Evolutionary-based Automated Testing for GraphQL APIs

Asma Belhadi
asma.belhadi@kristiania.no
Kristiania University College
Oslo, Norway

Man Zhang
man.zhang@kristiania.no
Kristiania University College
Oslo, Norway

Andrea Arcuri
andrea.arcuri@kristiania.no
Kristiania University College and
Oslo Metropolitan University
Oslo, Norway

ABSTRACT

The Graph Query Language (GraphQL) is a powerful language for APIs manipulation in web services. It has been recently introduced as an alternative solution for addressing the limitations of RESTful APIs. This paper introduces an automated solution for GraphQL APIs testing. We present a full framework for automated APIs testing, from the schema extraction to test case generation. Our approach is based on evolutionary search. Test cases are evolved to intelligently explore the solution space while maximizing code coverage criteria. The proposed framework is implemented and integrated in the open-source EvoMaster tool. Experiments on two open-source GraphQL APIs show statistically significant improvement of the evolutionary approach compared to the baseline random search.

KEYWORDS

GraphQL, EvoMaster, Evolutionary Algorithms, Automated Testing, Search-Based Software Testing, Fuzzing.

ACM Reference Format:
Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based Automated Testing for GraphQL APIs. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9--13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528952

1 INTRODUCTION

Web services are very common in industry, especially in enterprise applications using microservice architectures [21]. They are also becoming more common with the appearing of smart city technologies, where microservices are largely exploited in industrial internet-of-things settings [14, 15]. The investigation of automating techniques for generating test cases for web service APIs has become a research topic of importance for practitioners [6].

Due to the high number of possible configurations for the test cases, evolutionary techniques have been successfully used to address different software testing problems [3, 17]. Common examples are EvoSuite for unit test generation for Java programs [16], Sapienz for mobile testing [20] and EvoMaster for REST API testing [5, 8, 13].

The Graph Query Language (GraphQL) is a powerful language of web-based data access, created in 2012 and open sourced by Facebook in 2015 [2]. It addresses some of the RESTful API limitations, like the possibility of specifying what to fetch on a graph of interconnected data with a single query [18, 22]. Different companies have started to provide web APIs using GraphQL[1], like for example Facebook, GitHub, Atlassian, and Coursera. GraphQL is a query language and server-side runtime for application programming interfaces (APIs) [2]. Given a set of data represented with a graph of connected nodes, GraphQL enables to query such graph, specifying for each node which fields and connections to retrieve (and so recursively on each retrieved connected node).

To the best of our knowledge, only two recent approaches exist which explore the automated testing for GraphQL APIs, dealing with ``deviation testing'' [23] and ``property-based testing'' [19]. But none of them can analyze the source/byte-code of the application to generate better test cases.

At a high level, GraphQL APIs can be considered as Remote Procedure Call (RPC) services. Furthermore, there can be dependencies among operations (e.g., to test the retrieval of some data, a previous API call should have been made to create such data first). And so on.

In order to mitigate the combinatorial explosion in the test case generation, this paper presents a full framework based on evolutionary algorithms for automating GraphQL APIs testing, from the schema extraction to the generation of the test cases in executable test suite files (e.g., using JUnit).

The proposed framework has been implemented as an extension to the open-source EvoMaster tool, and exploits search-based code heuristics. It is freely available online, released as open-source. The main contributions of this research work are as follows:

(1) We develop a preprocessing strategy which first extracts the schema from the GraphQL API, and then defines chromosome representations for the test cases. We create different types of genes which enable the complete data representation of the GraphQL schema.
(2) We propose an evolutionary-based search which intelligently explores the test case space in GraphQL APIs. The genetic operators are used to explore the test case space while maximizing metrics such as code coverage.
(3) To validate the applicability of our presented framework, an empirical study has been carried out on two open-source GraphQL APIs. The results show a clear improvement of using the evolutionary algorithm compared with the random search baseline. Code coverage is improved up to +55.53%.

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

2 RELATED WORK

Automated testing of GraphQL APIs is a topic that has been practically neglected in the research literature. To the best of our knowledge, so far only two approaches have been investigated regarding the automated testing of GraphQL APIs [19, 23].

Vargas et al. [23] proposed a technique called ‘‘Deviation Testing’’. It consists of three steps. In the first step, an already existing test case is taken as input. This test constitutes a base to seed and compare the newly generated tests. The second step is the test case variation, where variations of the initial seeded test case are generated using deviation rules (where a deviation consists of a small modification). Four types of deviation rules are defined: 1) field deviation consists of adding and deleting the selection of fields in the original query; 2) not null deviation consists of replacing a declared non null argument with null; 3) type deviation consists of changing an argument type by another type; 4) empty fields deviation consists of deleting all fields and sub fields of the original query. The third step is the test case execution where the input test and its variations are executed. The last step consists of comparing the results between the input test and its variation (e.g., wrong inputs should lead to a response containing an error message).

Karlsson et al. [19] proposed a black-box property based testing method. The method consists of the following steps. First, all specifications of the types and their relations are extracted from the schema. Data is randomly generated according to the schema, with customized ‘‘data generators’’ provided by the user. In addition, the authors suggest two strategies to use as automated oracles: the first one aims to check the returned HTTP status codes, and the second one verifies that the resulting data returned conforms to the given schema.

In this research work, we present a new algorithm for automated testing of GraphQL APIs based on EvoMaster merits (the ability to do advance operations such as testability transformations [12] and SQL interaction analysis [11]). Our novel solution does not require any pre-existing test case (like in [23]), nor it requires the user to write customized input generators (like in [19]). In addition, we provide a complete testing pipeline, and an intelligent genetic-based exploration for the possible test case configurations.

3 GRAPHQL TEST CASE GENERATION

This section presents the proposed framework for automated test case generation of GraphQL APIs, built on top of the EvoMaster tool. The proposed framework targets white-box testing. The white-box testing is performed when the information related to the schema is provided and have access to the source code of the GraphQL API.

The process starts by extracting the schema from the GraphQL API. The chromosome template is then constructed from the schema. Test cases are represented by a sequence of HTTP requests, instantiated from the chromosome template. The test cases are evolved using MIO [7] enhanced with adaptive hypermutation [24], where each test case will contain genes representing how to build the GraphQL queries based on the given schema. The evolutionary search is performed by applying two mutation operators to evolve the test cases (one to change the queries/mutations in each HTTP call, and the other to add/remove HTTP calls in a test case). This enables to efficiently explore the solution space, with the aim of maximizing code coverage. From the final evolved solution, a self-contained test suite file (e.g., in JUnit format) is generated as an output of the search.

In the following, we describe the main components of the proposed framework in more details:

1. Problem Representation In order to fetch the whole schema from a GraphQL API, an *introspective* query is used. Given an entry point to the GraphQL API (e.g., typically a `/graphql` HTTP endpoint), GraphQL enables a standard way to fetch a schema description of the API itself. The schema specifies all the information about the available operation types, such as queries, mutations and all available data types on each of them. As a result, the GraphQL schema is returned in JSON format.

This latter is then parsed in our EvoMaster extension and used to create a set of action templates, one for each query and mutation operation. Each action will contain information on the fields related to input arguments (if any is present) and return values.

A chromosome template is defined for each action, which is composed of non-mutable information (e.g., field's names) and a set of mutable genes. In this context, each gene characterizes either an argument or a return value in the GraphQL query/mutation. For objects as return values, a query/mutation must specify which fields should be returned (at least one must be selected), and so on recursively if any of the selected fields are objects as well.

To represent the fact that a field is always optional for queries, a return gene is modeled by an object gene where all its fields are optional. However, we had to extend the mutation operator in EvoMaster with a post-processing phase, to guarantee that at least one field gene is selected during the search. In other words, if after a mutation of a gene, which represents a returned object value in the GraphQL query/mutation, all fields are de-selected, then the post-processing will force the selection of one of them (and so on recursively if the selected field is an object itself). On the other hand, if a return value is a primitive type, then there is no need to create any gene for it, as there is no selection to make.

To fully represent what is available from the GraphQL specification, the following kinds of gene types from EvoMaster are re-used and adapted:

(1) String: It contains string variables which are defined by an array of characters. A minimum length of the string is zero which represents the empty string. Each string gene cannot exceed a predefined maximum number of characters.
(2) Enum: This gene represents the enumeration type, where a set of possible values is defined, and only one value is activated at a given time. The elements in the set can be in different formats (e.g., enumerations of numbers or enumerations of strings).
(3) Float/Integer/Boolean: genes representing variables with simple data types. Boolean genes represent variables with true or false values. Integer and float genes represent integer and real-value variables, respectively.
(4) Array: This gene represents a sequence of genes with the same type. This gene has variable length, where elements can be added and removed throughout the search. In order to mitigate creating too large test cases, for instance with

millions of genes, the size of an array gene should not exceed a given threshold.

(5) Object: This gene defines an object with a specific set of internal fields. Differently from the array gene, where the elements should be with the same type, an object gene may contain elements with different types. To do so, this gene is represented by a map, where each key in the map is determined by the field name in each element in the object.
(6) Optional: a gene containing another gene, whose presence in the phenotype is controlled by a boolean value. This is needed for example to represent nullable types in arguments and selection of fields in returned objects.
(7) CycleObject: This special gene is used as a placeholder to avoid infinite cycles, when selecting object fields that are objects themselves, which could be references back to the starting queried object. Once a test case is sampled, its gene tree-structure is scanned, and all CycleObject genes are forced to be excluded from the phenotype.

After defining the possible type of genes supported by the proposed framework, we consider the solution space, where each solution is a set of test cases. A test case is composed of one or more HTTP requests. In order to represent an HTTP request, we typically need to deal with its components: HTTP verb, path and query parameters, body payloads (if any) and headers.

A GraphQL requests can be sent via HTTP GET (used only for queries) or HTTP POST methods with a JSON body (used for queries and mutations). For simplicity, we only use the verb POST for both queries and mutations. A GraphQL server uses a single URL endpoint (typically `/graphql`), where the HTTP requests with the GraphQL queries/mutations will be sent. In the context of test case generation for a GraphQL API, the main decisions to make are on how to create JSON body payloads to send. The genotype will contain genes (from the set defined above) to represent and evolve such JSON objects.

2. Search Operators and Fitness Function Once a chromosome representation is defined based on the GraphQL schema, test cases are evolved and evaluated in the same way as done for RESTful APIs in EvoMaster, including testability transformations [12] and SQL database handling [11]. Internally, the MIO algorithm is implemented in a generic way, independently of the addressed problem (e.g., REST and GraphQL APIs), and it is only a matter of defining an appropriate phenotype mapping function (e.g., how to create a valid HTTP request for a GraphQL API based on the evolved chromosome genotype).

When evaluating the fitness of an evolved test, we consider testing targets related to code coverage.

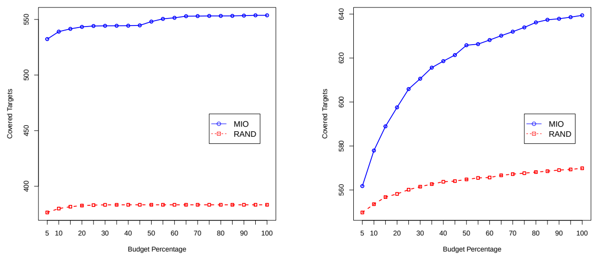
4 EMPIRICAL STUDY

4.1 Experimental Setup

In this section, several experiments have been carried out to answer the following research question:

RQ: How effective is MIO at maximizing code coverage compared to random search for GraphQL APIs?

We use two case of studies (i.e., *graphql-ncs* and *graphql-scs*) which are based on artificial RESTfull APIs from the existing EMB corpus [1]. For this study, we adapted them into GraphQL APIs, and added them to EMB [1]. *graphql-ncs* and *graphql-scs* are based on code that was designed for studying unit testing approaches on solving numerical [9] and string [4] problems.



(a) graphql-ncs **(b) graphql-scs**

Figure 1: Covered targets throughout the search

Table 1: Results for 100k HTTP call budget

SUT	Metrics	MIO	Random	$\hat{A}_{12}$	p-value	Relative
graphql-ncs	#Targets	**553.7**	383.3	**1.00**	**≤0.001**	**+44.44%**
	%Lines	**75.2%**	48.3%	**1.00**	**≤0.001**	**+55.53%**
graphql-scs	#Targets	**639.4**	569.9	**0.98**	**≤0.001**	**+12.21%**
	%Lines	**61.1%**	55.9%	**0.96**	**≤0.001**	**+9.36%**

The proposed framework is integrated in EvoMaster, where a comparison between MIO and the baseline random search algorithm is carried out. We set $100k$ HTTP calls as search budget in our proposed framework. To take into account the randomness of the algorithm, each experiment was repeated 30 times [10]. We selected covered testing targets (#Targets) and line coverage (%Lines), as metrics for the comparisons. The testing target (#Targets) is the default coverage criterion in EvoMaster. It comprises and aggregates different metrics, such as code coverage and status code coverage. Furthermore, we report (%Lines) which represents the line coverage.

4.2 Experiment Results

To compare MIO with Random, Table 1 reports their average #Targets, %Lines and an analysis of the pairwise comparisons using Mann-Whitney-Wilcoxon U-tests (*p-value*) and Vargha-Delaney effect sizes ($\hat{A}_{12}$), for each of the case studies.

Results show clearly better results for MIO compared to Random, with a high effect size (i.e., $\hat{A}_{12} \geq 0.98$) and a low p-value (i.e., $p \leq 0.001$), with the consistently best achievement in average #Target. Regarding the relative improvement for #Target, MIO achieves the most on *graphql-ncs* (i.e., +44.44%).

In Figure 1, we report plot-lines for demonstrating the performances of the two techniques for the two case of studies in detail, i.e., the number of covered targets throughout the search. MIO outperforms Random by a clear large margin throughout the search.

We also report average line coverage by MIO in Table 1. For both *graphql-ncs* and *graphql-scs*, a large improvement of MIO compared to Random search is observed. For instance, for *graphql-ncs*, MIO enables of covering 75.2% of lines, compared to the 48.3% of Random search.

Both *graphql-ncs* and *graphql-scs* contain many constrains based on numerical and string comparisons. Search-based techniques based on heuristics like the branch distance are highly effective on this kind of constraints. Getting significantly better results compared to random search is then not surprising.

***RQ**: In terms of line coverage, MIO demonstrates a consistent and significant improvements (up to 55.53%) compared with random search, on these numeric/string case studies. This shows the effectiveness of MIO for maximizing code coverage.*

5 THREATS TO VALIDITY

Threats to internal validity come from the fact that our experiments are derived from a software tool. Errors in such a tool could negatively affect the validity of our empirical results. Although our EvoMaster extension was carefully tested, we cannot provide any guarantee of it not having software faults. However, as it is open-source, anyone can review its source code.

Another potential issue is that the implemented solution in this research work is based on random generation. This happens in particular for population initialization of the evolutionary algorithm, where different test cases may be generated. To deal with this issue, each experiment was repeated 30 times [10], with different random seeds, and the appropriate statistical tests were used to analyze the results.

Threats to external validity are due to the fact that only two GraphQL APIs used in our empirical analysis. The generalization of such results to other APIs is not possible at this stage. More APIs should be investigated in the future. However, as this is the first work for white-box automated test case generation of GraphQL APIs, already achieving good coverage on two GraphQL APIs provide a promising first step.

6 CONCLUSION

This paper introduced a new approach for automated testing for GraphQL APIs. It is a full complete solution, starting from the schema extraction and ending by automatically generating test cases outputted in JUnit format. In order to intelligently explore the test case space, evolutionary computation techniques are used. In addition, two mutation operators (internal and structure mutation) are defined, where the goal is to maximize code coverage.

To validate the applicability of the proposed framework, it is integrated in the EvoMaster open-source tool. Our empirical analysis was carried out on two GraphQL APIs. The results show the clear improvement of using the evolutionary computation compared with the random search baseline. Our extension to support GraphQL APIs is now integrated in EvoMaster. EvoMaster is released as open-source, available at *www.evomaster.org*.

As future perspective, we plan to explore more case studies in order to generalize the effectiveness of the proposed framework. Of particular importance it will be to apply our techniques in industrial settings, to see and evaluate how practitioners would use tools like EvoMaster on their APIs.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 864972).

REFERENCES

[1] [n. d.]. EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB.
[2] [n. d.]. GraphQL Foundation. https://graphql.org/foundation/.
[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742--762.
[4] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175--203. https://doi.org/10.1002/stvr.v16:3
[5] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
[6] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959--1981.
[7] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195--206.
[8] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
[9] A. Arcuri and L. Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 265--275.
[10] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.
[11] Andrea Arcuri and Juan P. Galeotti. 2020. Handling SQL Databases in Automated System Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1--31.
[12] Andrea Arcuri and Juan P Galeotti. 2020. Testability transformations for existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 153--163.
[13] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.
[14] Edwin Cabrera, Paola Cárdenas, Priscila Cedillo, and Paola Pesántez-Cabrera. 2020. Towards a Methodology for creating Internet of Things (IoT) Applications based on Microservices. In *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 472--474.
[15] Flavio Cirillo, David Gómez, Luis Diez, Ignacio Elicegui Maestro, Thomas Barrie Juel Gilbert, and Reza Akhavan. 2020. Smart city IoT services creation through large-scale collaboration. *IEEE Internet of Things Journal* 7, 6 (2020), 5267--5275.
[16] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*. 416--419.
[17] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
[18] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*. 1155--1164.
[19] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. Automatic Property-based Testing of GraphQL APIs. *arXiv preprint arXiv:2012.07380* (2020).
[20] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 94--105.
[21] Sam Newman. 2015. *Building Microservices*. " O'Reilly Media, Inc.".
[22] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. 2018. GraphQL-LD: linked data querying with GraphQL. In *ISWC2018, the 17th International Semantic Web Conference*. 1--4.
[23] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane Ducasse. 2018. Deviation testing: A test case generation technique for GraphQL APIs. In *11th International Workshop on Smalltalk Technologies (IWST)*. 1--9.
[24] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).