP messages must be produced. Invalid messages will not test the SUT but the web-server hosting the SUT, and modern frameworks will filter none-conformant messages before reaching the SUT. (ii) To maximize industry reach, test generation should leverage the industry-standard specification format of OpenAPI. (iii) Input generation must not only conform to the JSON format but also the valid formats of the SUT. (iv) Test generation for a REST API is performed on system level which can include services depending on other networked services such as other REST APIs or databases.

For REST API testing there exist some tools used in industry. Some notable examples are Postman[5], Dreadd[6], and Insomnia[7]. These tools mostly rely on example-based tests, i.e., the user defines test cases that are then executed, including the exact expected output. This results in a very static test suite that does not explore the system under test by, for example, randomizing input or sequences of operations. This potentially results in low coverage of the input domain and a lot of manual work in setting up the test suit. In addition, such a test suite must be manually maintained and updated as the system is changing. Thus, there is a need for automatic approaches for test generation of REST APIs.

[3] `https://www.json.org/json-en.html`
[4] `https://www.openapis.org/`
[5] `https://www.getpostman.com/`
[6] `https://dredd.org/en/latest/`
[7] `https://insomnia.rest/`

2.1.2 GraphQL

GraphQL is a data query language. It was introduced to the public by Facebook in 2015 [2]. As stated by the GraphQL Foundation, *"It provides an alternative to REST-based architectures with the purpose of increasing developer productivity and minimizing amounts of data transferred"* [2]. REST can be a very "chatty" means of communication. It is common for a client to have to perform multiple calls to a REST API to get an entity in a parent/child hierarchy. In addition, a client can not control the format of the data returned from a REST API. GraphQL, on the other hand, allows a client to perform a graph-like query, reaching nested entities and selecting the specific fields of interest, in one call. This is what is referred to as "increased productivity" and "minimizing amounts of data transferred" from the above quote.

REST is still the dominant way of exposing Web APIs but the adoption of GraphQL has increased [84], especially for APIs where different clients benefit from the flexibility of being in control of the selection of entities, where REST is a "one-size-fits-all" API.

From an API point of view, a GraphQL API has a very limited set of operations, where *Query* is the typical one. Thus, the contract between the client and server is expressed in a typed schema and the client invokes a query by sending a valid GraphQL string conforming to the schema of the API.

GraphQL poses a similar set of challenges as REST APIs for automatic test generation. (i) Messages must be valid HTTP, (ii) must conform to the GraphQL specification, and (iii) must conform to the schema of the SUT, and (iv) is also on system level.

Since GraphQL is, as REST, based on HTTP, any tool for manual test creation targeting REST can be used on GraphQL. Thus, the state-of-the-practice for GraphQL APIs is very similar to that of REST APIs.

2.2 Property-based Testing

Property-based testing (PBT) is an automatic test generation technique based on random generation of test cases. Property-based testing was introduced as a concept of random testing for Haskell programs, introduced in the tool *QuickCheck* by Claesson et al. [30]. Since its first introduction, the approach has been improved with more advanced features that randomly generate test cases, such as the ability to shrink test cases, generate test cases based on state-machine models, and support to test concurrent applications [50].

The main components of PBT are generators for creating random-based tests (random input parameters, or random sequences of operations, etc.), and

predicates to *check* the outcome of interacting with a system under test (SUT). *Properties* are defined by combinations of generators and checks, i.e. given these generators, these checks should hold. An example of a common category of properties is invariants—properties that always should hold for a SUT. A simple example of an invariant is that the number of elements returned from a sorting function should always be equal to the number of elements given as input. More complex properties can be created, such as model-based properties, or meta-morphic properties [52].

A generated test case not passing its check is a failing test case. A very useful and cost-saving feature of PBT is the ability to *shrink* failing test cases to a minimal failing case [50]. Removing redundant information and making values as simple as possible, greatly aid in understanding and debugging a failure. During the shrinking process, a PBT library will iteratively create simpler and simpler input, in order to find the simplest value that still fails the test. The simplest value is then presented to the user as a minimal case that fails the property. In this context, simpler often means smaller or shorter. For example, a list only containing the elements required for a test to fail is shorter than a list with redundant elements and thus simpler. For input values such as integers, simpler typically means as close to the failing boundary as possible. If a test fails with values larger than 64, 65 would be the simplest and smallest failing value.

Since its introduction, Property-based testing has spread to more than 40 programming languages[8]—thus it is widely available. The fact that QuickCheck-like libraries keep being created for new programming languages, provide evidence that developers think it is a valuable approach worth the effort in porting.

In this thesis, we want to generate examples to explore API behaviours. To know if an API display instances of a behaviour, we need to check for conformance, given a sequence of operations. This concept fits very well within the frame of PBT. Thus, we can leverage PBT to search for behaviours by using generators and checks specifically defined—as part of the work in this thesis—for the types of examples we seek.

PBT is of interest to us since it provides a proven technique in how to generate random input to a system and evaluate the consistency of defined invariants—such as adhering to the shape of a general example behaviour. PBT tools are also, in general, very open to extension, required to adapt it to different kinds of APIs. PBT libraries provide basic generators as the building blocks needed to build more complex, domain-specific, generators. This is

[8] `https://en.wikipedia.org/wiki/QuickCheck`

achieved by the tooling providing generator combinators, allowing combining simple generators into more complex ones.

Random testing, in general, is an effective technique in software testing [9, 10]. However, an important aspect to consider is; random what? For example, in testing a Web-API it is hardly useful to generate random bits and send those over a network connection—the probability of generating a sequence of bits to successfully reach the SUT would be very small. The ability to define generators in PBT, provides a toolbox to randomly generate values where it makes sense.

PBT has been applied with success in multiple domains. In the literature, there are several examples of the usage of PBT to successfully find bugs—examples of faults—in software used in complex industry systems. Some notable examples have been in telecom systems [17], Dropbox [53] (a file synchronization service), the automotive industry [18], and on databases [51]. Since the work in this thesis aims to be applicable and practical to apply to industry applications, the fact that PBT has been used with success on industry systems strengthens the case for PBT as a foundation for the work in this thesis.

The main use-case for PBT has been to find faults. However, recently some applications of the technique in the area of furthering the understanding of applications have been proposed. For example, QuickSpec is a tool that automatically can generate specifications of a set of pure–as in no side effects–functions [31]. This approach then enables a user to better understand their program by reviewing the generated algebraic laws of the program. The approach has been extended to be more applicable, in terms of performance and handling more complex programs [91]. Automatically generating formal specifications for a program lowers the boundary for using PBT—the user no longer needs to come up with specifications. However, many programs are not without side effects and cannot benefit from QuickSpec. Another example of where PBT has been used as a foundation to further understanding is to generate examples of formal specifications [39]. Generating examples by using PBT is very related to the work in this thesis and will be further expanded in Section 4.2.

In summary, Property-based testing is a well proven, available, and extendable approach, useful not only for fault-finding but also as a foundation of understanding software by leveraging its capabilities.

Chapter 3

Research Overview

In this section, we will break down the overarching research goal into sub-goals and provide an overview of the research process followed and the research methods used.

3.1 Research Goals

The overarching **research goal** of this work is: *to support developers, testers, and end users in exploring the behaviour of a software system through its API*. While working towards this goal, we also aim for the proposed approach to be suitable for interactive use. However, this is not included in the main goal of the work due to that; (i) to be able to do something interactively, you must be able to do it *at all* and (ii) interaction put a lot of demands on the software engineering aspects of the software produced. For example, providing the proposed approaches with a user interface where the user could interactively learn, visualize and create new experiments to explore the behaviour of their system, is a significant engineering effort. Such engineering efforts would be valuable to practitioners, but, unfortunately, are of little value for academic publications [11]. However, there is still a large value of the goals of this thesis in enabling a future more interactive approach. We know how a better understanding of the requirements of such an approach regarding how to generate examples.

In order to propose and evaluate approaches to further this research goal, we have to be more concrete than targeting "software systems". Due to this fact, we chose to focus on Web APIs. Web APIs are ubiquitous in general and also used in our industrial context. Therefore, focusing on these kinds of APIs provides reach and relevance of the research performed both for practitioners

in a general context and in our specific industrial context.

The overarching research goal was worked on incrementally. This resulted in two sub-goals. The first sub-goal was to generate fault-finding examples, such as crashes, for REST APIs (Paper A) and GraphQL APIs (Paper B). Achieving this goal is an enabler for generating more, potentially, interesting examples. The second sub-goal was to generate examples of general API behaviours, automatically providing users with examples of how to, for example, create an entity in the system. This goal was approached both in the general case of any API (Paper C) and specifically for REST APIs (Paper D).

In summary, the research goals of this thesis are;

- RG: Support developers, testers, and end users in exploring the behaviour of a system through its API.
- RG-A: Understand how we can automatically generate examples of fault-finding and specification compliance in our context.
- RG-B: Understand how to automatically generate *relevant* examples of general API behaviours.

3.2 Research Process

The research presented in this thesis is the result of an industry-academia collaboration. I have spent close to two decades as a professional software engineer working in industry. My own experience, in combination with experience gathered from colleagues, resulted in me being able to bring actual industry problems into the research project. This might imply that these are not general problems, outside of the domain of software systems we have developed. However, the positive side is that this "context-driven" research can produce more realistic and industry-relevant results, compared to research performed in academia with the aim of being general [25].

The research process in these doctoral studies has been incremental. We started with an initial research goal, based on problems experienced in industry. Inspired by the research goal, some initial ideas were formulated and followed by a state-of-the-art survey. New research goals were incrementally created based on the results of the work performed. An overview of this process is shown in Figure 3.1.

It became clear at the beginning of this process that the current state-of-the-art in the area of automatic test generation of contemporary Web APIs was in its infancy. Therefore, the first part of the process focused on "getting off the ground"—we could not advance the understanding of an API if we could

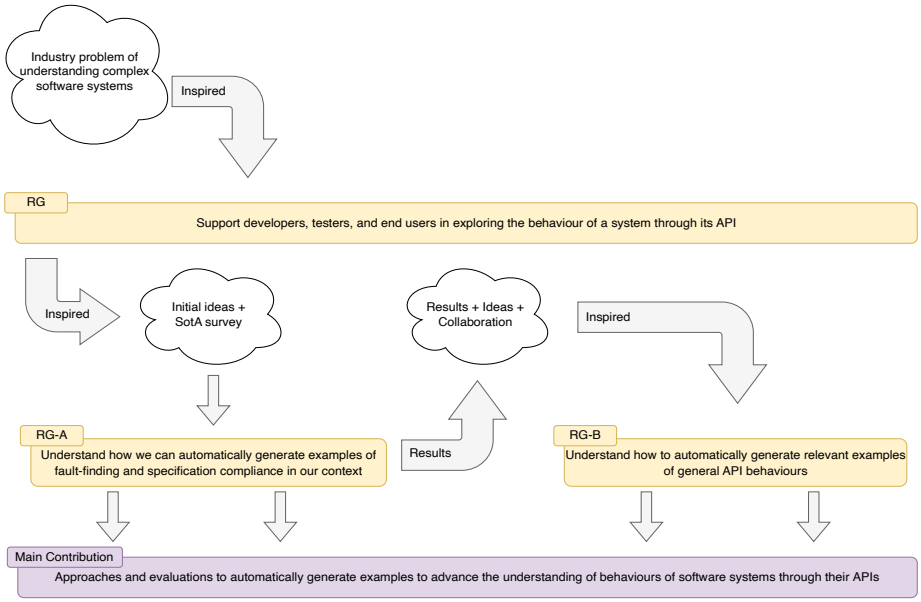

Figure 3.1: Overview of the research process

not automatically interact with it. A natural consequence was thus to focus the first part of this work on generating examples of outcomes traditionally close to automatic test generation, such as fault-finding, and base any behavioural explorations of the API on that work in the second part of the process.

After the first part of the research process, we collaborated with a distinguished researcher in the relevant field of Property-based testing—which all our approaches are based on—and with experience from example generation. This collaboration, in combination with the results from the first part of the process, resulted in the research goal for the second part of the research process.

To ensure the practical applicability of the results produced by this research process, all proposed approaches have been evaluated on real-world software. In addition, the software we have evaluated our approach on is available as open-source, which should ease in assessing if the proposed approaches fit in some other practical context. Further, all proof-of-concept implementations have been made available as replication packages.

During a verbal presentation, a very senior researcher in our field argued that in software engineering research we cannot do experiments as in other fields, for example, medicine. She argued that in a field such as medicine when conducting an experiment, it is possible to include a representative sample

of a population because we have data describing the population. However, in software we have no census data on the "population of programs", thus we cannot get a statistical representation of that population to perform any experiment on. What we are left with is case studies. For me, this was an enlightening argument. In addition, as a practitioner reading papers, I prefer case studies as it is more clear if the proposed approach is applicable to my context or not. As we, as a field, perform more and more context-driven case studies, a body of knowledge will emerge which can show general results [25]. With the industry-academia collaboration and context-driven setup, it might be unsurprising that case study [88] is the primary research method used in the contributing papers.

In the later part of the process, in Paper D, a proof-of-concept implementation containing several of the contributions of this work was evaluated using multiple focus group [63, 87] sessions with industry practitioners. This evaluation was placed in a late stage in the process in order to have an actual approach in place that could generate examples from an API. We sought to evaluate if our contribution actually could automatically generate *relevant* examples and not a hypothetical scenario.

Enabling a successful technology transfer is a preferable goal of an industry-academia collaboration. Gorschek et al. present a model for technology transfer in practice but also recognize that such a model must be adapted to the current industry context [44]. In this work, we had to adapt to organizational and technical changes at our industrial partner. The effects of this are that (i) our evaluations could not always use software from our industrial partner, and (ii) the transfer of solutions is not complete. The proposed approach in Paper A was applied to software from our industrial partner. However, after that work, there were technological changes making any test automation challenging. This change also resulted in practitioners close to this project needing to learn how to adapt to this technological change, making it hard to know what problems needed to be solved. At times, it was hard for the engineering teams to get the new platform running, making any scientific inquiry close to impossible. Thus the work following Paper A used open-source projects in the evaluations. Even though the underlying technology platform changed at our industrial partner, the context of Web APIs is still very relevant and our work could progress in that area. This relevance was validated in Paper D, where the results of the focus groups showed great utility and interest in the proposed approach.

To enable a successful technology transfer, well-engineered working software is essential. However, spending time on engineering software is not something that is of great interest to the academic community [11]. In the do-

main of test generation targeting Web APIs, the effort of a successful approach is significant. For example, one of the most mature solutions in the area of Web API test generation, EvoMaster, consists of approximately 200 000 lines of code and extending it with support for GraphQL took around 10-15 000 lines [24]. Currently, there have been seven people active in the development of EvoMaster and it has been used in close to twenty peer-reviewed publications [16]. Thus, from a research process point of view—of a single PhD student—, a trade-off must be made on how much effort to spend on the software vs. spending time on novel research findings. The software output from this thesis is somewhere between a research prototype and software usable by practitioners. For each contributing paper improvements have been made to the usability of the tool, resulting in the final version being possible to use without detailed knowledge. However, there is still a lack of engineering work hindering technology transfer. For example, OpenAPI specifications are essential to this line of test generation. However, their quality can vary wildly. Thus, a well-engineering tool in this domain must be able to handle a large set of OpenAPI specifications to not be too limiting to use.

In summary, the following research methods have been used in the contributing papers;

Paper A: This is a case study of automatic REST API exploration applied both to industry-provided services, as well as the Open Source Software (OSS) product GitLab.

Paper B: This is a case study of automatic GraphQL API exploration. In this case study the evaluation was performed on the OSS industry-grade product GitLab.

Paper C: This is a case study of generating relevant examples of general behaviours of an API. The evaluation was performed on the OSS industry-grade product GitLab. In addition, a controlled experiment was performed in order to measure the impact of different strategies of input parameter selection in the generated examples.

Paper D: This paper contains multiple case studies on a benchmark suite of systems commonly used to evaluate test-generation approaches for REST APIs. In addition, with the work in this paper, we performed two focus group sessions with industry practitioners.

Chapter 4

Related Work

In this chapter, we present the state-of-the-art in the areas closest related to the proposed thesis. On a high-level view, in this thesis we use Property-based testing as a foundation, to generate examples, mainly applied to explore behaviours of Web-based APIs. Thus, the two main related areas to the work in this thesis are Web-API test generation and example generation.

4.1 Web API Test Generation

In this work, we target the two contemporary most commonly used methods of exposing Web APIs—REST [37] and GraphQL. In this section, we review the state-of-the-art in automatic test generation of such APIs and how it relates to our proposed approaches.

4.1.1 Test Generation for GraphQL APIs

The interest in GraphQL in general in the academic community is increasing [84]. However, the specific field concerning test generation for GraphQL APIs is in its infancy with few approaches proposed.

Vargas et al. [95] proposed the first test generation approach targeting GraphQL APIs. Their approach is based on "deviation testing", where an initial test case is changed by applying different "deviations". An example of such a deviation is to remove a field from a query. The difference in executing the original test case and the deviating test case is compared and assessed according to the deviation rule used. The test case deviations are created automatically, but this approach requires a set of existing test cases.

In Paper B [57] we proposed the first fully automatic approach which only requires the GraphQL schema—in contrast to the required pre-existing test

cases by Vargas et al. [95]. Our proposed approach found new faults in an industry-grade system and can provide up to 100% schema coverage.

Zetterlund et al. introduce an approach which harvests production GraphQL queries and automatically creates test cases from these queries [99]. This approach requires interaction with the API in order to record queries. This is an additional effort—compared to a fully automatic approach—but has the possibility of producing realistic test cases not thought of during development. Using the schema coverage metric—one of the contributions in Paper B [57]—their proposed approach reached coverage of 26.9% on an open-source SUT and 48.7% on an industry SUT, while also finding new schema faults, where the SUT response and the schema do not match.

The most recent proposed approach in the area of GraphQL test generation is a white- and black-box approach proposed by Belhadi et al. [24]. This approach is based on evolutionary algorithms and is implemented as an extension to the tool EvoMaster [8]. By using the "Many Independent Objects" (MIO) [6] evolutionary algorithm, their approach improves code coverage with on average 7% and find on average 10.2% more faults, compared to a random-based approach.

4.1.2 Test Generation for REST APIs

Test generation for REST APIs is a very recent research area. This area has greatly expanded since the start of the work in this thesis.

Before the first approaches targeting fully automatic test generation, as we do in Paper A and Paper D, a few approaches were proposed to generate test cases for REST APIs that relied on manual work.

Chakrabarti et al. proposed a specification language to express test cases for REST APIs [28]. These specifications were then used to invoke a REST API and assert the responses. Although, there was some automation proposed in [28], it was limited to generating combinations of query parameter values.

In another approach, specifically targeting "connectedness", i.e., if resource URLs returned by the REST API could be reached, Chakrabarti et al. generated test cases given a manually created formal specification [29]. This approach requires the formal specification of the Web service to be available and only test for the defined "connectedness" criteria.

Model-based testing has been proposed to create test cases for REST APIs [83, 36]. For example, Pinheiro et al. proposed an approach where UML protocol state machines were used as a basis for test case creation [83]. Test cases was created based on covering the model. To use a Model-based approach, the user must first create the model, which can be a lot of manual

work, which might limit the practicality of such approaches.

Seijas et al. proposed an approach using Property-based testing to generate test cases for REST APIs [64]. However, this approach required a large amount of manual work since the model used for generating tests had to be created manually.

The approaches presented so far require a large component of manual work to be able to generate test cases. Hence, the automation is mostly focused on the execution of test cases, rather than automating the process of test creation without a large effort of manual work. The approaches included in this thesis aim to provide means for its users to better understand their systems *without* requiring a large manual effort in doing so.

At the start of this thesis work in 2019, two approaches for automatic test generation targeting RESTful APIs existed. The first, EvoMaster, was introduced in 2017 as a fully automated white-box approach, with the goal of optimizing for code-coverage and fault finding of the produced test cases [7]. EvoMaster based its test suite generation on the "Many Independent Objects" (MIO) [6] evolutionary algorithm [8]. In addition, EvoMaster leveraged OpenAPI (then known as Swagger) specifications to know which operations the REST API exposes. Using OpenAPI specifications has been the primary source of this information for all test generation tools in this area, it has become an essential component for research in REST API test generation. In contrast to EvoMaster, in Paper A, we proposed a black-box approach—making the approach available where white-box information is not accessible or not practical to use. In addition, we proposed an approach based on Property-based testing (PBT) to automatically create PBT generators and checks, i.e., properties to generate test cases for a REST API, differing from EvoMasters use of evolutionary algorithms.

The second approach available when this thesis work started, was introduced in 2018 by Ed-douibi et al. [34]. They proposed a black-box, model-based approach for test generation of REST APIs. This approach created a model based on the provided OpenAPI specification. The approach results in test cases being statically created, and no run-time information is included. This is a major difference between our property-based approach in Paper A. We continually generate new operation sequences, where values seen during the execution of the tests can be used in new generated sequences. In this way, entities created in running the tests can be interacted with, enabling sequences to be stateful—performing multiple operations on the same entity. This could potentially be achieved in the static case as well, but for some APIs, the required data is not known until run-time. For example, when creating an entity it could be assigned an identifier on the server—unknown to us prior to exe-

cuting the operation—which is required as a parameter when interacting with the entity.

As mentioned, when the work of this thesis presented in Paper A was started, the existing approaches were the ones proposed by Arcuri [7] and Ed-douibi et al. [34]. However, while working on Paper A, in 2019, another black-box approach was proposed in the form of RESTLER [19]. RESTLER is a stateful fuzzer for REST APIs. Before test execution, RESTLER performs a static analysis of the given OpenAPI specification. The result of the static analysis enables RESTLER to call sequences of operations in dependency order, i.e., successfully calling operation A might require that operation B is called first. In addition, RESTLER adapts to run-time information, such as if a sequence is successful or not—unsuccessful sequences are avoided in future calls to the API. A difference between our approach in Paper A and RESTLER is that we do not make any static analysis prior to executing operations. Combinations of operations are generated randomly. This might result in random sequences that "do not make sense" from a usage perspective of the API, but might also expose bugs for sequences the developers did not anticipate. Another difference is the generation of operation parameter values. RESTLER uses a pre-defined dictionary, such as 1, 10, and -1 for integer values, where our approach randomly generates and *shrinks* values. This results in a much larger coverage of the input domain, and when an input is found causing a fault, it will be shrunken down to the minimal reproducible case. The concept of shrinking also applies to sequences in our property-based approach—a sequence exposing a fault will be shrunken to a minimal sequence.

QUICKREST, introduced in Paper A, was presented at a conference in 2020. At the same conference, in the same session, another approach for test case generation for REST APIs was presented, called RESTTESTGEN [96]. The RESTTESTGEN approach by Viglianisi et al. is a black-box approach which builds a dependency graph between operations in the given OpenAPI specification. When building the operation dependency graph, the fields of the operation inputs and responses are considered. However, this approach also considers the case where names might not match and infer relationships by different textual properties. For example, one operation could return a "petId" and another operation using this information might have specified its parameter as just "id" but has "getPet" as its operation name. The operation name can then be used to infer a relationship. The dependency graph is used to create test cases for both nominal and expected failing cases. RESTTESTGEN has later been extended with support for authentication and testing time budget [33] and to detect the security vulnerability of "mass assignment", where read-only resources can be written to [32].

As mentioned, QuickREST, presented in Paper A, uses property-based testing at its core. Another proposed approach also based on property-based testing is "Schemathesis" [98, 47]. Schemathesis targets both OpenAPI and GraphQL APIs and one of its strengths is the ability to handle a wide range of OpenAPI specification instances—which in real-world scenarios can be full of errors, violating the OpenAPI standard. In addition, Schemathesis has performance checks and checks for conformance to HTTP semantics.

After this point in 2020, there has been a large increase in the focus of this research area. A recent survey found 92 papers related to this research area, where 9 were published before 2017 and the rest between 2017 and 2022 with a yearly increase in the number of papers [42].

After the initial set of approaches, new challenges were identified and approaches to address them were proposed.

One challenge when performing test generation for REST APIs is dependencies between parameters. Such "inter-parameter dependencies" are quite common, as shown by Martin-Lopez et al. [72]. In their study of 40 real-world APIs, 85% of the APIs had some sort of dependencies between their parameters. Some categories of these dependencies are, for example, "required"—some parameter requires the presence or specific values of another parameter, and "or"—one or more of the parameters must be included [72]. Currently, there is no support to formally declare such dependencies in the OpenAPI specification. Thus, it is currently, if at all, done by documentation annotations.

One proposed solution to the problem of finding "inter-parameter dependencies" in test generation, is to explicitly define them with a domain-specific language called "Inter-parameter Dependency Language (IDL)"[1] [69, 71]. For example, IDL can define that if a specific parameter is present, another parameter is required, or must have some specific value. IDL can be used as an extension included in an OpenAPI specification.

IDL support has been integrated into one test generation approach called RESTest [73, 74]. RESTest can leverage IDL specifications to use constraint solving to generate a much higher number of valid operation sequences when performing test generation, compared to random sequences. IDL constraints can also be used as oracles, to verify that the API do not break any specified constraints. However, the performance of the approach depends on the effort in specifying the parameter dependencies.

An additional novelty added to RESTTest is the ability to generate realistic values for parameters [4]. This approach uses information from the OpenAPI

[1] `https://github.com/isa-group/IDL`

specification, such as parameter names, and makes queries to some knowledge base to find realistic values to use. This approach can greatly increase the probability of successfully generating operation parameters. For example, if an operation parameter requires a country code as an input, the likeliness of randomly generating such is low, but finding a set of options in a knowledge base when searching for "count code" is much more likely.

The first proposed approach in this area, EvoMaster, has seen several extensions to address open challenges. It is currently the only tool supporting both black-box and white-box test generation [12]. EvoMasters white-box support started by targeting the Java Virtual Machine (JVM) [8], but now also support the NodeJS runtime [102, 103], and most recently the .NET platform and the C# language [43].

In addition to supporting more platforms and languages, EvoMaster has had several improvements to increase its fault finding and code coverage capabilities. For example, cases where the SUT uses a SQL database, which can be hard to cover if not explicitly considered in test generation [13, 14]. "Testability Transformations" is another area where EvoMaster has been extended, which transforms code to provide better heuristic values when searching for test cases covering code [15].

Another area of improvement is in how sequences of operations can be more efficiently generated. Since there typically are relationships between operations in an API, creating random sequences might not yield successful execution of operations. To improve on the random sequences of API operations generated by EvoMaster, Stallenberg et al. use hierarchical clustering to learn linkage trees to preserve patterns of successful operations [92]. This approach hinders the random removal of successful sequences in later generation iterations. Another approach implemented with EvoMaster to handle relationships, is to not only consider relationships between operations but also the dependencies of resources that operations of a REST API act on [104, 105]. Such dependencies can be detected both statically, from the OpenAPI specification, and dynamically, from EvoMasters runtime fitness feedback of the test generation process.

As mentioned, many different approaches have been proposed. Except for EvoMaster, the proposed approaches only rely on black-box information. However, as Martin-Lopez et al. show, a combination of approaches can be beneficial [70]. Their results show that seeding EvoMasters search-based white-box approach with tests generated by ResTest black-box approach, increase coverage and fault finding.

One of the early approaches already mentioned, RESTler [19], have seen several extensions. One extension adds security checks, checking for security

properties such as “use-after-free” and “resource-leaks” [21]. For example, the “use-after-free” property checks that a REST API resource that has been deleted is no longer accessible by other API calls. Another RESTler extension checks for regressions between different versions of a REST API [41]. Such regressions can be in the form of making breaking changes to the API specification, or in the responses returned from the API. Finally, to be more efficient in finding bugs, RESTler has been extended with “intelligent” fuzzing strategies to produce the values sent to the API under test [40]. These strategies manipulate the JSON payload sent to the server by, for example, duplicating values, changing the type of values and dropping values. In addition, test cases generated by RESTler have been used as seed input to Pythia, a grammar-based fuzzer which uses a learning component guided by coverage information to mutate its generated test cases [20]. Pythia improves both code coverage and fault finding compared to RESTler [20].

OpenAPI specifications are a critical resource for the current SotA in REST API test generation. Thus, it is desirable to extract as much information as possible from it. In addition to the formal parts of the specification, such as defining operations and parameters, users also put information in natural language in the specification. For example, descriptions of parameters can mention example values or allowed domains of the parameters. In order to formalize this information, Kim et al. leverage natural language processing (NLP) [61]. Their approach extracts informal natural language information from OpenAPI specifications and adds it to an updated version of the specification using its formal properties. For example, a description of a parameter might mention example values, the approach then extracts these and puts them into the OpenAPI specification’s formal property “examples”. As shown in their evaluation, providing this enhanced version of an OpenAPI specification increases the effectiveness of REST API test generation tools [61].

Although test generation and fuzzing of REST APIs have been a thriving research area in recent years, there are several challenges left to address. In a recent comparison of tools in this area, Zhang et al. showed that no current tool reaches above 60% of code coverage in a benchmark including 19 APIs [100]. In addition, the current SotA tools only find a small number of different types of faults, typically crashes and in some cases miss-alignment of the specification and the API. Some of the issues identified by Zhang et al. currently preventing tools from reaching a higher coverage are; handling of databases, understanding constraints on inputs, and schema inconsistencies—the schema might be under-specified making it hard for to tool to generate relevant tests [100].

Relating to another challenge, the time required for test generation, Zhang et al. found that good results can be achieved within 6 minutes compared to 1 and 10 hours [100]. A better result could be achieved with a larger time budget, but for several APIs, the improvements were small or non-existent.

By applying EvoMaster in an industrial case study, Zhang et al. identified challenges relevant to REST API test generation tools in general [101]. For example, the challenges identified included; handling the state of the system—as resetting the state can be very costly and is not practical to do on each test execution—, the time efficiency of fuzzing is important to industry practitioners, and producing more readable test cases. In addition, the industry practitioners wanted testing criteria more related to business logic, performance, and security [101], which can be seen as part of the identified challenge of producing stronger oracles [70]. Basing test generation on producing relevant examples, as we do in Paper D, makes an effort to address parts of these challenges. An example-based approach can produce test cases less sensitive to the state of the SUT in a short amount of time. In addition, as they are based on behaviours, the tests can be more related to the behaviour of the business logic.

Some challenges have been identified in the context of online testing—where the SUT is running in production. For example, the challenge of automatically identifying the cause of a fault, automatically selecting different test strategies based on the goal of the testing, and optimising the computational resources needed for the test generation process [75]. Although these challenges were identified in the context of online testing, they are relevant in the offline case as well.

In a case study on a real-world healthcare system, Sartaj et al. observed some lessons learned from applying a SotA REST API test generation approach [89]. Generating realistic data fit for the domain of the SUT is a challenge, but also adhering to more general formats as a phone number—a challenge also identified by Martin-Lopez et al. [70]. They also observed that not all properties and constraints stated in the OAS were considered by the applied approach. They also observed that time is a constraint when testing industry systems in development and generating and executing a large number of tests is not always possible—similar to what Zhang et al. also reported [101]. Therefore future test generation tools should aim to be more efficient and optimize the operations generated with respect to time.

The problems of generating input parameters satisfying constraints of the API, such as matching specific formats, and finding dependencies among operations—calling operations in a required sequence—remain challenging for the current SotA approaches [62].

EvoMaster has been the best performing tool in multiple evaluations [100, 62]. However, the area of REST API test generation keeps being a thriving research area with new approaches proposed. One recent promising approach to tackle the dependency problem of resources in the API is to base the dependencies on a tree-based rather than graph-based structure [66]. Combinatorial testing can be used to better cover the API operations [97]. One other suggested recent approach to automatically produce stronger oracles—one of the open challenges [70]—is to find invariants based on the specification and requests/responses interacting with the SUT [5]. Finally, a recent approach based on Reinforcement Learning, by Kim et al., is challenging the position of EvoMasters search-based approach as the best performing approach in coverage and fault-finding [80].

We have mentioned how some of our contributions relate to the work presented in this section. In summary, our first approach, QuickRest in Paper A, introduced an approach based on property-based testing generating stateful sequences and providing shrinking of parameter values and sequences. Following the initial set of proposed approaches, improvements were made to reach higher coverage and better fault finding. However, here we diverge from other proposed approaches, as we aim to leverage test generation as a means of better understanding the behaviour of an API, as in Paper D. Our example generation approach builds on the test generation core of QuickRest and faces the same challenges as other approaches in this area to generate valid operation sequences to invoke on the SUT. In addition to the challenges of test generation for REST APIs, an example generation approach also faces challenges of generating *relevant* examples, valuable to a user for understanding and not only furthering a criterion as code coverage.

4.2 Example generation

The main theme of this thesis is to propose approaches to automatically produce output that can advance the understanding of APIs by users. A common such output is examples. In this section, we introduce the related work in the area of generating examples. In addition, we also present the work done to show that examples are a valuable artefact of an approach aiming to increase the understanding of a software system.

4.2.1 Usefulness of Examples

In this thesis, we propose approaches to automatically generate examples of API interactions. Given that, it is relevant to ask the question if practitioners

care about examples to better understand software systems.

In a field observation of how to build more usable APIs, McLellan et al. found that code examples helped programmers better understand an API [76]. The code examples speed up the time needed to learn the API by the programmers, as well as understand relationships between API calls [76]. In addition, subjects were observed when consuming an example, to better understand how an example should be constructed. The researchers observed that the examples gave the subjects a deep understanding of the API [76]. However, examples could also confuse the subjects, for example, if they did not adhere to best programming practices, such as using hard-coded values [76]. This study shows that code examples are helpful to practitioners, but care must be taken in their design to not confuse the users.

Nykaza et al. performed 57 interviews of users consisting of both developers and technical support staff to assess what is needed in order to understand a software development kit (SDK) [82]. Examples of what the SDK can do and code samples of how to use the SDK was deemed as important [82]. In addition, programmers want samples that are robust and that can be used as part of new solutions, preferable being able to reuse by copy and paste [82]. Further, observations of the developers, show that developers want information as they need it to solve a problem as it occurs. Therefore, the developers would rather have examples solving real specific problems they experience, instead of long tutorials [82].

Investigating how developers can obtain an understanding of software artefacts for the purpose of using this knowledge when creating new software, Shull et al. showed that example-based learning was an effective way of learning [90]. The context of this study was in object-oriented frameworks with students as the subjects. This study showed that learning by examples helped novices quickly create working software systems [90]. In addition, one implication of the results presented in the study, is that reuse of much previously seen code is useful when constructing new applications [90]. This is in line with the findings by Nykaza et al. [82] discussed above—developers like to reuse code and examples can be such a source of reuse.

In a study of what users want in the documentation of software systems, examples were one of the parts identified [81]. Specifically, users of the software documentation wanted solutions to their problems expressed as examples [81]. The study was performed as 25 interviews of software users. This indicates that not only developers themselves benefit from examples, but also the end users of the software produced. The implication for the work in this thesis is that generated examples can both be a source of increased understanding for developers in creating software and also save time for developers in creating

examples for their end users.

To investigate what developers require in order to better understand APIs, a survey of developers was conducted by Robillard [86]. The result shows that developers want access to "good examples" [86]. When expanding the prior study with the goal of studying API learning obstacles, Robillard et al. found that, again, in order to help developers understand and use an API, examples are an important factor to include in learning resources [85]. In this study, Robillard et al. conducted surveys and interviews with more than 400 developers at Microsoft [85].

Gerdes et al. proposed and evaluated an approach that generates examples [39]. The evaluation shows that the participants' understanding of an API was significantly improved if having access to the approaches generated *good* examples, compared to a generated reference set of examples based on covering all the code [39]. Understanding was defined as being able to predict the values returned by an API given a sequence of API invocations. The subjects in the study were tasked to predict the outcome of sequences of API invocations. As an aid in doing so, one group had access to the generated examples based on being *good*, and one group had access to the examples generated to cover all the code. This study shows that access to *good* generated examples significantly can help in the understanding of an API [39]. In addition, Gerdes et al. results show that different heuristics must be considered when generating examples that should be *good*—in helping the users better understand an API—, compared to examples that cover all the code.

In summary, The current literature of examples relating to understanding software and APIs, show evidence towards what is perhaps quite intuitive; as humans we learn from examples and software is no exception. In addition, we have evidence that examples can cover the full range from novice [90] to senior developers [85], which is the range of experience probably found in many workplaces.

4.2.2 Good Examples

In the previous section, we established that examples are useful for helping developers and users better understand software systems. However, several studies put forth that users do not want—or are helped by—just any example, some properties of the examples must be fulfilled. In this section, we will present the findings in relation to such properties, in order to arrive at some notion of what a good example should contain.

As noted in the previous section, users want examples to show best practices [76], to be robust so they can be reused [82], and to solve specific prob-

lems [82]. In addition, some studies note that examples should be "good" [86, 85, 39]. But what is a "good" example? In the study by Robillard et al., including more than 400 developers at Microsoft, some aspects of "good" was found [85]. The study found important aspects to consider when creating examples, such as the size of examples, the complexity of the examples, and that examples should show "best practices" (as noted in the previous section, McLellan et al. observed that examples not following "best practices" can even confuse users [76]). The authors found that examples should be small in size, but not too small—examples showing just a single API invocation were not deemed as useful by the participants. Size also ties into complexity, the developers preferred examples which show a single concept—not intertwined API concerns. Thus, in summary, and relevant to our example-generating approaches, is that the participants in the study preferred small examples that show a distinct behaviour of the API.

In the approach introduced and evaluated by Gerdes et al., the notion of "good" was also discussed [39]. As mentioned in the previous section, the evaluation showed that access to good examples significantly helped the users in understanding an API. The proposed and evaluated approach generates examples that show—in a sequence of API calls—an interaction between two of the functions provided by the API [39]. In addition, as the authors note, to be "good", all the API calls in an example should be essential to the point of the example [39]. This definition is related to the aspects of size and complexity, as discussed above in the context of the study performed by Robbilard et al. [85], i.e., examples should not be too small and show distinct behaviours without unrelated operations.

With the background of the work discussed by Robbilard et al. [85, 86] and Gerdes et al. [39], in order to advance users understanding of an API, generated examples should be *good*. There are synonyms of "good" to choose from, such as *interesting* or *relevant*. In our work, we commonly use *relevance* to indicate examples that are helpful to developers, either in advancing their own understanding of an API, helpful as a source of tests, or to create documentation. We define *relevance* in accordance with the findings in the studies discussed. Thus, in our work, a *relevant* example is an example that shows a sequence of API operation invocations that demonstrate a specific behaviour of the API. In addition, a *relevant* example should be small. To achieve small examples, we shrink examples towards the smallest number of operations that can show the specific behaviour. Examples generated with this definition in mind were put to the test in the evaluation of Paper D, where users found the generated example to actually be relevant to their understanding of an API, as well as being a useful artefact in their development activities.

In order to facilitate the relevance aspect of the reuse of our generated examples, we use *symbolic references*—introduced in Paper C. By using symbolic references, the generated examples do not depend on specific values at the time of generation, but rather only how operations relate. This property of the examples increases the likelihood of reuse in other contexts.

4.2.3 Approaches to Generate Artefacts for Understanding

The primary goal of generating examples is to further users' understanding of a software system. In order to interact with the API of a system, some form of specification is required. The available specifications can range from only including the available operations, and their inputs and outputs, to a formal specification of the behaviours of the system.

Efforts to aid the user in understanding by generating some artefact go as far back as, at least, 1982, with the introduction of the "GIST English Generator" by Swartout [94]. GIST is a formal program specification language, and as the author points out, formal specification languages are oftentimes hard to read, for example, due to their syntax [94]. Swartouts approach generates natural language from these specifications to help the user better understand the specification—making it more readable—and to easier spot mistakes in the specification [94]. Some of the examples we generate in our proposed approaches are expressed in human-readable form. However, the work by Swartouts [94] shows that it is feasible to automatically generate information artefacts that can make system specifications more understandable, as we aim to do with example generation.

Another effort to make specifications more understandable, proposed by Lavoie et al., generates natural language descriptions from object-oriented models [65]. In this approach, in addition to producing natural language descriptions of the specification, an example section was included in the final artefact. These examples contained concrete instances of the classes used in the specification model, and also negative examples showing what the specification made not possible. In addition, as with all the approaches proposed in this thesis, the approach by Lavoie et al. [65] is independent of knowledge of the implementation domain. Hence, the approach can not detect semantic errors relevant to the actual domain but can aid the user in detecting those, given the automatically generated second view of the specification [65].

Burke et al. introduced a system to automatically generate natural language from a formal specification by using a grammar-based approach, thus any modifications required by the user can be done on the grammar level [26]. The goal of the work is to help users not familiar with formal specifications, such as cus-

tomers or managers, better understand the specification [26].

When only the specification is considered when generating an artefact usable to increase understanding, as in [94, 65, 26], and not the actual software, we can only better understand the specification and not the system which implements it. While all the approaches in this thesis use some form of specification, such as OpenAPI-specifications, all generated examples are produced by interacting with the running system. Hence, producing examples of the actual behaviour, which might not be the same as the specified behaviour.

Instead of using a specification as a source to generate examples, a programming language syntax grammar can be used. In an effort to generate examples for inclusion in tutorial-like material, Mittal et al. used Lisp syntax grammar to generate examples [77]. As the generated examples of this approach are indented for documentation or learning purposes, there is a focus on generating interesting examples. In the example-generating approaches we propose in this thesis, we do not require any syntax grammar or other language details, only the available API operations.

A common approach to generating examples is to use white-box information, such as a corpus of source code example uses.

Holmes et al. introduced an approach which uses the structure of source code to find relevant examples in a source code repository [49]. Existing applications are used to generate the repository to draw examples from. As the approach uses source code as the query interface, the burden on a developer using the approach is low [49]. When a developer initiates a search for a relevant example, the approach will match its structure and the best result is returned to the developer [49].

Another approach to recommend examples of API uses based on search is proposed by Mar et al. with PropEr [67]. PropEr uses a code search engine to find example candidates, with metrics added, for the user to evaluate.

Instead of presenting a user with relevant examples based on a search query, as in [49, 67], API documentation can be augmented with generated examples, as done by Kim et al. [60]. This approach is also based on an available repository of code to generate examples. Examples are generated from the repository and added to the documentation of an API [60]. Closely related to the approach by Kim et al. [60], the approach proposed by Montandon et al. also relies on a source code repository to mine for examples [78].

Examples can also be generated as executable code. By using an algorithm based on path-sensitive dataflow analysis, clustering, and pattern abstraction, Buse et al. generates API usage examples as program snippets [27]. But as with other white-box approaches, it relies on a software corpus of available usage examples. The algorithm proposed in this approach can produce ex-

amples which only contain the API calls required to show a behaviour [27] (which is a sign of a good example[85, 39], as discussed in more detail in the previous section). This is in contrast to approaches using mining strategies, which can extract examples containing unnecessary invocations. In addition, it is worth noting since the generated examples are syntactically correct and well-typed programs by construction, they can have a higher rate of correctness than human written examples, as humans sometimes make mistakes. In the evaluation, 94% of the participants prefered the generated examples by the approach proposed by Buse et al. [27] compared to the discussed approach of Kim et al. [60].

Another white-box approach that parses the source code of existing applications in order to find usage examples is MUSE, proposed by Moreno et al. [79]. The novelty in the approach is to use backward slicing of the usage examples in order to understand what parts of the found example can be removed, to simplify the example. In addition, the approach ranks the examples and takes measures to detect and group similar examples—only reporting one Representative example of a group [79]. In our work, the approach used to generate small examples with redundant operations removed, is to use shrinking, as is common when using Property-based test generation [30].

As an improvement of the approaches proposed by Moreno et al. [79] and Kim et al. [60], Gu et al. proposed an approach finding examples by using graph kernels [45]. This approach represents code examples as usage graphs instead of sequences of API calls with certain features, which are then clustered. However, like previous approaches, this approach also relies on a source code corpus to find examples.

Multiple approaches presented help the user in generating a set of candidate examples from a corpus. However, helping the user in selecting an idiomatic use of an example API method is also important. In an effort to do this, Barnaby et al. [22] proposed an approach which mines a code base for example uses and highlights common usage patterns in the examples, while also deemphasises less common parts of the example. This approach relieves the user from the burden of understanding which parts of an example are common or uncommon usages.

Another approach that also put more focus on the user, as in [22], is to aid a user in creating examples from an existing code base, as done by Head et al. [48]. This approach supports an interaction between the user and the proposed tool, that allow the user to extract and simplify an example from an existing code—interactively making the example concise and readable and with the intended behaviour [48]. The example generation approaches we propose in this thesis, especially in Paper C, are indented to be interactive, where

a user can react to what a generated example show and adapt accordingly.

Deep learning has also been used to generate sequences of API operations [46, 68]. By using a deep learning model, Gu et al. can learn and leverage the semantics of words to better generate examples [46]. The user provides a query in natural language and the approach generate a sequence of API calls. The language model used is trained on a corpus of source code containing API operation sequences annotated with natural language, which is extracted from the first sentence of the operations documentation comment [46].

Martin et al. [68] tried to reproduce the approach used by Gu et al. [46] and apply it to Python source code, instead of Java source code, as used in the original study. In addition, Martin et al. [68] also compare the approach by Gu et al. [46] with CodeBERT [35]—a pre-trained model for programming and natural languages. Their reproduction study shows that finding examples based on source code from Python and Java perform similarly if duplications in the datasets are removed. In addition, they show that CodeBERT performs better. Since the example generation approaches we propose do not require any white-box information, we do not have access to any training data, but there is a potential to use deep learning approaches on interactions generated during the search for examples—if such interactions were stored.

As can be inferred from the presented work up to this point, relying on white-box information, such as a code corpus, is the most common approach to generating examples. However, we can find at least one black-box approach, by Gerdes et al. [39]. Gerdes et al. [39] is the most related approach to the example generation approaches in this thesis. Gerdes et al. also use a Property-based testing approach to generate and shrink examples of sequences of API operations [39]. However, to be able to generate relevant examples, the approach proposed by Gerdes et al. requires a formal specification of the behaviour of the API [39]. This is the main difference between our work and Gerdes et al., we do not require a specification of behaviour, it is the main goal of our approach is to discover behaviours of the API.

Chapter 5

Contributions

In this section, we discuss the contributions and the results of the work included in this thesis and how the papers included contribute to the achieved result. The overall contributions of this thesis are approaches and evaluations to automatically generate examples to advance the understanding of behaviours of software systems through their APIs. This understanding is both in the area of finding faults, i.e., understanding how the API can fail, and also in the area of understanding the actual behaviour of the API. An overview of how the results fit into the research process, the included papers and their contributions are shown in Figure 5.1.

5.1 Generating Examples of Fault-finding

Two of the included papers, Paper A and Paper B, contribute to research goal A: understand how we can automatically generate examples of fault-finding and specification compliance in our context (Web APIs).

Paper A presents an approach to automatically produce Property-based

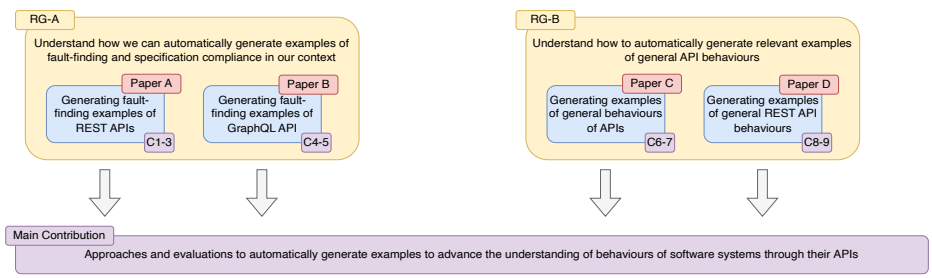

Figure 5.1: Overview of the contributions

generators from an OpenAPI specification. These generators are then used to generate examples of faults and examples where the SUT and OpenAPI specification is not aligned. The main contributions of Paper A are:

C1: A Property-based approach for automatic test generation of a REST API, given an OpenAPI specification.

C2: An approach to automatically generate stateful sequences of interactions for REST API:s given an OpenAPI specification.

C3: An approach to automatically finding the conformance of a REST API to an OpenAPI specification.

In Paper B we propose an approach to automatically produce Property-based generators based on a GraphQL schema, enabling automatic test generation for GraphQL APIs. While similar in its core idea to Paper A, considerations had to be made in order to support the graph nature of GraphQL. In addition, we propose a coverage metric for GraphQL schema. The main contributions of Paper B are:

C4: An automatic Property-based approach of GraphQL API test generation.

C5: An approach of measuring coverage of generated GraphQL queries.

Contributions C1-C5 enable the automatic generation of examples of faults and specification/schema miss-alignment for REST and GraphQL APIs.

5.2 Generating Examples of General API Behaviours

Paper C and Paper D contribute to research goal B: understand how to automatically generate *relevant* examples of general API behaviours. Paper C does this in the general API case, for example, in defining properties of state-changing behaviours. The main contributions of Paper C are:

C6: An approach to automatically generate *relevant* examples of actual system-under-test behaviours, as exposed by an API, without the need for a formal specification

C7: A set of general abstract meta-properties used to categorise API behaviours.

Paper D specializes the general properties from Paper D to the domain of REST APIs, in order to evaluate if practitioners find the generated examples relevant and the possibility of using the generated examples as a basis for test generation.

C8: An approach to generate *relevant* examples of actual REST API behaviours.

C9: A set of general REST API properties used to categorise REST API behaviours.

5.3 Software Artefacts

All the approaches of the contributing papers have publicly available replication packages. However, the main use of the software in the replication packages is to produce answers for the research questions posed in the papers, which makes them hard to use for a more general purpose.

In addition to the replication packages, we also publish a more engineered version, used in the evaluation of the final paper, Paper D. The final version can be found on GitHub[1].

C10: Proof-of-concept implementations of the proposed approaches.

5.4 Included Papers

This section contains a short description of the papers included in this doctoral thesis, including their abstract and a description of the contributions made. In addition, this section includes a description of my role in the contribution papers.

5.4.1 Personal Contributions

I have been the main author and driver of all the papers included in this thesis. The co-authors have provided feedback on my ideas, brainstormed new ideas and provided feedback on the paper manuscripts prepared by me. I have performed all the evaluations, with the exception of the execution and analysis of the focus group sessions of Paper D. In that evaluation, Robbert Jongeling contributed by being a co-moderator of the focus group sessions and

[1] `https://github.com/zclj/QuickREST`

performing an independent thematic analysis that was then merged with my own analysis, for the final evaluation.

The proof-of-concept software implementations have all been produced by me.

5.4.2 Paper A

Title: QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs
Authors: Stefan Karlsson, Adnan Čaušević, Daniel Sundmark
Status: Published at IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020
Abstract: RESTful APIs are an increasingly common way to expose software systems functionality and it is therefore of high interest to find methods to automatically test and verify such APIs. To lower the barrier for industry adoption, such methods needs to be straightforward to use with a low effort. This paper introduces a method to explore the behaviour of a RESTful API. This is done by using automatic property-based tests produced from OpenAPI documents that describe the REST API under test. We describe how this method creates artifacts that can be leveraged both as property-based test generators and as a source of validation for results (i.e., as test oracles). Experimental results, on both industrial and open source services, indicate how this approach is a low effort way of finding real faults. Furthermore, it supports building additional knowledge about the system under test by automatically exposing misalignment of specification and implementation. Since the tests are generated from the OpenAPI document this method automatically evolves test cases as the REST API evolves
Paper contributions: The contribution of this paper is a method for automatic test generation of Open API specified REST APIs based on property-based testing and a proof-of-concept implementation. In addition, the approach was evaluated on two industry case-studies, evaluating the fault-finding capability and the impact of different input generation configurations on API responses.

5.4.3 Paper B

Title: Automatic Property-based Testing of GraphQL
Authors: Stefan Karlsson, Adnan Čaušević, Daniel Sundmark
Status: Published at IEEE/ACM International Conference on Automation of Software Test (AST) 2021.

Abstract: In recent years GraphQL has become a popular way to expose web APIs. Several large industry companies have adopted GraphQL as a means of exposing their APIs. With this raise of adoption in industry, the quality of GraphQL APIs must be assessed, as with any part of a software system. However, there is currently a lack of methods to automatically generate tests to exercise GraphQL APIs.

In this paper we propose the first method of automatically producing GraphQL queries, with random arguments, to test GraphQL APIs. This is done using a property-based approach of creating a generator for queries based on the GraphQL schema of the system under test.

Our evaluation on a real world software system show that this approach is both effective, in finding real bugs, and efficient, the complete schema can be covered in seconds. In addition, we evaluate the fault finding capability of the method of known faults. The results show that the method is capable of finding faults that manifests as crashes or errors.

Paper contributions: This paper contributes with a method of automatically generate test cases of a recursive schema specification, GraphQL in this case, based on property-based testing. Since this is the first suggested approach for automatic GraphQL test generation we also propose a simple schema coverage criterion to evaluate the adequacy of the test generation. Finally, the approach is evaluated on a real industry-grade case-study, which showed that the approach is able to find real faults.

5.4.4 Paper C

Title: Exploring API Behaviours Through Generated Examples
Authors: Stefan Karlsson, John Hughes, Robbert Jongeling, Adnan Čaušević, Daniel Sundmark
Status: Published in Software Quality Journal (SQJ).
Abstract: Understanding the behaviour of a system's API can be hard. Giving users access to *relevant* examples of how an API behaves has been shown to make this easier for them. In addition, such examples can be used to verify expected behaviour or identify unwanted behaviours.

Methods for automatically generating examples have existed for a long time. However, state-of-the-art methods rely on either white-box information, such as source code, or on formal specifications of the system behaviour. But what if you do not have access to either? e.g., when interacting with a third-party API.

In this paper, we present an approach to automatically generate relevant

examples of behaviours of an API, without requiring either source code or a formal specification of behaviour.

Evaluation on an industry-grade REST API shows that our method can produce small and relevant examples that can help engineers to understand the system under exploration.

Paper contributions: This paper contributes an approach to automatically generate *relevant* examples of actual system-under-test behaviours, as exposed by an API, without the need for a formal specification and a set of general abstract meta-properties used to categorise behaviours. The approach is evaluated on an industry-grade case study, which shows that the method is capable of generating relevant examples, by the use of the proposed general meta-properties, that can provide knowledge of how an API behaves.

5.4.5 Paper D

Title: Exploring Behaviours of RESTful APIs in an Industrial Setting
Authors: Stefan Karlsson, Robbert Jongeling, Adnan Čaušević, Daniel Sundmark
Status: In submission. Pre-print available.
Abstract: A common way of exposing functionality in contemporary systems is by providing a Web-API based on the REST API architectural guidelines. To describe REST APIs, the industry standard is currently OpenAPI-specifications. Test generation and fuzzing methods targeting OpenAPI-described REST APIs have been a very active research area in recent years. An open research challenge is to aid users in better understanding their API, in addition to finding faults and to cover all the code. In this paper, we address this challenge by proposing a set of behavioural properties, common to REST APIs, which are used to generate examples of behaviours that these APIs exhibit. These examples can be used both (i) to further the understanding of the API and (ii) as a source of automatic test cases. Our evaluation shows that our approach can generate examples deemed *relevant* for understanding the system and for a source of test generation by practitioners. In addition, we show that basing test generation on behavioural properties provides tests that are less dependent on the state of the system, while at the same time yielding a similar code coverage as state-of-the-art methods in REST API fuzzing in a given time limit.
Paper contributions: This paper contributes an approach to automatically generate relevant examples of general REST API behaviours. In addition, we propose a set of general REST API behaviours to base generated examples

on. The output of the approach can be used as a source of understanding more about a REST API and as test-cases. The approach is evaluated in two different parts; first, we evaluate if practitioners deem the generated examples as relevant by performing multiple focus group sessions, and second, we evaluate the code-coverage of running the generated examples as test-cases. The first evaluation shows that practitioners do indeed consider the generated examples as relevant and helpful. The second evaluation shows that the generated examples provide comparable coverage to state-of-the-art REST API test generation approach. In total, the evaluation shows that we can provide practitioners with approaches that *both* provide test-cases and help in understanding an API better.

Chapter 6

Conclusions

Understanding the behaviour of complex software systems is hard. Any help provided to developers and testers in making this easier is valuable. We set out with the vision of providing approaches to support understanding complex software. We started by providing approaches closely related to the common goal of test generation of finding faults. This led to the understanding that, while examples of faults are good, to further the understanding of how a system behaves, examples of how API operations are very helpful.

The overall contributions of this thesis are approaches and evaluations to automatically generate examples to advance the understanding of behaviours of software systems through their APIs. The contributions of this thesis can both be used to enable fault-finding of Web APIs and to enable example generation to explore the behaviours of APIs. The contributions of automatic test generation for REST APIs and GraphQL APIs have contributed to the state-of-the-art in their respective area. The contributions following that work showed that test generation techniques can also be leveraged to generate examples deemed relevant by practitioners to further their understanding of the system.

Both the main related areas of test generation for Web APIs and black-box example generation are in their infancy. The combination of these concepts is the key idea making this thesis work novel. The future work enabled by the contributions of this thesis is (i) increasing the ability to cover a larger percentage of the SUT—we can not explore behaviours for what we cannot reach, (ii) providing a larger set of behaviours—different solution domains expose different general behaviours, and (iii) enabling the approach of example generation to be interactive.

In my view, the future of system-level test generation of Web APIs should consider two approaches; one is to spend the time given to cover all the code

and provide the best possible fault finding, and the other is to be interactive and provide feedback within seconds. Approaches requiring a larger quantity of time and resources can be used as part of continuous integration systems, perhaps running overnight. Interactive approaches can instead be used in the day-to-day development activity cycle of a developer—make a change to the software, and get close to instant feedback on the behavioural consequences of that change. In addition to providing fast feedback, an interactive approach can also provide the means for a user to "play with the system", tweaking different settings and running new experiments—all powered by test generation techniques.

In conclusion, we have set out with the goal of helping practitioners better understand their systems based on automatically generated artefacts, and we have shown that we can achieve that goal using our approach of automatically producing *relevant* examples.

Bibliography

[1] REST APIs must be hypertext-driven, 2008.

[2] Graphql foundation, 2020.

[3] How Did REST Come To Mean The Opposite of REST?, 2022.

[4] Alonso, Juan C. and Martin-Lopez, Alberto and Segura, Sergio and García, José María and Ruiz-Cortés, Antonio. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering*, 49(1):348–363, 2023.

[5] Alonso, Juan C. and Segura, Sergio and Ruiz-Cortés, Antonio. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1018–1030, New York, NY, USA, 2023. Association for Computing Machinery.

[6] A. "Arcuri. "many independent objective (mio) algorithm for test suite generation". In T. Menzies and J. Petke, editors, *Search Based Software Engineering*, pages 3–17, Cham, 2017. Springer International Publishing.

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

[8] A. Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1):3:1–3:37, Jan. 2019.

[9] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 219–230, New York, NY, USA, 2010. Association for Computing Machinery.

[10] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[11] Arcuri, Andrea. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering*, 23(4):1959–1981, Aug 2018.

[12] Arcuri, Andrea. Automated Black- and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software*, 38(3):72–78, 2021.

[13] Arcuri, Andrea and Galeotti, Juan P. SQL Data Generation to Enhance Search-Based System Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1390–1398, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Arcuri, Andrea and Galeotti, Juan P. Handling SQL Databases in Automated System Test Generation. *ACM Trans. Softw. Eng. Methodol.*, 29(4), jul 2020.

[15] Arcuri, Andrea and Galeotti, Juan P. Enhancing Search-Based Testing with Testability Transformations for Existing APIs. *ACM Trans. Softw. Eng. Methodol.*, 31(1), sep 2021.

[16] Arcuri, Andrea and Zhang, Man and Belhadi, Asma and Marculescu, Bogdan and Golmohammadi, Amid and Galeotti, Juan Pablo and Seran, Susruthan. Building an open-source system test generation tool: lessons learned and empirical analyses with EvoMaster. *Software Quality Journal*, 31(3):947–990, Sep 2023.

[17] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[18] T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing autosar software with quickcheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, April 2015.

[19] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.

[20] Atlidakis, Vaggelis and Geambasu, Roxana and Godefroid, Patrice and Polishchuk, Marina and Ray, Baishakhi. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations, 2020.

[21] Atlidakis, Vaggelis and Godefroid, Patrice and Polishchuk, Marina. Checking Security Properties of Cloud Service REST APIs. In *2020*

IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pages 387–397, 2020.

[22] C. Barnaby, K. Sen, T. Zhang, E. Glassman, and S. Chandra. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1353–1364, 2020.

[23] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[24] Belhadi, Asma and Zhang, Man and Arcuri, Andrea. Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Trans. Web*, aug 2023. Just Accepted.

[25] Briand, Lionel and Bianculli, Domenico and Nejati, Shiva and Pastore, Fabrizio and Sabetzadeh, Mehrdad. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. *IEEE Software*, 34(5):72–75, 2017.

[26] D. A. Burke and K. Johannisson. Translating Formal Software Specifications to Natural Language. In *Logical Aspects of Computational Linguistics*, pages 51–66, 2005.

[27] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 782–792, 2012.

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

[29] S. K. Chakrabarti and R. Rodriquez. Connectedness testing of restful web-services. In *Proceedings of the 3rd India Software Engineering Conference*, ISEC '10, page 143–152, New York, NY, USA, 2010. Association for Computing Machinery.

[30] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, 2000.

[31] K. "Claessen, N. Smallbone, and J. Hughes. "quickspec: Guessing formal specifications using testing". In G. Fraser and A. Gargantini, editors, *Tests and Proofs*, pages 6–21, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[32] Corradini, Davide and Pasqua, Michele and Ceccato, Mariano. Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2553–2564, 2023.

[33] Corradini, Davide and Zampieri, Amedeo and Pasqua, Michele and Viglianisi, Emanuele and Dallago, Michael and Ceccato, Mariano. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability*, 32(5):e1808, 2022.

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

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[36] T. Fertig and P. Braun. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, page 1497–1502, New York, NY, USA, 2015. Association for Computing Machinery.

[37] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, US, 2000.

[38] S. J. Fowler. *Production-Ready Microservices*. O'Reilly, 2016.

[39] A. Gerdes, J. Hughes, N. Smallbone, S. Hanenberg, S. Ivarsson, and M. Wang. Understanding Formal Specifications through Good Examples. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, Erlang 2018, page 13–24, 2018.

[40] Godefroid, Patrice and Huang, Bo-Yuan and Polishchuk, Marina. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 725–736, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Godefroid, Patrice and Lehmann, Daniel and Polishchuk, Marina. Differential Regression Testing for REST APIs. ISSTA 2020, page 312–323, New York, NY, USA, 2020. Association for Computing Machinery.

[42] A. Golmohammadi, M. Zhang, and A. Arcuri. Testing restful apis: A survey. *ACM Trans. Softw. Eng. Methodol.*, aug 2023.

[43] Golmohammadi, Amid and Zhang, Man and Arcuri, Andrea. .NET/C# instrumentation for search-based software testing. *Software Quality Journal*, Sep 2023.

[44] Gorschek, Tony and Garre, Per and Larsson, Stig and Wohlin, Claes. A Model for Technology Transfer in Practice. *IEEE Software*, 23(6):88–95, 2006.

[45] X. Gu, H. Zhang, and S. Kim. CodeKernel: A Graph Kernel Based Approach to the Selection of API Usage Examples. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–601, 2019.

[46] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 631–642, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Hatfield-Dodds, Zac and Dygalo, Dmitry. 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, 2022.

[48] A. Head, E. L. Glassman, B. Hartmann, and M. A. Hearst. *Interactive Extraction of Examples from Existing Code*, page 1–12. 2018.

[49] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. volume 32, pages 952–970, 2006.

[50] J. "Hughes. "quickcheck testing for fun and profit". In M. Hanus, editor, *Practical Aspects of Declarative Languages*, pages 1–32, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[51] J. Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, Cham, 2016.

[52] J. Hughes. How to specify it! In W. J. Bowman and R. Garcia, editors, *Trends in Functional Programming*, pages 58–83, Cham, 2020. Springer International Publishing.

[53] J. Hughes, B. C. Pierce, T. Arts, and U. Norell. Mysteries of dropbox: Property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 135–145, April 2016.

[54] S. Karlsson. Exploratory Test Agents for Stateful Software Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1164–1167, New York, NY, USA, 2019. Association for Computing Machinery.

[55] S. Karlsson, J. Hughes, R. Jongeling, A. Čaušević, and D. Sundmark. Exploring API behaviours through generated examples. *Software Quality Journal*, April 2024.

[56] S. Karlsson, A. Čaušević, and D. Sundmark. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, 2020.

[57] S. Karlsson, A. Čaušević, and D. Sundmark. Automatic Property-based Testing of GraphQL APIs. In *2021 ACM/IEEE 2nd International Conference on Automation of Software Test (AST)*, 2021.

[58] S. Karlsson, A. Čaušević, D. Sundmark, and M. Larsson. Model-based Automated Testing of Mobile Applications: An Industrial Case Study. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 130–137, 2021.

[59] Karlsson, Stefan. Towards Augmented Exploratory Testing, 2021.

[60] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Adding Examples into Java Documents. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 540–544, 2009.

[61] Kim, Myeongsoo and Corradini, Davide and Sinha, Saurabh and Orso, Alessandro and Pasqua, Michele and Tzoref-Brill, Rachel and Ceccato, Mariano. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1232–1243, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Kim, Myeongsoo and Xin, Qi and Sinha, Saurabh and Orso, Alessandro. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 289–301, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Kontio, J. and Lehtola, L. and Bragge, J. Using the focus group method in software engineering: obtaining practitioner and user experiences. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 271–280, 2004.

[64] P. Lamela Seijas, H. Li, and S. Thompson. Towards property-based testing of restful web services. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, page 77–78, New York, NY, USA, 2013. Association for Computing Machinery.

[65] B. Lavoie, O. Rambow, and E. Reiter. The ModelExplainer. In *In Proceedings of the 8th international workshop on natural language generation. 9–12*, 1996.

[66] Lin, Jiaxian and Li, Tianyu and Chen, Yang and Wei, Guangsheng and Lin, Jiadong and Zhang, Sen and Xu, Hui. foREST: A Tree-based Black-box Fuzzing Approach for RESTful APIs. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 695–705, 2023.

[67] L. W. Mar, Y.-C. Wu, and H. C. Jiau. Recommending Proper API Code Examples for Documentation Purpose. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 331–338, 2011.

[68] J. Martin and J. L. C. Guo. Deep api learning revisited. In *Proceedings of the 30th IEEE/ACM International Conference on Program Compre-*

hension, ICPC '22, page 321–330, New York, NY, USA, 2022. Association for Computing Machinery.

[69] A. Martin-Lopez. Automated Analysis of Inter-Parameter Dependencies in Web APIs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 140–142, 2020.

[70] Martin-Lopez, Alberto and Arcuri, Andrea and Segura, Sergio and Ruiz-Cortés, Antonio. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 231–241, 2021.

[71] Martin-Lopez, Alberto and Segura, Sergio and Müller, Carlos and Ruiz-Cortés, Antonio. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing*, 15(4):2342–2355, 2022.

[72] Martin-Lopez, Alberto and Segura, Sergio and Ruiz-Cortés, Antonio. A Catalogue of Inter-parameter Dependencies in RESTful Web APIs. In S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, editors, *Service-Oriented Computing*, pages 399–414, Cham, 2019. Springer International Publishing.

[73] Martin-Lopez, Alberto and Segura, Sergio and Ruiz-Cortés, Antonio. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings*, page 459–475, Berlin, Heidelberg, 2020. Springer-Verlag.

[74] Martin-Lopez, Alberto and Segura, Sergio and Ruiz-Cortés, Antonio. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 682–685, 2021.

[75] Martin-Lopez, Alberto and Segura, Sergio and Ruiz-Cortés, Antonio. 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*, ESEC/FSE 2022, page 408–420, New York, NY, USA, 2022. Association for Computing Machinery.

[76] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. volume 15, pages 78–86, 1998.

[77] V. O. Mittal and C. Paris. Generating examples for use in tutorial explanations: using a subsumption based classifier. In *In Proceedings of the 11th European Conference on Artificial Intelligence*, 1994.

[78] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 401–408, 2013.

[79] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How Can I Use This Method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890, 2015.

[80] Myeongsoo Kim and Saurabh Sinha and Alessandro Orso. Adaptive REST API Testing with Reinforcement Learning, 2023.

[81] D. G. Novick and K. Ward. What Users Say They Want in Documentation. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication*, SIGDOC '06, page 84–91, 2006.

[82] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What Programmers Really Want: Results of a Needs Assessment for SDK Documentation. In *Proceedings of the 20th Annual International Conference on Computer Documentation*, SIGDOC '02, page 133–141, 2002.

[83] P. V. P. Pinheiro, A. T. Endo, and A. D. S. Simão. Model-based testing of restful web services using uml protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*. SBC, 2013.

[84] Quiña-Mera, Antonio and Fernandez, Pablo and García, José María and Ruiz-Cortés, Antonio. GraphQL: A Systematic Mapping Study. *ACM Comput. Surv.*, 55(10), feb 2023.

[85] M. P. Robillard and R. DeLine. A field study of API learning obstacles. volume 16, pages 703–732, 2011.

[86] Robillard, Martin P. What Makes APIs Hard to Learn? Answers from Developers. volume 26, pages 27–34, 2009.

[87] Rosanna L. Breen . A Practical Guide to Focus-Group Research. *Journal of Geography in Higher Education*, 30(3):463–475, 2006.

[88] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, Dec 2008.

[89] Sartaj, Hassan and Ali, Shaukat and Yue, Tao and Moberg, Kjetil. Testing Real-World Healthcare IoT Application: Experiences and Lessons Learned, 2023.

[90] F. Shull, F. Lanubile, and V. Basili. Investigating reading techniques for object-oriented framework learning. volume 26, pages 1101–1118, 2000.

[91] N. SMALLBONE, M. JOHANSSON, K. CLAESSEN, and M. ALGEHED. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27:e18, 2017.

[92] Stallenberg, Dimitri and Olsthoorn, Mitchell and Panichella, Annibale. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 117–128, 2021.

[93] Stefan Karlsson and Robbert Jongeling and Adnan Causevic and Daniel Sundmark. Exploring Behaviours of RESTful APIs in an Industrial Setting, 2023.

[94] W. R. Swartout. GIST English Generator. In *In AAAI. 404–409*, 1982.

[95] Vargas, Daniela Meneses and Blanco, Alison Fernandez and Vidaurre, Andreina Cota and Alcocer, Juan Pablo Sandoval and Torres, Milton Mamani and Bergel, Alexandre and Ducasse, Stéphane. Deviation testing: A test case generation technique for GraphQL APIs. In *11th International Workshop on Smalltalk Technologies (IWST)*, pages 1–9, 2018.

[96] Viglianisi, Emanuele and Dallago, Michael and Ceccato, Mariano. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, 2020.

[97] Wu, Huayao and Xu, Lixin and Niu, Xintao and Nie, Changhai. Combinatorial Testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 426–437, New York, NY, USA, 2022. Association for Computing Machinery.

[98] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving Semantics-Aware Fuzzers from Web API Schemas, 2021.

[99] Zetterlund, Louise and Tiwari, Deepika and Monperrus, Martin and Baudry, Benoit. Harvesting Production GraphQL Queries to Detect Schema Faults. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 365–376, 2022.

[100] Zhang, Man and Arcuri, Andrea. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[101] Zhang, Man and Arcuri, Andrea and Li, Yonggang and Xue, Kaiming and Wang, Zhao and Huo, Jian and Huang, Weiwei. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan, 2022.

[102] Zhang, Man and Belhadi, Asma and Arcuri, Andrea. JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115, 2022.

[103] Zhang, Man and Belhadi, Asma and Arcuri, Andrea. JavaScript SBST Heuristics to Enable Effective Fuzzing of NodeJS Web APIs. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[104] Zhang, Man and Marculescu, Bogdan and Arcuri, Andrea. Resource-Based Test Case Generation for RESTful Web Services. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1426–1434, New York, NY, USA, 2019. Association for Computing Machinery.

[105] Zhang, Man and Marculescu, Bogdan and Arcuri, Andrea. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering*, 2021.