Deviation Testing: A Test Case Generation Technique for GraphQL APIs

Daniela Meneses Vargas
Universidad Mayor de San Simón, Bolivia

Alison Fernandez Blanco
DCC, Universidad de Chile

Andreina Cota Vidaurre
Semantics S.R.L., Cochabamba Bolivia

Juan Pablo Sandoval Alcocer
Universidad Católica Boliviana "San Pablo", Regional Cochabamba, Bolivia

Milton Mamani Torres
Object Profile SpA, Chile

Alexandre Bergel
DCC, Universidad de Chile

Stéphane Ducasse
RMOD, Inria-Lille Nord Europe, France

Abstract

GraphQL is a flexible and expressive query language. With the objective to replace the flawed and inefficient REST architectural style, GraphQL has been adopted by numerous online APIs and services. Despite its popularity, testing the implementation of a GraphQL schema is a crucial and still an open problem.

We found that classical techniques of test generation may be efficiently applied to GraphQL server. We propose a simple but expressive technique called *deviation testing* that automatically searches for anomalies in the way a schema is served. We demonstrate the feasibility of our approach using an implementation of GraphQL for Pharo and VisualWorks. Running our technique on the popular Yelp and Apollo GraphQL server uncovered several anomalies in the way the schema is served.

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

GraphQL is a query language for APIs and a runtime to resolve queries formulated in this language[1]. GraphQL provides a complete description of the data in an API in terms of types and fields. This description allows a GraphQL client to request the exact information that it needs in a *single* request. GraphQL uses these types to ensure that clients only ask for what is possible, providing clear and helpful errors.

Since the GraphQL public release, numerous software systems and programming languages have implemented GraphQL clients and server-side runtimes for resolving queries. These implementations, like any software program, may be subject to functional and performance issues. As far as we know, little effort has been done to improve the reliability of GraphQL servers.

In this paper, we propose *Deviation Testing*, a technique that measures the difference between a test case and its automatic generated variations, which we call *deviations*. Our technique takes an existing test case as input and automatically generates variations of this test case by using a set of deviation rules. As a result, deviation testing reports if the variations of the original test case meets or exceeds an acceptance criteria. The goal of *Deviation Testing* is to increase test coverage and help developers to find potential bugs in their GraphQL implementations and APIs.

[1] `https://graphql.org`

Our technique is inspired from mutation testing [13, 2] and amplification testing [1]. While mutation testing applies mutation operators to the system under test as a means to measure the strength of a test suite, deviation testing applies deviations (an equivalent of mutation operators) to generate new tests from existing ones as a means to measure the robustness of the API. We present a case study to show evidence of the applicability of our approach in different GraphQL APIs services.

The rest of this paper is structured as follows. Section 2 describes our implementation GraphQL for Pharo and VisualWorks. Section 3 details our approach, deviation testing. Section 4 lists the different deviation operators we employed to generate deviated tests. Section 5 gives an overview of our implementation. Section 6 illustrates our technique with two case studies. Section 7 briefly presents the work related to our effort. Section 9 concludes and outlines our future work.

2. SGraphQL in a Nutshell

This section gives a high-level description of *SGraphQL*, our implementation of GraphQL for Pharo[2] and VisualWorks, for illustration purpose.

2.1 Schemas and Types

The query language provides its own schema definition language. This schema allows developers to specify the fields and data types that are involved in queries. For instance, consider the following schema definition:

```
type Query{
  allProjects:[Project]
  project(name: String!): Project
}
type Project{
  versions: [Version]
  name: String!
}
type Version{
  number: Double
  author: String
  message: String
}
```

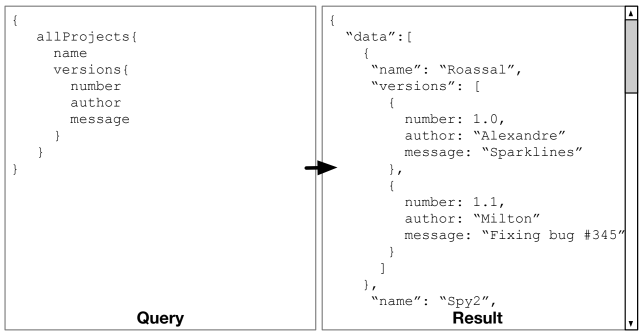
The schema defines three types `Query`, `Project`, and `Version`. Where the type `Version` has three fields: `number`, `author` and `message`, with types `Double`, `String` and `String` respectively. `Double` and `String` are scalar types already defined in SGraphQL.

The type `Project` has the fields `versions` and `name`. The `versions` field has the type `[Version]`. In this case, brackets (`[]`) are used to represent a list, therefore `[Version]` refers to a list of versions. The field `name` has a type `String!`, the symbol `!` means that this field cannot be null. The type `Query` contains all the fields that are used as entry-points. We further describe the usefulness of this type in the next section.

[2] `http://pharo.org/`

2.2 Basic Queries

Fields. Fields defined in a type (e.g., Query), represent the classical REST endpoints. The endpoint may be called directly from the client side. For instance, consider the following query:

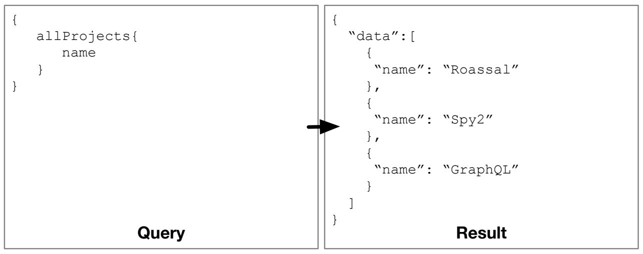


```
{
  allProjects{
    name
    versions{
      number
      author
      message
    }
  }
}
```

```
{
  "data":[
    {
     "name": "Roassal",
     "versions": [
       {
         number: 1.0,
         author: "Alexandre"
         message: "Sparklines"
       },
       {
         number: 1.1,
         author: "Milton"
         message: "Fixing bug #345"
       }
     ]
    },
     "name": "Spy2",
```

The query requests all the information of `allProjects` from the server. Note that, according to the schema defined in the previous section, it returns a list of project objects and the information of the respective fields.

Field Selection. A query specifies which objects fields are required. For instance, consider the following query:

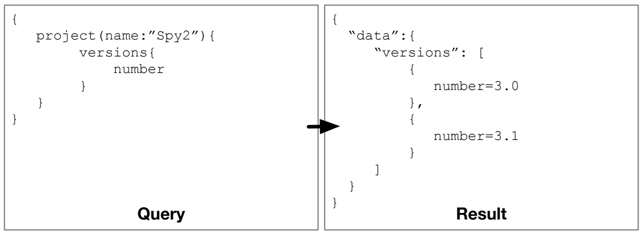


```
{
  allProjects{
     name
  }
}
```

```
{
  "data":[
    {
     "name": "Roassal"
    },
    {
     "name": "Spy2"
    },
    {
     "name": "GraphQL"
    }
  ]
}
```

Similar to previous example, it also requests `allProjects`. However in this case, the query specifies the fields that should be returned of each project object. In this particular example, it request only the `name` field.

Arguments. A query may also accept arguments. For example, consider the following query:



```
{
  project(name:"Spy2"){
      versions{
          number
      }
  }
}
```

```
{
  "data":{
     "versions": [
        {
          number=3.0
        },
        {
          number=3.1
        }
     ]
  }
}
```

The query uses the endpoint `project(name:String)` and sends the value `"Spy2"` as argument. As a result, the server returns only the information of the project `Spy2`. In addition, the query specifies that only the field `number` of the versions objects has to be returned. Note that the field `name` of the project `"Spy2"` is not in the result, since it was not specified.

2.3 Resolvers

In addition to the schema definition and queries, it is necessary to specify how the server will interpret and resolve a request. In our example, we need to specify how the server will handle the endpoints `allProjects` and `project( name:String)`. In this subsection, we describe how to implement the resolvers in Pharo smalltalk [3].

Schema Types. In SGraphQL, all the types must exist as a class in the image, which is a precondition to safely handle client requests. Therefore, in our example, we should have three classes: `Query`, `Project` and `Version`. For each field in the schema definition a method has to be defined in its corresponding class.

The Query Class. All the fields in the type `Query` are the entry points. Therefore, the class `Query` should have methods that correspond to that fields. In our example, we use the following method implementation for `allProjects` field:

```
Query>>allProjects
  "Return a list of the instances of the class Project"
  ^ ...
```

Field names are directly mapped to the method names. In the case of arguments, argument names are mapped to the selector parts of the method name. The following method implementation is for the field `project:(name:String)`:

```
Query>>projectName: aString
  ^ self allProjects detect: [ :project | project name = aString ]
```

3. Deviation Testing

Definition. We define *deviation testing* as a technique that measures the difference between a test case and its automatic generated variations (deviations). The goal of deviation testing is to increase the test coverage and help developers to find potential bugs in their GraphQL implementations.

The deviation test result reports if the variations of the original test case meets or exceeds an acceptance criteria.

Deviation Testing workflow. Figure 1 illustrates the deviation testing workflow. The deviation testing process is summarized in four steps.

- *Step 1: Input* – Deviation testing takes a test case as input. The test case is used as a seed for the automatic generation and used as the base for the acceptance criteria.
- *Step 2: Test Case Variations* – Generate small and controlled variations based on the input test case, we refer to these small variations as deviations. These variations are obtained by applying deviation rules on the original query. Deviation rules must consider two aspects: first, how the original test case will change, and second, how this change will affect the original test result.
- *Step 3: Test Case Execution* – Execute the initial test case and its variations.

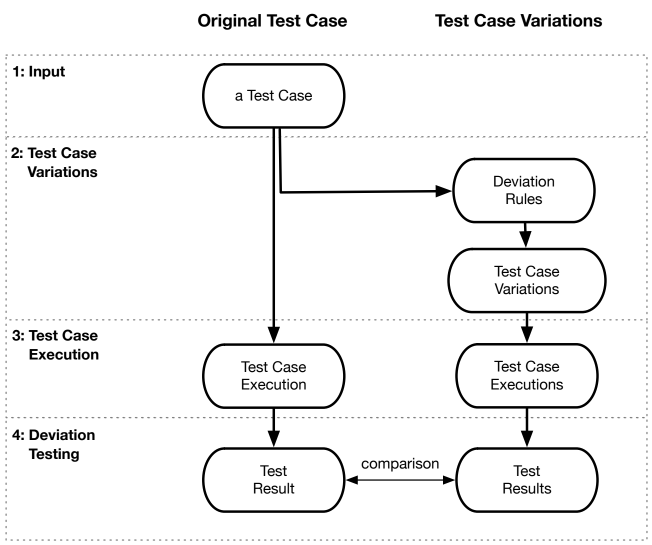


Figure 1: Deviation Testing Workflow

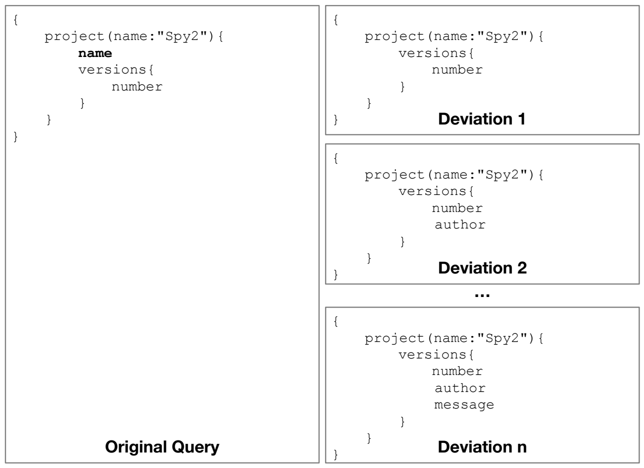
- *Step 4: Deviation Testing* – Measure and compare the difference of the results between the initial test case and its variations. The comparison should determine if the variations of the original test case met or exceeds an acceptance criteria, based on the deviation rules.

4. GraphQL & Deviation Testing

In this section, we propose the application of *Deviation Testing* to improve the test suite of GraphQL APIs.

4.1 Fields Deviation

Query Deviation. The Field deviation rule generates new queries by adding and deleting fields from the original query. By using the GraphQL schema, we are able to generate all the possible field deviations from the original query. Therefore, this deviation rule may generate a large number of query deviations. For instance, consider the following query deviation, where the field `name` is deleted from the original query.



```
{
    project(name:"Spy2"){
        name
        versions{
            number
        }
    }
}
```

Original Query

```
{
    project(name:"Spy2"){
        versions{
            number
        }
    }
}
```

Deviation 1

```
{
    project(name:"Spy2"){
        versions{
            number
            author
        }
    }
}
```

Deviation 2

...

```
{
    project(name:"Spy2"){
        versions{
            number
            author
            message
        }
    }
}
```

Deviation n

Expected Result. This deviation expects that the result of the deviated query is a subset of the result of the original query. In the previous example, the result of the deviated query should be the same than the original except that the project should not have the `name` in the result. In the case of a field addition, we have to check if the objects resulting from the query include the added fields.

Assumption. This rule assumes that the result of the original query is not an error and that the field `project(name:"Spy2 ")` returns an object. If the original query does not meet this assumption then this rule cannot be applied.

4.2 Not Null Deviation

Query Deviation. The Not Null Deviation rule replaces a declared not null argument with null. The GraphQL schema, has the information of which arguments cannot be null. Therefore, this deviation may generate an error by sending a null argument. For instance, consider the following deviation:

```
{
    project(name:"Spy2"){
        name
        versions{
            number
        }
    }
}
```

Original Query

```
{
    project(name:null){
        name
        versions{
            number
        }
    }
}
```

Deviation

Expected Result. It is expected that the result of the deviated query is an error. GraphQL defines a standard way to report this type of error, therefore we expect an error that meets the standard as answer.

Assumption. This rule assumes that the original query does not throw any error, particularly not an error related with null arguments. Otherwise, we could not have a strong conclusion about the deviation.

4.3 Type Deviation

Query Deviation. The Type Deviation rule replaces an argument with another one that does not match with the expected argument type. It is possible because the GraphQL schema contains the type of all the field arguments. For instance, consider the following deviation where a `String` argument has been replaced by an `Int` argument.

```
{
    project(name:"Spy2"){
        name
        versions{
            number
        }
    }
}
```

Original Query

```
{
    project(name:1){
        name
        versions{
            number
        }
    }
}
```

Deviation

Expected Result. The deviated query should return a type error. Similarly, to the previous rule, we expect an error like answer.

Assumption. The original query does not return an type error.

4.4 Empty Fields Deviation

Query Deviation. The Empty Fields Deviation rule deletes all the fields and subfields from the query. We need to consider that a field may be an object that also contains fields (subfields). This rule generates a number deviations, depending of the number of objects in the query. For instance, consider the following query has two possible deviations.

```
{
    project(name:"Spy2"){
        name
        versions{
            number
        }
    }
}
```

Original Query

```
{
    project(name:”Spy2”){
    }
}
```

Deviation 1

```
{
    project(name:”Spy2”){
        name
        versions{
        }
    }
}
```

Deviation 2

Expected Result. In GraphQL an object without fields may no be requested. Therefore, anyone of the deviations should result in a syntax error.

Assumption. This rule assumes that the original query does not throw a syntax error. Otherwise, we could not have any conclusion about the result of the deviated query.

5. Implementation

We created a prototype of deviation testing using the previous four rules described above. Our prototype presents different parts as shown in Figure 2. This section describes these parts.

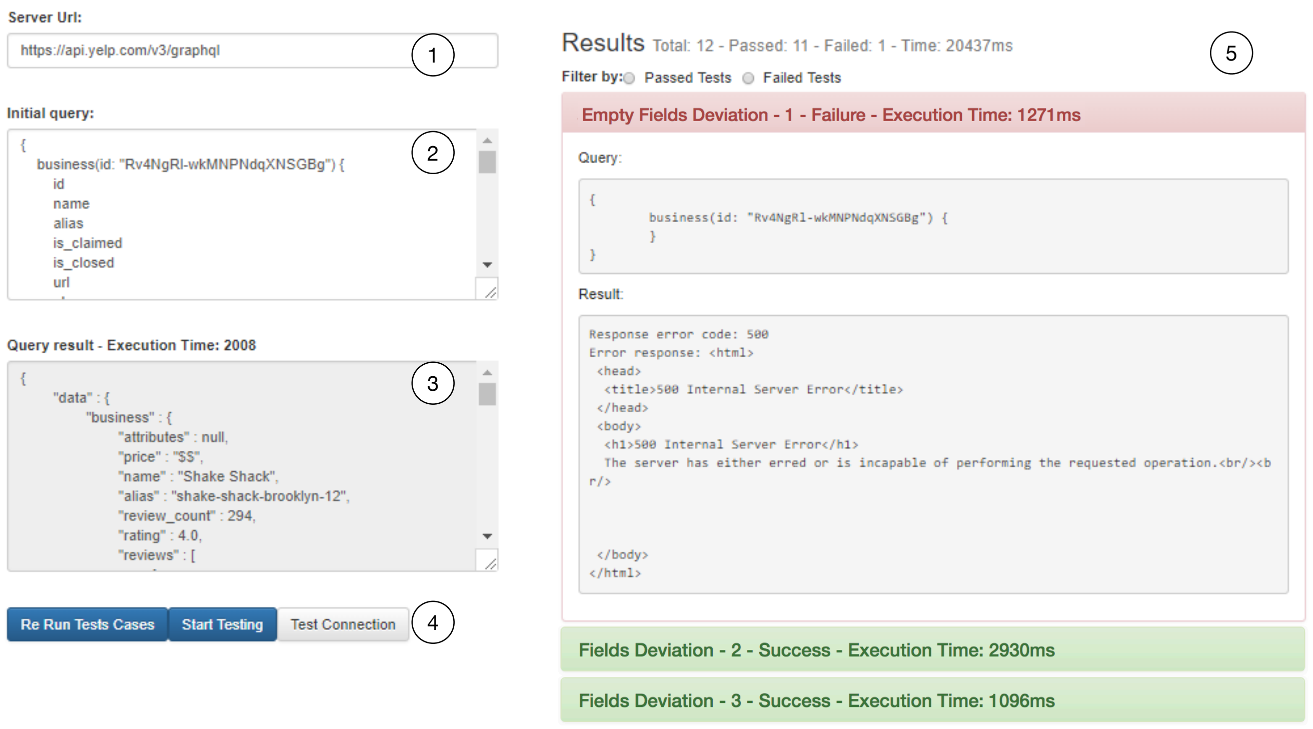
Server Url. We provide a little text area marked as 1 in Figure 2. This text area must contain the url of the selected server to be evaluated.

Initial Query. We give a text area marked as 2 in Figure 2. This text area must contain the initial query with the conditions described on the previous section.

Query Result. Marked as 3 in Figure 2 we have a text area with the response of the selected server to the initial query and the time of response on milliseconds.

Buttons. Below the Query Result we have a set of buttons marked as 4 in Figure 2. Each button has a particular associated action:

- *Re Run Test Cases.* This button is only useful when are test cases previously generated to run them again.
- *Start Testing.* This button generates tests using the deviation testing, given the selected server and the initial query.
- *Test Connection.* This button simply verifies that the server is still active and able to receive incoming requests.


Server Url:

```
https://api.yelp.com/v3/graphql
```

Initial query:

```
{
   business(id: "Rv4NgRl-wkMNPNdqXNSGBg") {
      id
      name
      alias
      is_claimed
      is_closed
      url
```

Query result - Execution Time: 2008

```
{
      "data" : {
            "business" : {
                  "attributes" : null,
                  "price" : "$$",
                  "name" : "Shake Shack",
                  "alias" : "shake-shack-brooklyn-12",
                  "review_count" : 294,
                  "rating" : 4.0,
                  "reviews" : [
```

Results Total: 12 - Passed: 11 - Failed: 1 - Time: 20437ms

Filter by: ○ Passed Tests ○ Failed Tests

Empty Fields Deviation - 1 - Failure - Execution Time: 1271ms

Query:

```
{
        business(id: "Rv4NgRl-wkMNPNdqXNSGBg") {
        }
}
```

Result:

```
Response error code: 500
Error response: <html>
 <head>
  <title>500 Internal Server Error</title>
 </head>
 <body>
  <h1>500 Internal Server Error</h1>
  The server has either erred or is incapable of performing the requested operation.<br/><b
r/>



 </body>
</html>
```

Fields Deviation - 2 - Success - Execution Time: 2930ms

Fields Deviation - 3 - Success - Execution Time: 1096ms

Figure 2: Prototype of deviation testing

Results. This section shows all the tests generated with deviation testing. At the beginning we have general information: the total number of test generated, the number of test successful, the number of test failed and the total response time of the server in milliseconds.

Also just below the general information we present a simple filter to select passed tests and failed tests. After the test generation, the list gives the test cases that are generated. By default each test is compressed and shown as a simple header. By clicking on the header, a generated test case is expanded with the following information:

- *Header.* The header presents some information of the test: the deviation rule used to generate the test case, the number of tests, the state of the test and response time in milliseconds of the server.

  Also the header is colored depending on the result of the test, green if is successful, otherwise red.
- *Query.* The new query sent to the server is shown on a text area.
- *Result.* The response of the selected server to the new query is shown on a text area.

Test Generation. First, our tool retrieves the schema defined for the GraphAPI specified in the GraphQL Type System. Then the test are generated using the initial query and the deviations rules. In the generation process, our tool attempts to apply the deviation rules from the initial query. For instances, the not null deviation rule will be applied in all the arguments that are defined as not null in the schema. Therefore, the number of tests generated by using this rules is related to the number of not null arguments in the schema. The process is similar of the remaining deviation rules.

To compress an expanded test case, just click on the header of the test.

6. Case Studies

This section presents three cases studies, each one describes the results of applying our deviation rules on a GraphQL APIs.

6.1 Case 1: Smalltalk GraphQL Demo API

The Smalltalk GraphQL Demo API was designed to essentially perform online demonstration, executes load and performance tests during the development phase. We have designed Demo API as part of SGraphQL.

Schema. The Smalltalk GraphQL Demo API has a relatively small schema, given by the Smalltalk GraphQL developers shown on the left side of Figure 3.

```
type Query {
    allClasses : [ Class ]
    allRectangles : [ Rectangle ]
    allFilms: [ Film ]
    film(name:String!):Film
},
type Film {
    name : String
    rating : Int
    director : Person
},
type Person {
    name : String
    age : Int
},
type Class {
    name : String
    methods : [ CompiledMethod ]
},
type CompiledMethod {
    selector : String
},
type Rectangle {
    origin : Point
    corner : Point
},
type Point {
    x : Float
    y : Float
}
```

Schema

```
{
    allClasses{
        name
        methods{
            selector
        }
    }
    allRectangles{
        origin{x y}
        corner{x y}
    }
    allFilms{
        name
        rating
        director{
            name
            age
        }
    }
    film(name:"terminator"){
        name
        rating
        director{
            name
            age
        }
    }
}
```

Initial Query

Figure 3: Schema and initial query for Smalltalk GraphQL

Initial Query. We use an initial seed query given on the right hand of Figure 3. This seed query exercises our schema. This initial query produces a JSON object and does not throw any error during its execution.

Results. Applying the deviations rules results in the generation of 48 test cases. Which 38 test cases successfully pass while 10 fail. A closer review reveals that the failing tests related to the deviation rules: *Type Deviation* and *Empty Fields Deviation*. Running deviation testing on the Demo API reveals part of our server that is still under construction. This punctual and small experiment illustrates the application of test-driven development in our development process.

6.2 Case 2: Yelp API

Yelp[3] is a well-known application to help user get recommendations about different business. Typical usages of Yelp includes requests about restaurants, medical assistance, or cafes.

Schema. Yelp has a public GraphQL, therefore we got the schema from the Yelp documentation shown in Figure 4.

```
type Query {
    business(id: String): Business
    business_match_best(name:
String!, address1: String, address2:
String, address3: String, city:
String!, state: String!,  country:
String!, latitude: Float, longitude:
Float, phone: String, postal_code:
String): Business
    search(term: String, location:
String, country: String, offset:
Int, limit: Int, sort_by: String,
locale: String, longitude: Float,
latitude: Float, categories: String,
open_now: Boolean, open_at: Int,
price: String, attributes: [String],
radius: Float): Businesses
    event(id: String): Event
    event_search(locale: String,
offset: Int, limit: Int, sort_by:
String, sort_on: String, start_date:
Int, end_date: Int, categories:
[String], is_free: Boolean,
location: String, latitude: Float,
longitude: Float, radius: Float):
Events
}
type Business {
    name: String
    id: String
    alias: String
    is_claimed: Boolean
    is_closed: Boolean
    url: String
    phone: String
    display_phone: String
    review_count: Int
    categories: [Category]
    rating: Float
    price: String
    location: Location
    coordinates: Coordinates
    photos: [String]
    hours: [Hours]
    reviews(limit: Intoffset: Int):
[Review]
    distance: Float
    attributes: GenericScalar
}
```

Schema

```
type Category {
    title: String
    alias: String
    parent_categories: [Category]
    country_whitelist: [Country]
    country_blacklist: [Country]
}
type Country {
    code: String
    locales: [Locale]
}
type Locale {
    code: String
    language: Language
    country: Country
}
type Language {
    code: String
    locales: [Locale]
}
type Location {
    address1: String
    address2: String
    address3: String
    city: String
    state: String
    zip_code: String
    country: String
    formatted_address: String
}
type Coordinates {
    latitude: Float
    longitude: Float
}
type Hours {
    hours_type: String
    open: [OpenHours]
    is_open_now: Boolean
}
type OpenHours {
    is_overnight: Boolean
    end: String
    start: String
    day: Int
}
type Review {
    id: String
    rating: Int
    user: User
```

Schema

```
    text: String
}
type Review {
    id: String
    rating: Int
    user: User
    text: String
    time_created: String
    url: String
}
type User {
    image_url: String
    name: String
}
type GenericScalar {}
type Businesses {
    business: [Business]
    total: Int
}
type Event {
    name: String
    id: String
    business: Business
    business_id: String
    attending_count: String
    category: String
    cost: Int
    cost_max: Int
    country: String
    description: String
    event_site_url: String
    image_url: String
    interested_count: String
    is_canceled: Boolean
    is_free: Boolean
    is_official: Boolean
    latitude: Float
    longitude: Float
    location: Location
    tickets_url: String
    time_end: String
    time_start: String
    zip_code: String
}
type Events {
    events: [Event]
    total: Int
}
```

Schema

Figure 4: Schema of Yelp

```
{
    business(id: "Rv4NgRl-wkMNPNdqXNSGBg"){
        id
        name
        categories {
          title
          country_whitelist {
            code
          }
          country_blacklist {
            code
          }
        }
        price
        location {
            address1
        }
        coordinates {
            latitude
            longitude
        }
        photos
        hours {
            open {
                end
                start
            }
        }
        reviews(limit: 10, offset: 0) {
            id
            rating
            user {
              name
            }
        }
    }
    business_match_best(name: "Shake
Shack", address1: "1 Old Fulton St",
address2: "", address3: "", city:
"Brooklyn", state: "NY", country: "US",
latitude: 40.7030151, longitude:
-73.9939538, phone: "+13474352676",
postal_code: "11201") {
        id
        name
        categories {
            title
            country_whitelist {
                code
            }
            country_blacklist {
                code
            }
        }
        location {
            country
        }
```

Initial Query

```
        coordinates {
            latitude
        }
        hours {
            open {
                end
                start
            }
        }
        reviews(limit: 10, offset: 0) {
            id
            rating
            user {
                name
            }
            url
        }
        attributes
    }
    search(term: "shake shack", location:
"new york", country: "US", offset: 0,
limit: 10, radius: 40000, sort_by:
"best_match", categories: "tradamerican",
locale: "", latitude: 40.7030151,
longitude: -73.9939538) {
        total
        business {
            name
            alias
            categories {
                title
                country_whitelist {
                    code
                }
                country_blacklist {
                    code
                }
            }
            location {
                address3
            }
            coordinates {
                latitude
            }
            photos
            hours {
                open {
                    day
                }
            }
            reviews(limit: 10, offset: 0) {
                id
                rating
                user {
                    image_url
                }
                text
            }
```

Initial Query

```
                distance
        }
    }
    event_search(location: "ny", offset: 0,
limit: 10, sort_by: "desc", sort_on:
"time_start", start_date: 151986240,
end_date: 1519862400, categories: "3",
is_free: true, latitude: 40.7030151,
longitude: -73.9939538, radius: 40000) {
        total
        events {
            id
            name
            business {
                id
                name
                location {
                    address1
                    city
                }
            }
            business_id
            category
            location {
                address1
                address2
            }
            zip_code
        }
    }
    event(id:"new-york-from-inmail-to-
interview-how-to-land-a-job-in-the-digital-
age") {
        id
        name
        business {
            id
            name
            phone
            location {
                address2
            }
        }
        business_id
        attending_count
        cost
        cost_max
        country
        location {
            address1
        }
        tickets_url
        time_end
        time_start
    }
}
```

Initial Query

Figure 5: Initial query for Yelp

Initial Query. We build a initial query based in *Yelp* website documentation and the schema. We make sure that the initial query does not throw any error and returns a set of JSON objects. The query is shown in Figure 5.

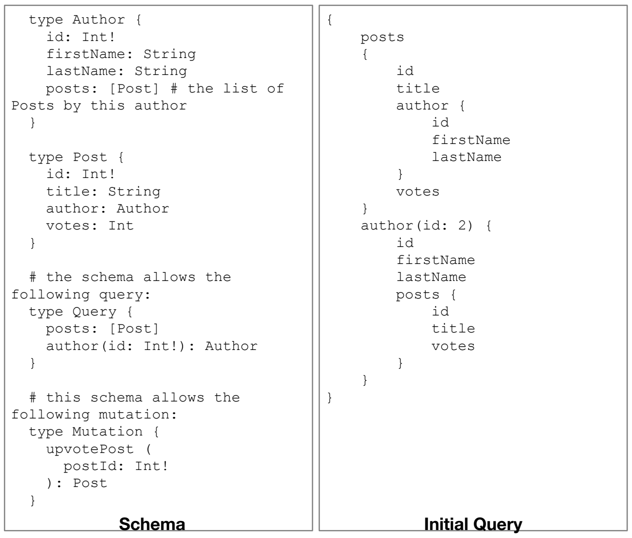
Results. As a result we generated 94 test cases, for which 5 of them fail and the remaining test cases pass successfully. After a closer look at the deviated queries and the obtained results, we conclude that Yelp API has an issue by handling the *Empty Fields Deviation*. The deviation results produces an HTML 500 error instead of a producing a JSON describing the syntax error, as the GraphQL standard requires.

[3] `https://www.yelp.com/developers/graphql/guides/intro`

6.3 Case 3: Apollo Demo API

Apollo[4] a set of technologies to help migrate from REST to GraphQL. Apollo includes tools, an engine, and an infrastructure to translate REST API's to GraphQL Schema. Apollo provide a demonstration website to test GraphQL APIs.

Schema. By using the introspection capability of the Apollo GraphQL server, we obtained the schema, as given on the left-hand side of Figure 6.



```
  type Author {
    id: Int!
    firstName: String
    lastName: String
    posts: [Post] # the list of
Posts by this author
  }

  type Post {
    id: Int!
    title: String
    author: Author
    votes: Int
  }

  # the schema allows the
following query:
  type Query {
    posts: [Post]
    author(id: Int!): Author
  }

  # this schema allows the
following mutation:
  type Mutation {
    upvotePost (
      postId: Int!
    ): Post
  }
```

```
{
    posts
    {
        id
        title
        author {
            id
            firstName
            lastName
        }
        votes
    }
    author(id: 2) {
        id
        firstName
        lastName
        posts {
            id
            title
            votes
        }
    }
}
```

Figure 6: Schema and initial query for Apollo Demo

Initial Query Based on this schema, we built the GraphQL query shown on the right side of Figure 6. This query does not throw any error and returns a set of JSON objects.

Results. Based on the initial query, our prototype generate 134 test cases, which all of them pass successfully. We therefore conclude that Apollo does not have any flaws regarding the generated tests cases.

7. Related work

This section summarizes testing tools for GraphQL, also tools and techniques focused on generate new tests to improve software development [4, 6].

7.1 GraphQL testing tools

There are different tools to test a GraphQL server. For example there are some frameworks used to test servers on Node.js which are applicable to a GraphQL server. Mocha[5] is one of the popular javascript test framework used to test GraphQL servers in Node.js. Mocha allows one to generate asynchronous testing, run tests serially, generate reports and map the exceptions to the test cases. Other frameworks are available, including SuperTest, Sails.js[6] and Chai[7].

To complement the activity to test a query, the schema itself can also be tested. Mocking[8], from Apollo, allows one to write tests with real queries focusing on the type definition of the schema. These tests are useful to avoid potential type conflicts using mocks.

One important task is to simulate queries and observe the response. It is one feature of using Test Client from Graphene[9]. It allows one to test that a query request is rendered by a Django template with certain values.

All the tools mentioned facilitate the creation of tests for GraphQL servers. However none generate automatically new test cases given an initial one.

7.2 Test amplification

There are many works focused on generate new test cases to improve the coverage or find faults on the implementation of software. At following we describe them:

Mutation testing. There are many existing works based on mutation testing to improve the coverage of the testing [5]. Mutation testing consists in executing two or more program mutations against the same test suite to evaluate the ability of the test suite to detect this alterations [13].

Baudry [2] presents the mutation testing focused to improve the quality of test cases. This paper shows case studies for automatic testing, using a bacteriological algorithm based on genetic algorithms. Their results show that the adaptations were good heuristics to their goals.

Later Tillman [12] presents parameterized unit testing to improve the coverage of the existing tests, turning the unit tests into parameterized unit tests. This work find inputs for parameterized unit tests using a way to analyze the behavior of a program for all the possible inputs called symbolic execution. Also Smith [11, 10] offers a set of mutation operators to produce new tests and increase the branch coverage. They also highlight the importance of selecting how to mutate the tests.

These related works study the program effects of mutate the tests to improve the coverage and presents an considerable effort to generate a mutated test with a known assertion. Contrary to our work we only focus on the schema defined on the server, an initial test case and generate new test cases based on the controlled deviations described.

New techniques or criteria. Harder *et al.* [8] use a technique that generates operational abstractions from test suites, adding cases until the operational abstraction stops changing. Once generated new test cases, they use the operational

[4] `https://www.apollographql.com`

[5] `https://mochajs.org/#getting-started`

[6] `https://sailsjs.com`

[7] `http://www.chaijs.com/api`

[8] `https://graphql.org`

[9] `http://docs.graphene-python.org/en/latest/testing/`

difference technique to select the test cases to improve fault detection.

Later Xu [14] on his work, analize the factors affecting test suite augmentation discovering that one of them is the technique used. Xu realize experiments using concolic as a technique to generate test cases and presents his algorithm for directed test suite augmentation. The results show that his algorithm was more effective and efficient than concolic technique on the code coverage.

Fraser [7] presents test generation and mutation to generalize pre and post-conditions, separating the test code from test input. This gives the chance to abstract a large piece of code input with symbolic parameters. Also they identify the relevant behavior using variation, while more errors a post-condition caches, more relevant is.

Later on, Pezze *et al.* [9] generate new integration test cases from existing unit test cases. To construct more complex test cases they use the information of unit test cases like: how to instantiate classes, how to construct arguments for method calls and the result expected. These test cases are focused on class interactions instead of single methods calls. This work present positive results finding faults on software.

The previous works present different techniques or new criteria to generate test cases improving the coverage or finding faults. Our work is focused on generate test cases for a typed query language. We can predict the expected results thanks to the rules of GraphQL.

8. Discussion & Future Work

We generate new test cases according the deviation rules described on section Section 4. The test generation ends when all the possible combinations are made. However, since we have a limited number of deviation rules, it is not possible cover all possible possible GraphQL queries that may reveal a flaw in a GraphQL server. As future work, we plan to expand the number of deviations rules to cover more cases and help developers to analyze how deviated are their GraphQL APIs.

The test generation also depends of the initial query. Therefore, a different initial query may generate different set of test cases. As future work, we plan to evaluate our approach using different real world initial queries and analyze the effect of the use of multiple initial queries in the test generation and test coverage.

Our prototype apply all the deviations rules over the original initial query, therefore the application order of the rules do not have an impact in generated test cases.

9. Conclusions

This paper presents a practical approach to test the implementation of a GraphQL server. In particular, we focus on determining how close an implementation is from the original GraphQL standard. Our technique, which we call *deviation testing*, is able to identify anomalies in the implementation by generating queries deviated from an original query. We use a set of deviation rules to generate deviated queries.

We have applied our technique on three case studies, namely SGraphQL, our home-made Smalltalk-based GraphQL server, Yelp, and Apollo. In two of them we found significant deviations from the GraphQL specification.

Acknowledgment

We gratefully thank LAM Research for its financial support. We also thanks to Truextend for the continuous support in Bolivia.

References

[1] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus. DSpot: Test Amplification for Automatic Assessment of Computational Diversity. Technical Report hal-01162219, HAL, 2015.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2):73–96, 2005.

[3] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[4] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry. The emerging field of test amplification: A survey. *arXiv preprint arXiv:1705.10692*, 2017.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[6] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[7] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374. ACM, 2011.

[8] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 60–71. IEEE, 2003.

[9] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 11–20. IEEE, 2013.

[10] B. H. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.

[11] B. H. Smith and L. Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009.

[12] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE software*, 23(4):38–47, 2006.

[13] M. Trakhtenbrot. New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 151–160. IEEE, 2007.

[14] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 406–413. IEEE, 2009.