Verification of Microservices Using Metamorphic Testing

Gang Luo[1], Xi Zheng[1](✉), Huai Liu[3], Rongbin Xu[4], Dinesh Nagumothu[2], Ranjith Janapareddi[2], Er Zhuang[1,5], and Xiao Liu[2]

[1] Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
james.zheng@mq.edu.au
[2] School of Information Technology, Deakin University, Geelong, VIC 3220, Australia
[3] Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, Australia
[4] School of Computer Science and Technology, Anhui University, Hefei, China
[5] School of Information and Electronic Engineering, Zhejiang University of Science and Technology, Hangzhou, China

Abstract. Microservices architecture is drawing more and more attention recently. By dividing the monolithic application into different services, microservices-based applications are more flexible, scalable and portable than traditional applications. However, the unique characteristics of Microservices architecture have also brought significant challenges for software verification. One major challenge is the oracle problem: in the testing of microservices, it is often very difficult to verify the test result given a test input, due to the features of wide distribution, heterogeneity, frequent changes, and numerous runtime behaviors. To tackle such a challenge, in this paper, we investigate how to apply metamorphic testing into the verification of microservices-based applications, which is a simple yet effective approach to oracle problem. Empirical studies are conducted to evaluate the performance of metamorphic testing based on real-world microservice applications, against the baseline random testing technique with a complete oracle. The results show that in the absence of oracles, metamorphic testing can deliver relatively high failure-detection effectiveness. Our work demonstrates that metamorphic testing is both applicable and effective in addressing the oracle problem for the verification of microservices, similar to many other application domains.

Keywords: Automatic test case generation · Microservice · Metamorphic Testing · Mutation testing · Software verification

1 Introduction

In recent years, Microservices architecture emerged as a new paradigm in software architecture patterns. Many large companies like Amazon [23], Netflix [28], Gilt [18], LinkedIn [22] and SoundCloud [4] have adopted Microservices to build their services. By dividing large and complex applications into a set of services, microservices achieve significant improvement in scalability, deployment, and portability than the traditional monolithic applications.

While Microservices architecture is gaining more and more popular today, its unique characteristics have brought some challenges in software verification. The heterogeneity of microservices requires testing tools to be specific to each service's programming language and runtime environment. Failure-recovery logic rather than business logic becomes the main focus due to the rapidly evolving code [20]. No unified model of system validation can support continuous integration of microservices [36]. Among all these issues, in this paper, we focus on the following oracle problem in the testing of microservices.

Moving from monolithic architecture to Microservices architecture usually requires to decompose the applications, then to distribute and process the data into hundreds of microservices [18]. Keeping each test specification up-to-date with such a large number of service interfaces is very difficult or even practically infeasible [5]. The lack of latest and comprehensive test specifications makes it almost impossible to precisely verify the test result given any possible test input, that is, the oracle problem exists. The interactions among services are more complicated compared to the traditional Service-Oriented Architectures (SOAs). The goal of traditional SOAs is to integrate different software assets to realize business processes, while microservices are aiming at improving the characteristics of software. Thus, it is a challenge to track the test dependencies within each microservice [19]. With the size and complexity growing in deployments, the behavior of each underlying microservice and how microservices are configured to interact also change. All these challenges make finding oracles (e.g., invariants) even more untenable for microservices-based applications [31]. In other words, the oracle problem becomes even more serious in Microservices architecture, as compared with SOA, not to mention other traditional application domains. To ensure the correctness of such application, an effective way of testing microservices without oracle is urgently needed.

Aiming to address the challenges in the verification of microservices-based applications, this paper mainly investigates into the state-of-the-art testing tools and models. Throughout the investigation, we find that all the existing microservice testing tools and models assume (explicitly or implicitly) the presence of oracle. If the oracle problem exists, the effectiveness of these techniques is significantly hindered. Moreover, these techniques may even become useless if the test results cannot be verified.

In the context of software testing, metamorphic testing [8,37] has emerged as a mainstream approach to the oracle problem. It simply utilizes some necessary properties of the software under test, termed as metamorphic relations, among inputs and outputs of multiple executions of the software under test for verifying the rest results. It has been justified that by simply using a small number of diverse metamorphic relations, metamorphic testing is able to detect a similar number of bugs as a complete orale [27]. Metamorphic testing has been successfully applied to detect various faults in a variety of application domains, including bioinformatics [9], compilers [25] and web services [38,39]. It is thus natural to consider the usage of metamorphic testing in addressing the predominant oracle problem for the verification of microservices-based applications. In this paper, we investigate the applicability and effectiveness of metamorphic testing via a series of experiments on real-world microservices-based applications.

There are three contributions as follows.

(1) For the first time, metamorphic testing is presented to detect and identify the failures in microservices, which has been evaluated as an effective approach in various application domains.
(2) A series of metamorphic relations are built to validate the correctness of each application, which has been verified as an effective variable from software specification.
(3) Compared with random test results. Prove that using metamorphosis tests in microservices is an effective method.

The remainder of this paper is organized as follows: Sect. 2 provides an overview of Microservices architecture and compares it with the traditional monolithic architectures to identify the difficulties in software verification brought by the characteristics of Microservices architecture. Section 3 presents some state-of-the-art microservice testing tools and models. Section 4 explains the metamorphic testing approach and justifies why we choose it for the verification of microservices. Section 5 evaluates metamorphic testing against a few real-world microservices-based applications and presents various convincing results. Section 6 concludes the paper and points the future work.

2 Microservice Preliminary

Microservice is aimed to decompose large, complex applications into a set of services to achieve a better performance in the development and deployment of various services [26]. Each service implements some distinct features and functionalities of the application. However, microservice works as an individual mini-application, which means each service can have its architecture, enabling the technical possibilities to be implemented more flexibly. Some microservices would expose an API consumed by other microservices or by the client of applications. Other microservices might implement a web user interface (UI). When a request is received, the relevant microservices work together as a whole to give back a response.

Microservices have many benefits. By dividing complex monolithic applications into a set of more manageable services, it is much faster to develop and much easier to understand individual services [35]. Also, each service can be developed independently by a team focusing on that service. As long as the functionality of this service is correctly implemented, any technology could be used. One service may be implemented by JAVA, and another service may be programmed by C++. The communication between services will be achieved by a predetermined protocol. At present, the entities provided by each service are identified by a universal Uniform Resource Identifier (URI) [3], and Hypertext

Transfer Protocol [15] is used as the communication protocol. In [16], the representational state transfer (REST) has been presented as the general conceptual framework for API designing, which becomes a popular practice for microservice APIs. Moreover, each microservice can be deployed and scaled independently, enabling the application to achieve better performance with heterogeneous resources. For example, for a frequently used service, more resources will be allocated to it than a less popular service. In [41], a case study is conducted where an enterprise application is developed and deployed in the cloud with a monolithic approach and a microservice approach. These results show that the microservices approach has a lower infrastructure cost but a higher response time.

However, microservice has its drawbacks. The Microservices architecture will require more effort from developers to build applications due to the complexity of the distributed system [26]. Besides, how to partition the existing system into microservices requires much more consideration [29]. Multiple services call for stronger coordination in the development teams. And the complicated interservice communication brings big challenges for software verification. Therefore, microservice also has oracle problems because of the system structure of microservice, such oracle problems in microservice programs are more serious than those in traditional single-chip microcomputer structure.

3 Related Work

Though the microservice is a rising concept, it shares many similarities with traditional web services. Many testing tools and models used in web services are also applicable to microservice testing. In [2], an automatic test case generation method is presented with the Web Service Description Language (WSDL) in webservices. In [30], an improved test case generation method is proposed for web services with the pairwise testing technique. In [6], a method is proposed to test RESTful services using a formal specification of the web services to generate the test cases. In [33], a model-based approach is presented to test RESTful webservices with the UML protocol state machines. In [14], a model-driven testing approach of RESTful APIs is presented. In [24], property-based testing with the RESTful web service is proposed to achieve promising results. In [45,46], a native language based specification is provided to verify the monitored applications at runtime with bounded resource usage. However, none of them can address the verification challenge (i.e., testing without oracle) we are raising in this paper. Next, we will investigate some most promising testing tools and models for microservices.

The Directed Automated Random Testing (DART) is a testing tool proposed in [17]. With DART, unit testing can be performed completely and automatically on any program without any test driver or harness code. The static source-code analysis is applied to automatically extract the interface of a program with its external environment. A test driver for this interface can be automatically generated to perform random testing to simulate the most general environment that the program can operate in. Moreover, how the program behaves under random testing is analyzed dynamically to automatically generate new test inputs to systematically direct the execution along alternative program paths.

The symbolic execution plays an important role in the DART. Starting with random input, a DART-instrumented program calculates an input vector during each execution for the next execution. This vector contains values that are the solutions of symbolic constraints gathered from predicates in-branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths [17]. DART makes automatic test case generation possible, which solves the first challenge. However, DART can only be applied at the unit testing level. Also, the computational expense of running tests with DART is huge due to the symbolic execution.

In [1], EvoMaster is a prototype microservice testing tool that is proposed aiming to generate test case for RESTful API automatically in the white-box test. In this tool, the APIs are directly retrieved from the JSON [10] definition generated by Swagger, which is a popular REST documentation generation tool available for more than 25 different programming languages. Then Genetic Algorithm (GA) is applied to generate the test cases for a given test problem. The Whole Test Suite approach is adapted with the extra usage of a test archive to evolve the test suites. A set of randomly initialized test cases with variable size and length are treated as GA individuals. A test suite's fitness will be the total fitness of all its test cases. When new offspring is generated, the crossover operator will mix test cases from two-parents sets.

In EvoMaster, each test case will be modified by the mutation operator, such as increasing or decreasing a numeric variable by 1. This supports all valid types in JSON. When a test is executed, the tool will check all the targets it covers. If a new target is covered, then the test will be duplicated from the test suite and put into an archive to save the target during the search process. At the end of the search, all the tests in the archive with the removed tests are checked to find out the minimized suite to be written as a test class file. Regarding the verification aspect, HTTP status codes are used to check the behavior of the system. For example, an HTTP invocation with the returned status of 403 (unauthorized) can imply that the authorization check is wrongly relaxed. EvoMaster can automatically generate test cases for higher-level testing. However, services are assumed in isolation, and it is a white-box testing which requires access to the source code of the services. Besides, code coverage results are relatively low compared to the existing, manually written results. The tool is restricted to JAVA, while many other programming languages such as C++ are used to build the microservices-based application.

In [42], a REST API testing model is proposed with an expressive description language based on JSON to solve the challenges like multidimensional API validation requirements, call sequencing, organization of test cases and data dependency between test cases. The test model description language has a good expressing capability and makes it possible to automatically generate test cases through an interface. Following the thought of traditional programming languages, the whole test plan is considered as a program and the test case is considered as a function, which are the basic units of the whole test plan to be executed.

The test suite is also considered as a function, which is the call of a function group. A function has some parameters and a return value. Parameters indicate what function depends on from the external environment; the return value indicates what influences the function will make on the external environment. Similarly, if one test case depends on the data generated by a previously executed test case, the data are input as parameters of the current test case. If the subsequent test case depends on the data generated by the current test case, the data generated by the current test case is, as an input, transmitted to the subsequent test case. The idea of this testing model is straightforward and effective. Implementing this model will solve the challenge of how to organize the test sequence of test cases and the test data dependency between test cases.

After the investigation of the existing testing tools and models for microservices, we can observe that much attention has been paid to the automatic test case generation under the assumption of the presence of oracle. However, none of the existing approaches has addressed the oracle problem, which is a common problem in microservices testing and verification. In this study, we aim to apply a novel approach, namely metamorphic testing, to address the oracle problem.

4 Metamorphic Testing

Most previous studies in test case generation assumed the availability of a test oracle, which can verify whether the result is correct or not for any given test cases [21]. However, in many practical situations, it is very difficult, or even impossible, to verify the test result, given a test input. Such a problem, termed as the oracle problem, is a fundamental problem in software testing.

More specifically, in the scenarios of microservices, as explained before, a test oracle either does not exist or is impractical to be used. This oracle problem thus becomes a fundamental challenge for microservice verification, because it significantly restricts the applicability and effectiveness of most test strategies [27]. Similar to many other domains, the applicability and effectiveness of many existing test case generation strategies are greatly hindered due to the oracle problem. In particular, the outcomes of many microservices can only be vaguely described without precise fixed values.

Metamorphic Testing (MT) [7] has been empirically justified as a cost-effective solution to address the oracle problem in many areas. The core part of MT is a set of Metamorphic Relations (MRs), which are identified based on the necessary properties of the system under test. Each MR actually represents the verifiable relationship among multiple inputs and their corresponding expected outputs. Once an MR is identified, it can be used to transform some existing test cases (called source test cases) to generate test cases (called follow-up test cases). Different from the traditional way of verifying test results against the existing oracles, MT verifies the source and follow-up test cases against the corresponding MR. For instance, consider the mathematical function $\min(a, b)$ that calculates the minimum value of two integers a and b. The order of the inputs should not influence the output, which can be expressed as the following metamorphic relation: $\min(a, b) = \min(b, a)$. In this MR, (a, b) is called the source test case, and (b, a) is called the follow-up test case. Let P_{min} be a program implementing the minimum function. P_{min} can be tested against the MR by running some metamorphic tests where specific input values are used. For instance, we can first run $P_{min}(2, 3)$ and then run $P_{min}(3, 2)$ and then check whether these two outputs are equal or not. If the outputs of a source test case and its followup test case(s) violate the MR, we can conclude that the program under test contains a bug.

In this paper we use MT to alleviate the oracle problem in microservice verification under a similar intuition: even if we cannot determine the correctness of each test case created for microservices, it might still be possible to use MRs to identify the failures in Microservices [8].

Following the general procedure of MT [27], the basic steps for implementing MT in microservices can be summarized as follows:

– Identify necessary properties of the microservices under test and use MRs to represent these properties.
– Generate the source and follow-up test cases based on each MR, and execute these test cases.
– The test outputs are evaluated against the corresponding MRs to check whether MRs are satisfied or violated. Any violation of MRs indicates a program failure in the observed microservices.

In the following section, we will evaluate the effectiveness of MT in the verification of microservices.

5 Evaluation

For evaluating the effectiveness of the metamorphic testing, some real-world microservices-based applications are selected and tested against random testing with oracle for comparison. The research questions that need to be addressed are as following:

– Can metamorphic testing alleviate the oracle problem in microservice verification?
– How effective is metamorphic testing when compared with random testing with oracle?

5.1 Objects

As one of the advocates to develop and update the applications rapidly using microservices, Eventuate is one platform that develops the sample microservices which make the business logic easy to be implemented [13]. Numerous microservices that are developed by Eventuate and the selected applications are three applications across three different areas. As shown in Table 1, one microservice application is Transactional Service [40] in the Event-Sourcing + CQRS example application [12], which involves mathematical operations for business transactions. Another microservice application is Restaurant Management Application [34], which comprises search and return functions. The third application is the Customers and Orders Management Application [11] that is made of both mathematical and search functions.

Table 1. Basic statistics of the sample microservice

Selected application in eventuate	Application area	Function
Business transactions	Event-Sourcing + CQRS	Mathematical operations for business transactions
Restaurant management application	API gateways, orchestrated APIs	Search and return functions
Customers and orders management application	Message buses/brokers, direct calls	Mathematical and search functions

5.2 Implanting Mutants to Source Microservice Application

We use mutation to plant errors for these three microservices-based applications. For testing purposes, we create four mutant versions of each application and the number of mutants implanted in each mutant version is shown in Table 2. We implant mutants for the above three microservice programs from three aspects, which are internal implementation of microservice (Version 1), microservice communication (Byzantine failure and Omission failure, Version 2), microservice data (Version 3) (thus each microservices-based application has three different mutated versions), and a version combination with all these errors combined (Version 4). For internal mutations in microservices, we use the automated mutation generation tool LittleDawina [32]. Call mutations between microservices

Table 2. The number of mutants implanted in each mutant version

Application object	Version 1	Version 2	Version 3	Version 4
Business transactions	5	6	5	12
Restaurant management application	4	5	3	10
Customers and orders management application	3	5	3	6

are achieved by changing the relationship between publication and subscription of events (publisher and subscriber modes are the communication mechanism among microservices [43,44]). Data mutations are achieved by changing the order, by which the communication data are stored in each microservice.

- MR1.1 - The sum balance of debtors and creditors before the transfer should be equal to the values after the transfer. $\sum$ M[i] (before) = $\sum$ M[i] (after), where M[i] is the account balance in customer i's account and i = 1, 2, 3, ..., X, where X is the total number of the debtors and creditors.
- MR1.2 - The sum balance of creditors before the transfer divided by the sum balance of the creditors after the transfer will always be less than one. Mb[i] (before)/Mb[i] (after) < 1, where Mb[i] is the account balance in creditor i's account and i = 1, 2, 3, ..., Y, where Y is the total number of the creditors.
- MR1.3 - The sum balance of debtors before the transfer divided by the sum balance of the debtors after the transfer will always be greater than one. Md[i] (before)/Md[i] (after) > 1, where Md[i] is the account balance in debtor i's account and i = 1, 2, 3, ..., Z, where Z is the total number of the debtors.

For the Restaurant Management Application, the restaurants are created along with the corresponding details like Zip Code, Opening Time, Closing Time and the Day of the week. The microservice GET function can fetch the results of the restaurants when any of those details are entered as inputs. The metamorphic relations we created for this microservice are as follows:

- MR2.1 - The restaurant details we get when only "zip code" is entered must be a superset of the details we get when both "Zip Code" and "day of the week" are entered.
- MR2.2 - The restaurant details we get when only "zip code" is entered must be a superset of the details we get when both "Zip Code" and "hour(time)" are entered.
- MR2.3 - The restaurant details we get when only "day of the week" is entered must be a superset of the details we get when both "day of the week" and "hour(time)" are entered.

For the customers and orders application, the developed metamorphic relation is that the status of one order is not equal to the status of another order where the credit limit and order amount are inversed.

- MR3.1 - If f(x,c) defines the order creation service, where x is the order amount and c is the credit limit, then the status of f(x,c) $\neq$ f(c,x).

5.3 Validating the Relations

After identifying a Metamorphic relation (MR), we validate the correctness of MR by reading the software specification, analyzing the source program of the relevant part of the MRs, and performing unit testing on this part of the program. To further validate our MR from the software specification, we also conduct source code analysis and unit testing of the microservices. We have assigned different people for MR creation and verification to avoid the validity issues.

5.4 Variables

Independent Variables. As we are going to measure the effectiveness of the metamorphic testing, we have considered the effectiveness as one independent variable. There is a comparison for measuring the effectiveness, and random testing with oracle has been chosen as a baseline technique. If both these variables show similar results or if the metamorphic testing outperforms the random testing, we can deduce the metamorphic testing as a reliable technique.

Dependent Variables. The metric of evaluation is the dependent variable in this case, where we consider a number of failures identified. For a given number of test cases in all mutated versions of three microservices we have tested, we compare the number of detected failures by random and metamorphic testing. This metric is used in measuring the effectiveness of metamorphic testing.

5.5 Generation of Test Cases

Random Test Case Generation. Random testing has been performed on the transactional microservice by creating random users and adding several self-accounts and third-party accounts for each user. The transactions are carried out by transferring some random amount of money that is generated by a random function $randi(imax)$, which can return an equally distributed random number from 1 to $imax$. For the restaurant management application, the restaurants are created with some random attributes. Then, queries are created using "zipCode", "hour" and "dayOfweek" by randomizing each value. The customers and orders application comprises creating customers and placing orders. The customers have the attributes like credit limit and orders have bill amount. Both of them are generated using a random function. The orders are queried using the order id randomly for the verification of response. By using the random function, we generate random tests for each version of each object. Here, the number of random test cases created for each object is different and it is related to the function of the application. Random test cases need to be written by considering functional coverage. So there will be fewer test cases with fewer features.

Metamorphic Test Case Generation. For executing the metamorphic tests, we need to use the metamorphic relation to generate follow-up test cases based on the original test cases (source test cases). Along with execution of both source and follow-up test cases, the test results are compared with the metamorphic relation for finding the fault. If the relation is not satisfied, then it can be considered that the program contains bugs and needs to be modified. Iterative testing consists of repeated application of the relation to generate various follow-up test cases based on source test cases [13]. For the transactional service application, we randomly generated source test cases and use the metamorphic relation to generate follow-up test cases to verify the application. We have generated random numbers as the balances for debtors and creditors accounts and other random numbers which are the transfer amounts. For the restaurant management service application, we have created some restaurants randomly using the random input values for various attributes in each restaurant. For generating the source test cases, we have created random Zip Code for a list of restaurants. To generate the follow-up test cases for the first metamorphic relation, we have selected Zip Code and Day of the week to verify the list of those restaurants which are open. We have created the source and follow-up test cases randomly for the other relations in a similar fashion, by selecting the Zip Code, Hour, and Day of week respectively to verify the list of those restaurants which are open. For the order application, we also use random values for credit limit and order amount. The source test cases are generated using random testing and the follow-up test cases are generated by reversing the credit limit and order amount.

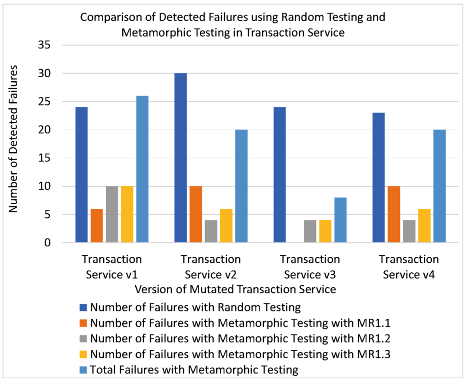
5.6 Evaluation Environment

To run these microservices for our evaluation, we have chosen to use the Ubuntu Linux operating system and have installed Docker-Compose in it. We use the personal computer with Intel $Core^{TM}$ i7-7500U CPU @ 2.70 GHz 2.90 GHz along with 12 GB DDR4 2133 MHz SODIMM RAM and Intel HD Graphics 620, NVIDIA GeForce 920MX. The microservices are compiled using the Gradle provided in the applications and these compiled applications are uploaded to the Docker, which runs in the Oracle Virtual Box. The applications that are uploaded to the Docker can be accessed from any browser, and we have used the built-in Firefox for the evaluation in this case.

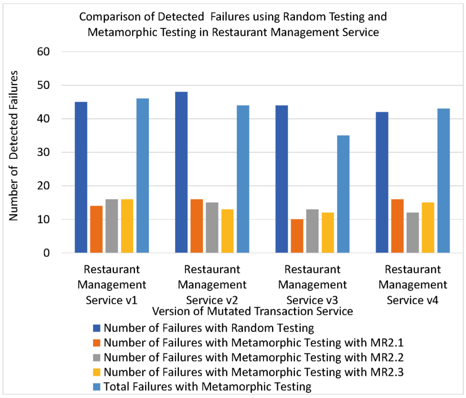
5.7 Results

As illustrated in Fig. 1a, we have run 30 test cases for all the mutated versions (four versions) of the transaction service application, and it is found that 24 failed test cases have been identified by using the random testing for the first mutated version with a fully developed oracle. On the other hand, the metamorphic testing has achieved 26 total failed test cases out of 30 without a comprehensive oracle. The situations for the other 3 mutated versions get 30, 24 and 23 failures respectively while the metamorphic testing gets 20, 8 and 20 failed cases without a completely developed oracle. It can demonstrate that the metamorphic testing outperforms the random testing for the first version where as it performs on par for the 2^{nd} version and 4^{th} version. This justifies the effectiveness and validity of metamorphic testing.

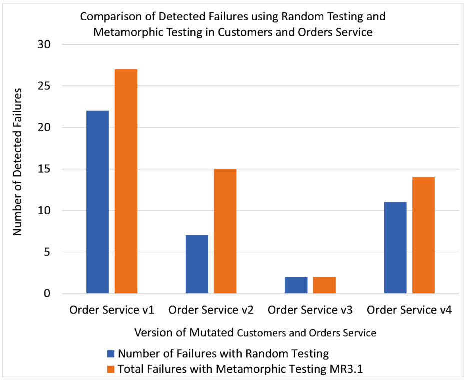
As illustrated in Fig. 1b, we have run 48 test cases for all the mutated versions (four versions) of the restaurant management service application, and it is found that 45 failed test cases have been identified by using the random testing for the first mutated version with a fully developed oracle. On the other hand, the metamorphic testing has achieved 46 total failed test cases out of 48 without a comprehensive oracle. The situations for the other 3 mutated versions get 48, 44 and 42 failures respectively while the metamorphic testing gets 44, 35 and 43



(a) Transaction



(b) Restaurant Management



(c) Customer and Order Service

Fig. 1. Comparison of failure detection using RT and MT

failed cases without a completely developed oracle, which implies a comparable performance to random testing with oracle.

As illustrated in Fig. 1c, there is only one identified relation for the Customers and Orders service, where there has been 22 failures for the first version of order service by random testing and 27 failures obtained by metamorphic testing. There have been 7, 2 and 11 failures for the other 3 mutated versions, and 15, 2 and 14 failures for the metamorphic testing with a partial oracle. These results indicate that metamorphic testing is achieving almost similar results to that of random testing in many mutated versions of the microservice based applications and sometimes achieving better results without proper oracle where the random testing is tested with a perfect oracle.

6 Conclusion and Future Work

In this paper, a major verification challenge of testing microservices without oracle has been identified and some state-of-the-art microservice testing tools and models have been studied to highlight the contributions we made by adopting metamorphic testing approach. Then, after walking through the Microservices architecture and metamorphic testing methodology, we presented our empirical study and results by applying metamorphic testing on three real-world microservice based applications. The empirical results have been clearly shown that metamorphic testing is able to alleviate the oracle problem in microservice verification and has a relatively high effectiveness in identifying problem failures compared with a state-of-the-art testing strategy with perfect oracle. Our next step is to evaluate the metamorphic testing against much larger real-world microservice based applications to identify further research direction in applying metamorphic testing to aid the verification of microservices-based applications.

References

1. Arcuri, A.: Restful API automated test case generation. In: IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 9–20. IEEE (2017)
2. Bai, X., Dong, W., Tsai, W.T., Chen, Y.: WSDL-based automatic test case generation for web services testing. In: IEEE International Workshop on Service-Oriented System Engineering (SOSE 2005), pp. 207–212. IEEE (2005)
3. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifier (URI): generic syntax. Technical report (2004)
4. Calçado, P.: Building products at soundcloud—Part I: dealing with the monolith (2014). https://tinyurl.com/jxy8yl9. Accessed 23 June 2019
5. de Camargo, A., Salvadori, I., Mello, R.D.S., Siqueira, F.: An architecture to automate performance tests on microservices. In: Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services, pp. 422–429. ACM (2016)
6. Chakrabarti, S.K., Rodriquez, R.: Connectedness testing of restful web-services. In: Proceedings of the India Software Engineering Conference, pp. 143–152. ACM (2010)

7. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Technical report, HKUST-CS98-01, Department of Computer Science, Hong Kong (1998)
8. Chen, T.Y., et al.: Metamorphic testing: a review of challenges and opportunities. ACM Comput. Surv. (CSUR) **51**(1), 4 (2018)
9. Chen, T., Ho, J.W., Liu, H., Xie, X.: An innovative approach for testing bioinformatics programs using metamorphic testing. BMC Bioinform. **10**(1), 24 (2009)
10. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). Technical report (2006)
11. Customers and orders management application. https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders. Accessed 23 June 2019
12. Event-sourcing + CQRS example application. https://tinyurl.com/y3fqzz8y. Accessed 23 June 2019
13. Eventuate.io microservice provider. https://eventuate.io/. Accessed 23 June 2019
14. Fertig, T., Braun, P.: Model-driven testing of restful APIs. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1497–1502. ACM (2015)
15. Fielding, R., et al.: Hypertext Transfer Protocol - HTTP/1.1. Technical report (1999)
16. Fielding, R.T., Taylor, R.N.: Architectural styles and the design of network-based software architectures, vol. 7. Irvine Doctoral dissertation, University of California (2000)
17. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. ACM SIGPLAN Not. **40**, 213–223 (2005)
18. Goldberg, Y.: Scaling gilt: from monolithic ruby application to distributed Scala micro-services architecture (2014)
19. Heinrich, R., et al.: Performance engineering for microservices: research challenges and directions (2017)
20. Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., Sekar, V.: Gremlin: systematic resilience testing of microservices. In: International Conference on Distributed Computing Systems (ICDCS), pp. 57–66. IEEE (2016)
21. Hierons, R.M.: Oracles for distributed testing. IEEE Trans. Softw. Eng. **38**(3), 629–641 (2011)
22. Ihde, S., Parikh, K.: From a monolith to microservices + REST: the evolution of LinkedIn's service architecture (2015)
23. Kramer, S.: The biggest thing Amazon got right: the platform. Gigaom, 12 October 2011
24. Lamela Seijas, P., Li, H., Thompson, S.: Towards property-based testing of RESTful web services (2013)
25. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. ACM SIGPLAN Not. **49**, 216–226 (2014)
26. Lewis, J., Fowler, M.: Microservices (2014). martinfowler.com
27. Liu, H., Kuo, F., Towey, D., Chen, T.Y.: How effectively does metamorphic testing alleviate the oracle problem? IEEE Trans. Softw. Eng. **40**(1), 4–22 (2013)
28. Mauro, T.: Adopting microservices at netflix: lessons for architectural design (2015). Acesso em 8 2016
29. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Int. J. Open Inf. Technol. **2**(9), 24–27 (2014)

30. Noikajana, S., Suwannasart, T.: An improved test case generation method for web service testing from WSDL-S and OCL with pair-wise testing technique. In: International Computer Software and Applications Conference, vol. 1, pp. 115–123. IEEE (2009)
31. Panda, A., Sagiv, M., Shenker, S.: Verification in the age of microservices. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, pp. 30–36. ACM (2017)
32. Parsai, A., Murgia, A., Demeyer, S.: LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex Java systems. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 148–163. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_10
33. Pinheiro, P.V.P., Endo, A.T., Simao, A.: Model-based testing of restful web services using UML protocol state machines. In: Brazilian Workshop on Systematic and Automated Software Testing, pp. 1–10 (2013)
34. Restaurant management application. https://github.com/eventuate-examples/eventuate-examples-restaurant-management. Accessed 23 June 2019
35. Richardson, C.: Microservices: decomposing applications for deployability and scalability. InfoQ **25**, 15–16 (2014)
36. Savchenko, D., Radchenko, G.: Microservices validation: methodology and implementation. In: 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists, pp. 21–28 (2015)
37. Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A.: A survey on metamorphic testing. IEEE Trans. Softw. Eng. **42**(9), 805–824 (2016)
38. Segura, S., Parejo, J.A., Troya, J., Ruiz-Cortés, A.: Metamorphic testing of restful web APIs. IEEE Trans. Softw. Eng. **44**(11), 1083–1099 (2017)
39. Sun, C.A., Wang, G., Mu, B., Liu, H., Wang, Z., Chen, T.Y.: Metamorphic testing for web services: framework and a case study. In: 2011 IEEE International Conference on Web Services, pp. 283–290. IEEE (2011)
40. Transactional service application. https://github.com/cer/event-sourcing-examples. Accessed 23 June 2019
41. Villamizar, M., et al.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC), pp. 583–590. IEEE (2015)
42. Hu, W., Yu, H., Liu, X., Chen, X.: Study on REST API test model supporting web service integration. In: 2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), pp. 133–138. IEEE (2017)
43. Yu, D., Jin, Y., Zhang, Y., Zheng, X.: A survey on security issues in services communication of microservices-enabled fog applications. Concurr. Comput. Pract. Exp. **31**, e4436 (2018)
44. Zheng, T., et al.: SmartVM: a SLA-aware microservice deployment framework. World Wide Web **22**(1), 275–293 (2019). https://doi.org/10.1007/s11280-018-0562-5
45. Zheng, X., Julien, C., Podorozhny, R., Cassez, F.: BraceAssertion: runtime verification of cyber-physical systems. In: 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems, pp. 298–306. IEEE (2015)
46. Zheng, X., Julien, C., Podorozhny, R., Cassez, F., Rakotoarivelo, T.: Efficient and scalable runtime monitoring for cyber-physical system. IEEE Syst. J. **12**(2), 1667–1678 (2016)