Automatic Property-based Testing of GraphQL APIs

Stefan Karlsson
ABB AB, Mälardalen University
Västerås, Sweden
Email: stefan.l.karlsson@{se.abb.com, mdh.se}

Adnan Čaušević
ABB AB
Västerås, Sweden
Email: adnan.causevic1@se.abb.com

Daniel Sundmark
Mälardalen University
Västerås, Sweden
Email: daniel.sundmark@mdh.se

2021 IEEE/ACM International Conference on Automation of Software Test (AST) | 978-1-6654-3567-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/AST52587.2021.00009

***Abstract*—In recent years, GraphQL has become a popular way to expose web APIs. With its raise of adoption in industry, the quality of GraphQL APIs must be also assessed, as with any part of a software system, and preferably in an automated manner. However, there is currently a lack of methods to automatically generate tests to exercise GraphQL APIs.**

In this paper, we propose a method for automatically producing GraphQL queries to test GraphQL APIs. This is achieved using a property-based approach to create a generator for queries based on the GraphQL schema of the system under test.

Our evaluation on a real world software system shows that this approach is both effective, in terms of finding real bugs, and efficient, as a complete schema can be covered in seconds. In addition, we evaluate the fault finding capability of the method when seeding known faults. 73% of the seeded faults were found, with room for improvements with regards to domain-specific behavior, a common oracle challenge in automatic test generation.

***Index Terms*—property-based testing, graphql, automated testing**

I. Introduction

GraphQL[1] is a technology publicly introduced by Facebook in 2015 [1]. GraphQL provides a method to state declarative, compositional, and strongly typed queries for Web APIs. Since its inception, GraphQL has become a popularly used alternative to RESTful APIs. Several large internet services provide parts of their APIs via GraphQL [2]. Notable examples are GitHub[2], GitLab[3], Shopify[4], Facebook[5] and Pinterest[6]. Industry adopters, such as Shopify and GitHub, express benefits of moving to GraphQL [3], [4] and some of those benefits are supported in the literature. For example, Brito et al. report that in their experiments the returned payload size was reduced by up to 99% by porting several RESTful client calls to GraphQL [5]. In addition, in a controlled experiment subjects spent less time on client implementation using GraphQL versus REST [6]. Some industry adopters, GitLab for example, state that GraphQL will be the primary interaction model for their APIs going forward [7].

RESTful APIs [8] have been, and still are, a common approach to expose APIs of internal and external software systems. As a result of this, methods to automatically test RESTful APIs have been proposed. Some recent examples of methods that automatically produce tests, targeting RESTful APIs, are EvoMaster [9], RESTler [10], QuickREST [11] and RESTTESTGEN [12]. However, with the increased usage to expose Web APIs using GraphQL, there are new challenges in testing Web APIs, which calls for new testing methods in addition to Web APIs using REST.

```
{ person { pet { name } } }
```

Fig. 1. Query example in GraphQL

```
{"data": {
   "person": {
     "pet" : {
       "name": "Rex"}}}}
```

Fig. 2. Example query result

GraphQL and REST expose Web APIs in fundamentally different ways. GraphQL provides the clients with a query language and REST typically provides operations on URL encoded resources. Other than exposing Web APIs commonly using JSON as format, there is not much in common between the two. GraphQL enables a client to query only the data it requires, where the responses from a REST API depends on the server implementation, the client has no control of the shape of the response and what to fetch. The client is provided with a typed schema, which REST does not provide, of the possible entities to query and their fields. Figure 1 shows an example query expressed in GraphQL and Figure 2 shows the result from executing such a query. Note how only the data explicitly asked for is returned and the layout of the result corresponds to the client's query. This is in contrast to REST, where querying the same information might require several operations.

Given the benefits of GraphQL and its rise in popularity in industry, it is of high interest to find methods to automatically produce test cases for GraphQL APIs, as has been done for RESTful APIs. Such methods must be able to produce GraphQL queries, conforming to the typed schema, covering the complete schema under test, and handle the graph nature of the schema, with the possibility of recursive types.

In this paper, we introduce a method for automated testing of GraphQL APIs. We build upon property-based testing tech-

[1]https://graphql.org/
[2]https://github.com/
[3]https://about.gitlab.com/
[4]https://www.shopify.com/
[5]https://www.facebook.com/
[6]https://www.pinterest.com/

978-1-6654-3567-3/21/$31.00 ©2021 IEEE
DOI 10.1109/AST52587.2021.00009

niques to randomly generate test cases in the form of GraphQL queries. The generated queries are then automatically executed on the API under test and the result is automatically verified to not have caused any server error and to conform to the schema. One main challenge in property-based testing is to produce generators for the specific system under test. A main contribution in this paper is a method for automatically creating a generator for GraphQL queries conforming to the schema, handling the recursive nature of the schema (which results in a potentially infinite search-space), to produce valid queries of increasing complexities in a timely fashion. In addition, we propose a method of how to measure the coverage of a GraphQL schema achieved by test cases. In this paper, we contribute, to the best of our knowledge, with the first proposed method of how to fully automatically produce test cases, and measure schema coverage, for GraphQL APIs. We also present the findings of evaluating the method on GitLab, a large open-source, industry-grade, software system, where we found and reported several new bugs. We have evaluated both the fault finding capability of the method as well as the coverage of the schema for the produced test cases.

II. Background

In this section, we present an overview of the GraphQL specification[7] and of property-based testing. For GraphQL, we restrict ourselves to the parts of the specification that are of relevance to understanding this paper and refer the interested reader to the full specification for a more in-depth view.

A. GraphQL

GraphQL is a query language for APIs. It was internally developed by Facebook, starting in 2012, and was open-sourced in 2016. Today, GraphQL is curated by the GraphQL Foundation who aims at increasing its adoption [1]. GraphQL provides a language that allows clients to query an API for only the data needed by the client. This is both more efficient and more predictable for the client since the client knows the structure of the returned query result. This property both reduces the size of the result and the number of network calls. It also lessens the complexity of the client implementation, since the client does not need to keep track of intermediate query results, as is common in a RESTful API.

GraphQL is, as its name hints to, graph-based. This means that a query follows references of queried resources and a client can thus query a graph of related resources. To enable clients to inspect what resources are available to query, GraphQL has a type system that is accessible to clients. Clients can use query introspection to access meta-data about the queries, types, and fields available. Figure 3 shows an example result of an introspection query that has queried for the available types in the schema. The result includes the meta-data information of the `Project` type and its fields.

A query in GraphQL defines a set of *fields* of *objects* to be queried. In the example of Figure 1, the root level object

[7]https://spec.graphql.org/June2018/


```
{"data": {
    "__schema": {
      "types": [{
          "kind": "OBJECT",
          "name": "Project",
          "description": null,
          "fields": [
            {
              "name": "description",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null}},
            {
              "name": "id",
              "description": null,
              "args": [],
              "type": {
                "kind": "NON_NULL",
                "name": null,
                "ofType": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null}}},
            {
              "name": "members",
              "description": null,
              "args": [],
              "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {
                  "kind": "OBJECT",
                  "name": "User",
                  "ofType": null}}},
            {
              "name": "name",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null}},
            {
              "name": "owner",
              "description": null,
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "User",
                "ofType": null}}]}]}}}
```


Fig. 3. Project example schema type

is `person`. The `person` object contains a field, `pet` which also is an object. From `pet` the `name` field is selected. In this example there were two related objects (`person` and `pet`) and one scalar field (`name`).

There are several *scalar* field types defined in the GraphQL specification such as *Int, Float, String, Boolean* and *ID*, where *ID* is a special type representing a unique identifier in the business domain. In addition to the scalar types, GraphQL also defines an enumeration type and a list type.

An object field can have arguments. These can be used to, for example, limit a query selection in a search. Imagine an object, `Project`, that contains the field `name`. For the client to be able to select from which project the name should be selected from, the schema can define an argument such as `path`. The client would then execute a query as in:

```
{project(path:"projects/project1"){name}}
```

B. Property-based Testing

Property-based testing (PBT) is today synonymous with the random testing introduced by Claessen et al. in the tool QuickCheck [13]. PBT is a testing technique that aims to verify the invariant properties of a system under test (SUT). Test cases are randomly produced and the PBT tools then try to find random values that prove the invariant wrong. If such an example is found it is a failing test case. Random values

```
{:object/Query
 [:map
  [:fields
   [:map
    ["person"
     [:map
      [:type [:ref :object/Person]]]]]]]

 :object/Person
 [:map
  [:fields
   [:map
    ["pet" {:optional true} [:ref :object/Pet]]]]]

 :object/Pet
 [:map
  [:fields
   [:map
    ["name" [:gql.scalar/string]]]]]}
```

Fig. 4. Illustrative example of specifications generated from a schema.

are produced by invoking *generators*. A PBT library typically delivers generators for basic scalar types, such as integers, strings and booleans, and generators that are the combination of other generators. In this way, a user can combine generators to randomly produce relevant values for the domain under test.

Compared to example-based tests, PBT requires a shift in thinking. Instead of thinking of the specific output produced to a specific example input, we must instead think in invariants. For example, imagine that we test a search function. The search function requires a search string as input and returns a list of items that match the search string. Instead of using a predefined search string with a predefined outcome, we instead use a string generator for the input string and formulate the invariant properties of the resulting list of items. Such properties include (1) the name of the resulting entities should always contain the search string as a substring in their name (2) the returned entities should always conform to a given format and (3) the function should always return a successful result, i.e., not crash. With this generator and properties, a tool like QuickCheck can generate any number of test cases to try and find cases where the properties are proven wrong.

In the implementation of our proposed method, we have leveraged a property-based approach and defined a generator to randomly produce GraphQL queries and provided properties to verify schema conformance and that no GraphQL errors are returned by the SUT.

III. Proposed Method

Our proposed method automatically generates tests based on a graph-based schema of possible operations. In our case, we have applied our method to GraphQL schema. As input, the method requires a GraphQL schema. Optionally, value generators can be provided allowing the user to express any required format not included in the schema for a specific schema type.

Tests are generated as a query to be executed on the SUT and an oracle, a way to assert the results of the test, that verify invariant properties of the SUT. The queries are randomly generated based on the available *objects* and *fields* in the schema. Since a schema *object* can contain both *scalar* and *object* fields, the randomization of query fields is recursively resolved to be able to select fields available to the current object being processed. This means that starting with a root object we randomly select fields from that object. Any fields that are themselves an object will have nested fields that need to be resolved in the next recursion. This recursion continues until there are no more *object* fields to resolve or a recursion limit is hit.

Since the method targets cyclic graph schemata the recursion depth is taken into consideration. We do this by allowing the user to control the number of fields selected for each depth of the query generation. This is explained in more detail in Section III-B.

Query fields can require input arguments, for example, to limit a selection by a search string. The schema expresses the types of these arguments and our test generation process uses generators for the specific types. The argument generators produce random values to be used as arguments in the query. A sizing parameter is used to control the size of generated arguments. This allows for our generators to start with the generation of simple arguments, such as an empty string, and continuously increase the complexity of the argument parameter value. The reasoning being to fail fast on arguments that are faster to produce.

Depending on the given parameters, the size of the generated queries will vary. This allows a user to start with the generation of smaller queries and, as the confidence in the API increases, increase the size of the queries. It should also be noted that in the case of GraphQL the SUT might have restrictions on the complexity of the queries. Such restrictions are put in place to protect the SUT from spending too many resources on executing the query. It is then necessary to be able to control the size of the queries to produce queries that pass the complexity validation and are executed.

Figure 5 presents an overview of the proposed method including the following steps:

1) From the *Schema*, generate specifications of the types in the *Schema*.
2) From the *Schema*, create a generator capable of randomly generating a list of query objects with fields.
3) Generate n number of queries.
4) Given the generated query data, transform the query in GraphQL format.
5) Execute the queries on the SUT.
6) Evaluate the result in a property-based fashion

A. Creation of type data specifications

To verify that the returned result of executing a GraphQL query conforms to the expected result, we need specifications of the types and their relations from the given schema. As a one-time operation, we generate specifications that correspond to the objects and their fields in the schema. Figure 4 shows a specification for the schema example queried in Figure 1.

The *Query* (1), describing the root level entities of a valid query, *Person* (9) and *Pet* (15) are all specified as *Objects* since they are composites and contains *fields*. The specification contains references for fields that are referring to another object as its type, such as the *Person* object referencing a

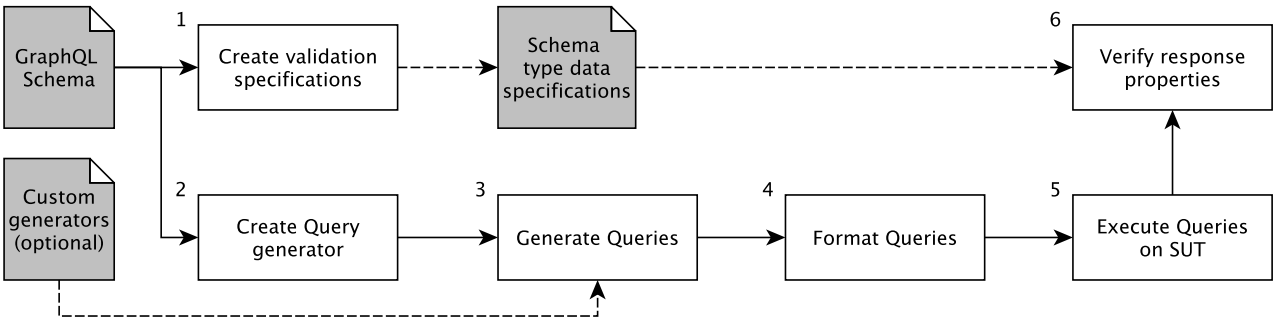

Fig. 5. Overview of the automatic test case generation method

Pet as the type of its field *pet* (13). The *Pet* object shows an example of a field, *name*, being of a scalar type (19). This specification is then used to verify that returned data from a query, as in Figure 2, conforms to the Schema of the SUT.

The input of this step is a GraphQL Schema and the output are specifications of all object types and their fields defined in the schema.

B. Creation of the query generator

To be able to generate GraphQL queries, a generator is needed. Such a generator can be generic in that it can generate queries given any schema conforming to the GraphQL specification.

Queries in GraphQL can be recursive and cyclic. This must be taken into consideration when implementing a query generator. In our method, we propose to instead of making a depth recursive generator, make an iteration based generator. As mentioned in II-A, a query schema contains objects with fields where fields can be scalar types or objects. A random generator must then first randomly generate the object to start with of the available query types, i.e., the root object. The fields to randomly generate must be selected from the available fields of the root object. This means that the generator must have the context of the current object available to make the field selection. We can then either do a depth-first recursion of expanding all fields that are objects, and for the new fields that are objects expand those further, etc., or we can make an iterative flat process where only the currently generated fields are expanded.

During the development of the method we found that having a schema with a low level of recursions in the types, a depth-first approach could be used. However, with a larger and more recursive schema, it did not scale and the iterative approach described generates queries in a more controlled and timely fashion.

C. Query generation

Figure 6 shows an example of the flat list of query nodes produced by the generator. The root of the query is the `"projects"` in line 1. This object has a generated argument, (3-9), with the generated value of ”7x8Z” (9). There are

```
1  [{:name "projects",
2    :generation 0,
3    :args
4    [{:name "id",
5      :type
6      {:kind "NON_NULL",
7       :name nil,
8       :ofType {:kind "SCALAR", :name "ID", :ofType nil}},
9      :value "7x8Z"}],
10   :fields
11   [{:name "description",
12     :args [],
13     :type {:kind "SCALAR", :name "String", :ofType nil}}
14    {:name "members",
15     :args [],
16     :type
17     {:kind "LIST",
18      :name nil,
19      :ofType {:kind "OBJECT", :name "User", :ofType nil}}}],
20   :type {:kind "OBJECT", :name "Project", :ofType nil},
21   :id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a"}
22  {:name "description",
23   :generation 1,
24   :args [],
25   :type {:kind "SCALAR", :name "String", :ofType nil},
26   :field-id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a",
27   :from-type "Project"}
28  {:name "members",
29   :generation 1,
30   :args [],
31   :fields
32   [{:name "projects",
33     :args [],
34     :type
35     {:kind "LIST",
36      :name nil,
37      :ofType {:kind "OBJECT", :name "Project", :ofType nil}}}
38    {:name "name",
39     :args [],
40     :type {:kind "SCALAR", :name "string", :ofType nil}}],
41   :type
42   {:kind "LIST",
43    :name nil,
44    :ofType {:kind "OBJECT", :name "User", :ofType nil}},
45   :id #uuid "9c358b36-b5f8-42c6-8b80-aef3cac6f5f1"
46   :field-id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a",
47   :from-type "Project"},
48  {:name "name",
49   :args [],
50   :type {:kind "SCALAR", :name "String", :ofType nil},
51   :field-id #uuid "9c358b36-b5f8-42c6-8b80-aef3cac6f5f1",
52   :from-type "User"}
53  {:name "projects",
54   :args [],
55   :type
56   {:kind "LIST",
57    :name nil,
58    :ofType {:kind "OBJECT", :name "Project", :ofType nil}},
59   :field-id #uuid "9c358b36-b5f8-42c6-8b80-aef3cac6f5f1",
60   :from-type "User"}]
```

Fig. 6. Randomly generated schema nodes

two fields randomly generated, `"description"` (11) and `"members"` (14). For each generated node which is an object, we attach a unique id (line 21 for `project`) and for each generated field we attach a *field-id* which has the same value as the object the field belongs to. By doing this we can later create a tree from the flat list of nodes.

During generation, the flat list is iterated and each object or field which is processed gets a `generation` value attached. The generation value is used to keep track of which nodes have already been processed and which nodes need processing in the current iteration.

To restrict the number of nodes generated, the depth of the resulting query, a sizing parameter is used. It is common for property-based random generators to have a size parameter, without one we have no control of the different sizes of random values produced by the generator. As an example, the library provided *string* generator uses the size parameter to produce random strings with a length proportional to the size parameter.

When a PBT library generates test cases, it will start with a small size parameter value and increase it. The reason is to first try to find faults with simpler values before generating more complex values, as a simple example is preferred for reproducibility of a failure. Our method uses the size parameter in two ways. First, for any argument generation, such as the argument value in Figure 6 line 9 (”7x8Z”). This means that any argument value we produce will start with small values and increase the complexity as the random testing increases the size parameter. Secondly, we use it to restrict the numbers of query nodes generated in the flat list. This is done by controlling the number of generating iterations of query nodes to be the minimum of the current *size* and the *maximum-allowed-iterations*. This means that our query generator starts with the generation of a few number of nodes and increases the number of nodes as the size parameter increases, but can be capped at a `maximum-allowed` value to restrict the depth. Expansion of new object fields will stop at this generation size. However, in many cases the depth will be randomly controlled by the number of fields generated for objects, if only scalar fields are generated for the current generation, there is no more field generation to perform for the next generation.

D. Query creation

In this step of the method, the tool uses the flat list of randomly generated query nodes and transforms them into a tree. This is done by using the object id to field-id connection produced in the query generation. Figure 7 shows a graphical representation of the relationships between the flat list of nodes and the query graph. This process starts from the last generation down to the first, i.e., the last fields to be generated are the bottom leafs of the tree and those should first be inserted into their containing object as the containing object can be a leaf to another object. By this method, we end up with a tree as in Figure 8. In Figure 8 we can see the flat list from Figure 6 transformed into its tree form.

In this step, some cleaning of the flat list is also done before the tree is created. This process is simplified by using the flat list approach, which we propose, instead of letting the generator generate trees. An example of a cleaning operation is to remove any fields from the last generation with the type of *object*. To be a valid query, an object must have a field

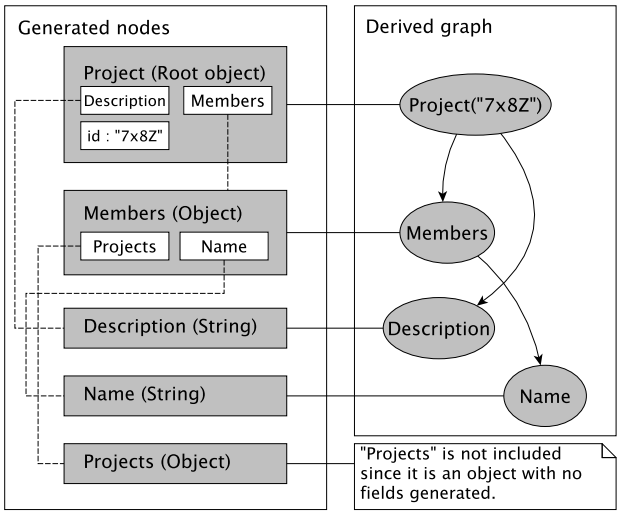

Fig. 7. Relationships of generated nodes to the query graph. Dashed lines are the object-id to field-id relation. Solid lines are the relation of the nodes to the query. Arrows are the relations in the query graph.

selection but the generation limit was hit before these were processed and they should be removed from the resulting tree.

With the resulting tree structure, we make the final transformation of turning the tree data structure into an actual executable GraphQL query. This transformation recursively processes the tree and composes the name of the fields with the required delimiters in the resulting string. Figure 9 shows the final executable output of the tree data structure of the query in Figure 8. It is notable to mention, this method creates test cases that are human-readable. In addition to the automatic test execution of the generated query, the resulting query can be executed manually or stored as input to bug reports.

E. Execution and result verification

As this is a property-based method we need both a generator for our test data and properties to verify the results of executing the test data. We have created the query generator as previously described. Running test cases are done by automatically letting the generator generate a query, execute the query on the SUT, and verify the stated properties of the result.

We suggest three properties generic to GraphQL test generation, which were also used in our evaluation. The first one is to verify the HTTP status code returned by executing the query. HTTP has well-defined status codes. For example, a ”200” status code is the ”OK” status code, a success, while ”500” means ”Internal Server Error”. Thus a status code other than ”200” would result in a failed test. However, GraphQL is not tied to HTTP and can produce errors, contained in the returned result, while still producing an ”OK” HTTP status code. The second property verifies the lack of GraphQL errors in the returned result. The third suggested property is to verify that the resulting data returned conforms to the given schema, i.e if the query is for a *Project* with a field of *name*, this property

```
[{:name "projects",
  :type {:kind "OBJECT", :name "Project", :ofType nil},
  :id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a",
  :generation 0,
  :args
  [{:name "id",
    :type
    {:kind "NON_NULL",
     :name nil,
     :ofType {:kind "SCALAR", :name "ID", :ofType nil}},
    :value "7x8Z"}],
  :fields
  [{:generation 1,
    :name "description",
    :args [],
    :fields [],
    :type {:kind "SCALAR", :name "String", :ofType nil},
    :field-id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a",
    :from-type "Project"}
   {:name "members",
    :generation 1,
    :args [],
    :fields
    [{:name "name",
      :args [],
      :type {:kind "SCALAR", :name "String", :ofType nil},
      :field-id #uuid "9c358b36-b5f8-42c6-8b80-aef3cac6f5f1",
      :from-type "User"}],
    :type
    {:kind "LIST",
     :name nil,
     :ofType {:kind "OBJECT", :name "User", :ofType nil}},
    :id #uuid "9c358b36-b5f8-42c6-8b80-aef3cac6f5f1",
    :field-id #uuid "c02efeee-f161-42b4-b589-f2ca34692e8a",
    :from-type "Project"}]}]
```

Fig. 8. Composed tree of randomly generated schema nodes

```
{projects(id: "7x8Z"){description members{name}}}
```

Fig. 9. Randomly generated example in GraphQL

verify that the response returned include only valid fields for a project, name, and that the name scalar contains a value of the correct scalar type, a string.

Being a property-based method a user can always add more properties that better verify the specific domain properties of the SUT.

IV. Evaluation

In this section, we provide an evaluation of the proof-of-concept (PoC) method implementation applied both to a real world software system, and a controlled example application, with the goal of evaluating the schema coverage and fault finding capability. The PoC was implemented in the Clojure programming language [14] and is available as a replication package[8]. We used the PBT library TestCheck[9] to build generators, and to run the test cases. Basic generators from TestCheck was used as primitives to create the more complex GraphQL specific generators based on the schema. We implemented a black-box version of the method, meaning we did not assume any access to the source code or any other internal artifact of the SUT. The only input required was the GraphQL Schema.

We do not provide any comparison with tools targeting REST APIs. While both REST and GraphQL expose Web APIs, they are very different methods and thus any tool specific to one is not applicable to the other.

For the purpose of this evaluation, we formulated the following research questions:

- **RQ1**: What schema coverage can be achieved by the method?
- **RQ2**: What is the fault finding capability of the method?

The focus of the first research question is to measure if the method can cover a complete schema and the effect the inputs to the method have on the coverage of the schema. We additionally set out to evaluate the efficiency of generating achievable coverage.

The second research question evaluates if the method can find both real faults and injected faults by generating random queries and input arguments executed on the system under test.

A. Studied Cases

We evaluated our method using two different cases. One example application where we could control the specific errors injected and one real industry system, GitLab.

1) Example application: We implemented a small example application with a small GraphQL Schema, shown in Figure 3. The purpose of the application was as a SUT with known faults, to allow for evaluation of the fault finding capability of the method.

2) GitLab: For our evaluation of an industry system we have used GitLab as the system under test. GitLab is a product that provides tools for the complete DevOps lifecycle. This includes tools such as code version control, issue handling, and build-pipelines. GitLab have been recognised as a leader in cloud-native continuous integration by Forrester[10] [15].

GitLab is provided both as a Software-As-A-Service offering and an on-premise product. The product also comes in two different versions; the Enterprise Edition (EE) and the Community Edition (CE) which differ in features and support options. GitLab CE is available as open-source software, has a GraphQL API, is a real industry software with a large usage, and with the combination of being able to run on-premise, it is a good industry case study for our method. In addition, GitLab is a large system with close to 2M lines of code.

Many services are using GraphQL APIs. However, to be able to run thousands of test cases, a hosted service is not a viable option. Industry grade services use rate limits and denial-of-service protections to not allow clients to abuse the service. Therefore we need to run our experiments in a local setup with software that is available to do so.

B. Measuring Schema Coverage

The proposed method contains a random exploration, since, given the recursive nature of a GraphQL schema, the potential search space is infinite. Using a random based method we have to ask ourselves, is the complete schema covered while tests are randomly generated? To evaluate the effectiveness of our approach in generating tests that cover the complete schema, we need to define a coverage criterion.

For GraphQL APIs, observed in a black-box approach, we propose a schema coverage criterion. This coverage criterion is based on the *objects* and their *fields* defined for all the types in

[8]https://github.com/zclj/replication-packages/tree/master/GraphQL
[9]https://github.com/clojure/test.check
[10]https://go.forrester.com

the schema. To be effective, generated test cases should cover all the available objects and fields defined in the schema.

We propose that each field defined in the schema is expressed in a tuple with its containing object. We use the tuple format to express all the unique pairs since field names can be the same on different objects. This method results in the set $\{[Object_t, Field_n]\}$ containing the tuples of all fields, n, of the objects of type t in the schema. For example, imagine a *Person* object with the available fields of *name* and *age* and a *Pet* object with a field of *name*. The resulting tuple set for the schema in this example would then be represented as $\{[Person, name][Person, age][Pet, name]\}$. During test generation, the same kind of tuples are created. A query of {`person {name age}`} would cover the tuples $\{[Person, name][Person, age]\}$. To measure which parts of the schema that have been tested by the generated queries, we observe how many of the possible tuples of the schema are present in the generated query data. Tuples in the schema which are not present in the generated test are thus not tested and not covered.

This coverage metric can be used both online and offline, meaning that the generated tests can be evaluated on how much of the schema is covered, and once executed, the results from the system under test can be converted into tuples and assessed for coverage. In summary, since no established coverage metric exists for black-box GraphQL test evaluation, we propose to start with a basic and intuitive one.

C. Experimental Setup and Method

We ran GitLab with the official Docker[11] image[12] and the official GitLab GraphQL schema. The coverage and fault finding experiments were executed on commodity developer hardware, a MacBook Pro, 2,9 GHz i9, 16 GB RAM.

1) RQ1: schema coverage: To measure the schema coverage of the generated queries we created object-field tuples of both the GitLab GraphQL schema and the generated query data. We used the tuple format described in Section IV-B to produce all the unique pairs. The goal of the generated query data is to cover all such tuples and in doing so get coverage on all available fields in the schema.

We generated *n* queries with a *max-field* limit of *m*. The field limit controls the max number of fields to be generated for each object. A low number will decrease the likelihood of a deep query and a high number will increase the likelihood of a deep query.

Each set of parameter values, *n* and *m* was executed 30 times. The average coverage was reported but also the aggregated coverage over all 30 iterations. The reason for measuring the aggregated coverage is due to the possible complexity limits of the SUT. Very deep queries have high coverage, but the query might not be allowed to execute on the SUT. The aggregated coverage then allows evaluating if high coverage is achieved with smaller queries but with a higher number of executions.

[11]https://www.docker.com/

[12]https://docs.gitlab.com/omnibus/docker/

RQ2: Fault finding capability To evaluate the fault finding capabilities of our method, we applied it to GitLab. To evaluate fault finding, queries must be executed on a running service, a local GitLab instance in our case.

The full process in Figure 5 was followed. Queries were generated from the schema, formatted, and executed on the SUT, and properties of the results returned were verified. A property was defined to check for the presence of "Internal server errors", i.e, status code 500, in the returned result. Any such returned result can be the effect of an unhandled crash on the server and is categorized as a fault, as the result requires further attention. In addition, a property verifying that the result payload conforms to its specific schema type was also defined.

In addition to the GitLab evaluation, we sought to evaluate which type of faults the method can find. For this purpose, we used an example application with seeded faults. The GraphQL schema of the application contained the types `Project` (with fields id, name, description, owner, members) and `User` (with fields id, name, age, projects). The `owner` and `members` of a project are references to entities of the `User` type and the `projects` field of a `User` is a reference to a `Project`, making the schema cyclic.

Each object reference in the GraphQL query will be handled by a *resolver*. A resolver contains the logic of finding the data for that query node. For example, a resolver for `Project` with an `Id` argument could connect to a database and select the project with the given id, fulfilling that part of the query. To handle the example schema we needed four resolvers. The logic for those resolvers was the target for the injected bugs. The injected bugs were reviewed by senior industrial developers and deemed as plausible.

In total 15 bugs were injected. The bugs were selected to cover all the resolvers and to cover the bug categories of input validation, logic error resulting in a crash, logic errors resulting in the wrong result, and return type errors.

- 3 input validation bugs, for the project resolver.
- 4 logic bugs causing a crash, one for each resolver
- 4 bugs where filtering is done using the wrong field, i.e., trying to find an entity by name instead of by id when given the value for the id. One for each resolver.
- 4 bugs returning the wrong type, i.e., returning a list where a single value is expected. One for each resolver.

We executed the generated test cases on our working implementation without finding any problems, it was also reviewed with no known problems. We then injected each of the described bugs, one at a time. With a planted bug in place, test cases were then generated and executed as with the setup previously described.

D. Results

In this section, we provide the results of the case studies performed on GitLab and our example application.

1) RQ1: Schema coverage: It turns out that our method can produce queries with full coverage of the fields defined in the schema. As can be seen in Table I, a high coverage can be

achieved both by running shallow queries a large number of times or deeper queries a fewer number of times. However, before deciding on which set of parameters to use, the SUT must be considered. If the complexity limits of the SUT will not allow deep queries a larger number of shallow queries can be generated for the same achieved coverage. In addition, deep queries will carry a larger cost in processing. It will be more resource-intensive to generate a GraphQL query string, serialize and send to the server for deserialization than with smaller queries.

With the results from Table I we can further examine at which iteration count the full coverage was achieved. Each configuration of the parameter values was executed until 100% coverage was achieved for 30 times. The average iteration count required for 100% coverage is shown in Table II. This result, in combination with Table I, gives more information about the trade-off between size, field selection size, execution time and coverage to consider for the specific SUT.

Full schema coverage can be achieved by randomly generating test cases, both for many smaller queries or with fewer larger ones. However, there is a trade-off between query size, complexity properties of the system under test and efficiency.

TABLE I
SCHEMA COVERAGE RESULT ON GITLAB SCHEMA

size	max-limit	avg.	total	time (s)
10	1	3.18%	20.58%	0.03
100	1	14.04%	49.59%	0.62
1000	1	36.95%	87.24%	9.12
10000	1	73.33%	100.00%	94.83
10	2	6.56%	38.89%	0.04
100	2	28.94%	85.60%	1.26
1000	2	64.87%	100.00%	19.32
10000	2	97.96%	100.00%	199.65
10	3	12.00%	54.94%	0.07
100	3	49.08%	99.59%	2.56
1000	3	90.62%	100.00%	40.97
10000	3	100.00%	100.00%	416.31
10	4	20.86%	77.16%	0.11
100	4	71.51%	99.79%	5.73
1000	4	99.29%	100.00%	93.57
10000	4	100.00%	100.00%	931.63

TABLE II
NUMBER OF ITERATIONS, ON AVERAGE, REQUIRED FOR 100% SCHEMA COVERAGE RESULT ON GITLAB SCHEMA

size	max-limit 1	max-limit 2	max-limit 3	max-limit 4
100	-	-	61.43	18.0
1000	-	26.63	6.57	2.2
10000	26.0	3.77	1.13	1.0

2) RQ2: Fault finding capability: Regarding the evaluation of the example application with seeded faults, in total, 11 of the 15 bugs were found (73%). Bugs were found in all of the four resolvers of the domain entities, this means that the generated test queries covered all resolvers.

All 3 input validation bugs were found as well as the 4 logic bugs resulting in a crash. These bugs all failed the crash detecting property and thus failed the generated test cases. The 4 bugs where the wrong type of data was returned for a given field was found by the property verifying that the result of executed queries did not include any ”Error” section in the response.

The 4 bugs not found where all the bugs were we filtered the entities requested with the wrong field, such as selecting for a project with *name* ”1” when *id* was the correct field to select for. Since there is no black-box information available for the tool to judge what a correct result is in this case, the tool can not find these bugs without a stronger state aware oracle.

All bugs resulting in a crash or an error message was found by our generated test cases and generic properties. Bugs that result in the wrong response returned would need stronger domain aware properties to be found.

As for the large-scale evaluation of fault-finding capability, we found several, to the best of our knowledge, new unreported input validation issues in GitLab by applying our method, both for Query and Mutation types. The issues were reported to the project's issue tracking system with queries generated by our implementation as reproducible examples [16]–[23]. Several of the issues have attracted developer attention and are scheduled for correction and inclusion in the product. These issues were caused by inserting our generated strings as input to the query arguments which required input values. For the Query type, when our string generators produced alpha-numeric strings no faults were found. However, when string generators producing any char value, 0-255, were used, input validation bugs were detected. While good at finding bugs, the string generators produced a lot of invalid input. About 40% of the executed queries returned with a client error of bad input. This indicates that the string generator produces input that finds bugs but at the same time produces a lot of queries that are not valid for further processing by the SUT.

One of the issues required a domain-specific modification to the generated values. The bug only manifested when a specific query field was selected for an existing GitLab project. An existing project is referenced with an argument in the format `"<user>/<project>"`. When our generator was configured to return an existing project, the query to find the bug was produced. This shows that knowledge of domain entities need to be supplied to get a stronger test suite, without existing project entities and a way to generate their paths, we would not have been able to find the bug.

These results indicate that our proposed method, of random generation of queries and input values, can find real faults in industry-developed applications.

Test cases in the form of randomly generated GraphQL queries can find real new faults in real industry software systems. There is a trade-off between the fault finding capability of different value generators and input deemed as valid by the system under test. To be more effective, custom generators can be used.

3) Threats to Validity: All findings presented are based on data gathered using our generators. The implementation of the GraphQL query generator is thus a threat to the internal validity of the results. While the exact numbers might vary with a different implementation, we have no reason to believe that another implementation following the proposed method would not reach the same conclusions as we have.

Our proposed method has a large component of randomness based on the usage of random generators. Given this, the exact results vary between executions. To mitigate such differences, we ran each configuration 30 times.

Considering the external validity, we applied our method on several publicly available APIs [24]. However, APIs not requiring authentication or registrations are typical demonstrations or non-production APIs. With that in mind, we focused our case-study, as reported, on an industry-grade API and a controlled example. The learnings from a set of public APIs were integrated into the solution used in our case-study. Those were (1) it is common for the schema to contain some domain-specific information that is expressed in a scalar type (such as a string) but requires a specific format found in the documentation. (2) For schema with a less recursive structure, a more straight forward generation approach can be used, but with a recursive schema, control of the recursion size is needed to control generation. These two points were then integrated into the solution used on our real-world industry-grade case-study, GitLab, by allowing for custom generators and to control the depth of the generation as described.

With that said, the main evaluation was performed on one industrial case study and one smaller example software. Performing an evaluation on a larger set of industry-grade software systems would increase the confidence in the generalizability of the proposed method. However, since GraphQL has a defined specification, the method is applicable to different APIs than those used in the case study.

V. Related Work

Since we are testing GraphQL APIs and using a PBT technique we present prior work in those areas and how it relates to ours.

A. GraphQL

The only current work in GraphQL API testing, to the best of our knowledge, is a proposed method by Vargas et al. where a user-provided initial test case is mutated by a set of ”deviations” to produce test cases [25]. This method differs from ours since we aim for a fully automatic method, where no user-provided test cases are required. Also, we use a PBT approach where Vargas et al. use a mutation-based approach.

The current state-of-the-practice in GraphQL testing is a manual practice, either by using tools that require manually created test-cases or recordings of manually created traffic, or manually coded tests. Since GraphQL uses the HTTP protocol, any testing tool with manually formulated test cases that target REST can be used, such as Postman.[13] However, a recent development is an automatic tool by Meeshkan,[14] but its impact in practice remains to be seen.

B. Property-based Testing

Property-based testing is a testing technique that have been applied to test systems in many different industry domains [26]–[29], including web services [11], [30]–[33], with different levels of automation. However, the targets for web-based APIs have been for RESTful services, WSDL based services, and JSON Schema, no method for GraphQL based APIs have been proposed.

The most recent example, QuickREST, is an automatic method that applies property-based testing to RESTful APIs [11]. The technique used is similar to ours, an automatic black-box approach leveraging property-based testing, but QuickREST targets RESTful APIs and does not provide solutions for the specific challenges posed by testing GraphQL APIs. An example of such a challenge is the cyclic nature of a graph-based API where the randomly generated recursion depth must be controlled, which we do in this work.

VI. Discussion and Future Work

We evaluated our method on GitLab. GitLab has parts of their complete API ported to GraphQL while the complete set of operations are still available in their REST API. This is an example of how one service API, GraphQL in this case, depends on resources created with another service, the REST API. Thus, for future work, it would be interesting to find an automatic method that can combine the test generation of both GraphQL and REST to be able to create, query, and verify resources of a system of combined services. Industry systems that are in the transition from REST APIs to GraphQL APIs could also benefit from such a method, since then feature parity test cases could be generated between the functionality exposed by the REST API and GraphQL API.

To increase the fault finding effectiveness an automatic test generation method would benefit from more domain knowledge. However, it is preferable not to break automation and require a human to codify all knowledge. Thus it would be interesting to automatically find domain knowledge from running a set of generated test cases and improve future generation with the knowledge gained from previous executions. In addition, interleaving both Query and Mutation operations would have the potential of increased fault finding.

The basic schema coverage criteria proposed could be expanded to include the coverage of sub-graphs in the query. Such an extension would allow a user to define important paths to make sure such paths are covered in the generated test cases.

With this paper, we would like to highlight the importance of performing research in automatic testing of GraphQL APIs since its usage is increasing in industry and encourage other researchers to propose white-box methods to solve this task, complementary to our black-box approach. In doing so, practitioners would have a stronger set of proven methods to apply to test their GraphQL APIs.

[13]https://www.postman.com/

[14]https://meeshkan.com/

VII. Conclusion

In only a few years GraphQL has become a widely adopted technique to expose Web APIs in industry. However, currently, the testing of such APIs is a manual task, requiring practitioners to manually create test cases. With this work, we want to alleviate this pain point and make automatic test case generation of GraphQL APIs possible and effective.

In this paper, we propose, to the best of our knowledge, the first method to fully automatically generate test cases for GraphQL APIs, as well as a method of how to evaluate the coverage of such test cases. We have proposed a property-based method where generators are used to randomly produce test cases in the form of GraphQL queries and values for any arguments in the query.

Our evaluation results show how this method achieves high coverage, up the 100% for many configurations, the sizing considerations to make when generating queries, and the fault finding capabilities of the method. In addition to being an effective method in reaching a high coverage of the schema, the method is as well effective in finding real faults, as shown with the reported issues to GitLab. Further, given a different set of input parameters and custom generators, the method is capable of producing queries that are valid depending on the complexity requirements of the system under test and to produce domain-specific input values.

Acknowledgements

This work is supported by ABB and the industrial postgraduate school Automation Region Research Academy (ARRAY) funded by The Knowledge Foundation.

References

[1] "GraphQL Foundation," 2020. [Online]. Available: https://foundation.graphql.org/

[2] "Who's using GraphQL?" 2020. [Online]. Available: https://graphql.org/users/

[3] "GraphQL and its benefits," 2020. [Online]. Available: https://help.shopify.com/en/api/getting-started/shopify-and-graphql/graphql-benefits

[4] "Why GitHub is using GraphQL," 2020. [Online]. Available: https://developer.github.com/v4/

[5] G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 140–150.

[6] G. Brito and M. T. Valente, "REST vs GraphQL: A Controlled Experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 81–91.

[7] "GitLab GraphQL API," 2020. [Online]. Available: https://docs.gitlab.com/ee/api/graphql/

[8] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, US, 2000.

[9] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 3:1–3:37, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3293455

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[11] 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)*, 2020, pp. 131–141.

[12] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 142–152.

[13] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1988042.1988046

[14] R. Hickey, "A History of Clojure," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386321

[15] "GitLab named a Leader in Cloud-Native CI," 2020. [Online]. Available: https://about.gitlab.com/analysts/forrester-cloudci19/

[16] "GitLab Reported Bug 1," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/issues/208121

[17] "GitLab Reported Bug 2," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/issues/208122

[18] "GitLab Reported Bug 3," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/issues/208125

[19] "GitLab Reported Bug 4," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/issues/208672

[20] "GitLab Reported Bug 5," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/issues/233927

[21] "GitLab Reported Bug 6," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/issues/233924

[22] "GitLab Reported Bug 7," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/issues/233922

[23] "GitLab Reported Bug 8," 2020. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/issues/233921

[24] "API Guru - Public GraphQL APIs," 2020. [Online]. Available: https://github.com/APIs-guru/graphql-apis

[25] D. M. Vargas, A. F. Blanco, A. C. Vidaurre, J. P. S. Alcocer, M. M. Torres, A. Bergel, and S. Ducasse, " Deviation Testing: A Test Case Generation Technique for GraphQL APIs ," in *11th International Workshop on Smalltalk Technologies (IWST)*, 2018, pp. 1–9.

[26] 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*, ser. ERLANG '06. New York, NY, USA: ACM, 2006, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/1159789.1159792

[27] J. Hughes, *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Cham: Springer International Publishing, 2016, pp. 169–186. [Online]. Available: https://doi.org/10.1007/978-3-319-30936-1_9

[28] 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)*, April 2016, pp. 135–145.

[29] 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)*, April 2015, pp. 1–4.

[30] M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro, "Turning Web Services Descriptions into Quickcheck Models for Automatic Testing," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 79–86. [Online]. Available: http://doi.acm.org/10.1145/2505305.2505306

[31] L. Lampropoulos and K. Sagonas, "Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services," in *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, WWV 2012, Stockholm, Sweden, 16th July 2012.*, 2012, pp. 3–16. [Online]. Available: https://doi.org/10.4204/EPTCS.98.3

[32] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco, "Automating Property-based Testing of Evolving Web Services," in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '14. New York, NY, USA: ACM, 2014, pp. 169–180. [Online]. Available: http://doi.acm.org.ep.bib.mdh.se/10.1145/2543728.2543741

[33] L. r. Fredlund, C. B. Earle, A. Herranz, and J. Mariño, "Property-Based Testing of JSON Based Web Services," in *Proceedings of the 2014 IEEE International Conference on Web Services*, ser. ICWS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 704–707. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2014.110