DOI 10.31891/2307-5732-2022-305-1-107-108
УДК 004.42:004.05

YAKIV YUSYN
https://orcid.org/0000-0001-6971-3808
e-mail: yusin.yakiv@gmail.com
TETIANA ZABOLOTNIA
https://orcid.org/0000-0001-8570-7571
e-mail: tetiana.zabolotnia@gmail.com
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

METAMORPHIC TESTING-AS-A-SERVICE: A NEW DESIGN PATTERN OF CLOUD SERVERLESS SYSTEMS FOR METAMORPHIC TESTING

The task of quality assurance of software systems in IT is still an urgent problem, and due to the growing complexity of these systems is becoming increasingly difficult to use old methods of automated testing. One of the new methods of automated testing is metamorphic testing, which can be applied to systems of any complexity and which can be performed efficiently in the cloud. However, software for performing metamorphic tests in the cloud is still in the early stages of its development, due to the still low popularity of the method in the industry. So, the purpose of this work is to improve the software to perform metamorphic tests in the cloud by developing the corresponding design pattern to improve the expected results against the software code quality metrics. To achieve this objective, the new MTaaS design pattern has been developed that is based on the idea of metamorphic relation decomposition into individual parts together with automatic code generation of the relations' and functions' bodies. The combination of these two ideas allows the developer to concentrate only on the implementation of the logic of metamorphic relations, hiding from him all other details (such as creating serverless functions). To evaluate the developed design pattern, two software systems for metamorphic testing of the same software artifact were developed: one software system was implemented without the use of the MTaaS pattern, the other with its use. The following four code quality metrics were used in this evaluation: maintainability index, cyclomatic complexity, class coupling, lines of code. The analysis of evaluation results has demonstrated improvement of the class coupling and maintainability index metrics without worsening other metrics. Thus, the evaluation showed the effectiveness of using the developed design pattern during developing software systems for metamorphic testing based on serverless computing.

Keywords: metamorphic testing; cloud computing; serverless computing; design pattern.

ЮСИН Я. О., ЗАБОЛОТНЯ Т. М.
Національний технічний університет України «Київський політехнічний інститут ім. Ігоря Сікорського»

METAMORPHIC TESTING-AS-A-SERVICE: НОВИЙ ШАБЛОН ПРОЄКТУВАННЯ ХМАРНИХ БЕЗСЕРВЕРНИХ СИСТЕМ МЕТАМОРФІЧНОГО ТЕСТУВАННЯ

Задача забезпечення якості програмних систем в IT досі залишається актуальною, при чому через зростаючу складність цих систем все складніше стає застосування старих методів автоматизованого тестування. Одним із нових методів автоматизованого тестування є метаморфічне тестування, яке може бути застосованим до систем будь-якої складності та яке може ефективно виконуватись у хмарі. Проте, програмне забезпечення для виконання метаморфічних тестів у хмарі досі знаходиться у початковій фазі свого розвитку, що викликано поки що невеликою популярністю методу в індустрії. Відповідно, метою даної роботи є удосконалення програмного забезпечення для виконання метаморфічних тестів у хмарі шляхом розроблення відповідного шаблону проєктування, що покращить отримувані результати за метриками якості програмного коду. Для досягнення поставленої мети розроблено новий шаблон проєктування MTaaS, в основу якого покладено ідеї декомпозиції метаморфічного зв'язку на окремі складові разом з автоматичною кодогенерацією тіл зв'язків та функцій. Поєднання цих двох ідей дозволяє розробнику зосередитись тільки на написані логіки метаморфічних зв'язків, приховуючи від нього всі інші деталі (такі як створення безсерверних функцій). Для оцінювання розробленого шаблону проєктування розроблено дві програмні системи метаморфічного тестування одного і того самого програмного артефакту: одна програмна система реалізована без використання шаблону MTaaS, інша з його використанням. При оцінювані використано наступні чотири метрики якості програмного коду: індекс підтримуваності, цикломатична складність, зв'язність класів, кількість рядків коду. Аналіз отриманих результатів показав покращення метрик індекса підтримуваності та зв'язності класів, при відсутності погіршення інших двох. Таким чином, проведене оцінювання показало ефективність використання розробленого шаблону проєктування при розроблені програмних систем метаморфічного тестування на основі безсерверних обчислень.

Ключові слова: метаморфічне тестування; хмарні обчислення; безсерверні обчислення; шаблон проєктування.

Introduction

Quality assurance of software systems is still an urgent problem in the IT area because software errors may lead to financial and/or reputational losses, create security problems, and in the worst case – lead to human fatality.

According to the World Quality Report 2021 [1], the software quality assurance expenditures amount at 18 through 35 percent of the total IT budget of an average company, and 36% of that share (i.e. 6-12% of the total budget) is spent on development and/or purchase of the testing software tools. At the same time the share of automated software methods use instead of manual human testing increases every year, and, correspondingly, in the category of software tools the emphasis shifts to development of automated tests.

A huge number of automated testing methods have been developed as of today, but simple oracle-based tests remain the most popular ones [2]. Although this method is based on the obvious idea of comparing obtained output data against the expected values for the defined input data, in practice determination of this pair "expected

input data – expected output data" can be complicated. First of all, complexity of the oracle determination depends on complexity of the software artifact to be tested – the come complex software artifact, the more complex determination of the expected output data. Apparently, that for the multiplication function it is much simpler than for the text clustering software.

Accordingly, a part of newer automated testing methods is called to solve the problem of determining expected output data (so called "oracle problem" [3]) in the most obvious way – through complete avoidance of specific input and output data operation. Metamorphic testing is one of such methods based on the idea of *metamorphic relations* – relevant relations between inputs and outputs specific for the given domain area [4]. Metamorphic relation is the relationship describing how the output data should change when certain input data have been changed. For instance, the following relation can serve as metamorphic relation for the multiplication function: if one multiplicand is increased 2 times (change of input data), the result would also increase 2 times (change of output data). Accordingly, metamorphic relations make the basis of metamorphic testing that checks whether the specified relations are fulfilled for the given software artifact or not.

Today the cases of successful application of the metamorphic testing method for validation of web applications [5–7], compilers [8,9], computer graphics [10, 11] and bioinformatics [12, 13] software tools are described in the literature, but this method remains not widespread in the industry. It means that development of the public software tools for development of metamorphic tests, architecture of such software tools, and relevant design patterns is still an urgent problem. One of the advanced areas of activities in this field is the use of cloud technologies for running metamorphic tests as the last ones may run efficiently in parallel owing to their independent nature. Sharing a set of metamorphic tests for parallel execution in the cloud in practice results in significant reduction of the total execution time because in a majority of cases the tests are complex and such that are performed at the level of end-to-end testing.

Thus, **the purpose** of this work is to improve the software to perform metamorphic tests in the cloud by developing the corresponding design pattern to improve the expected results against the software code quality metrics.

Related Works

The idea of metamorphic testing was proposed for the first time in [4], and since that time it was successfully applied in various areas.

The first meta-analysis of the studies on metamorphic testing was provided in [14] and extended in [15]. The second meta-analysis besides the description of the current state of the metamorphic testing area also describes the challenges and pending issues, with the execution of metamorphic tests using cloud calculations among them.

Execution of metamorphic testing using cloud calculations was described for the first time in [16]. In this study the developed software was oriented at the use of virtual machines EC2 from the cloud provider AWS with their manual creation, control, and deletion.

The study [17] proposed to apply the serverless computing technology (Function-as-a-Service pattern) for designing, implementing, and executing of metamorphic testing. The study demonstrated the expediency of using serverless computing and proposed generic architecture for implementing of metamorphic testing with their use.

Overview of Metamorphic Testing Serverless Architecture

The study [17] describes the generic serverless software architecture for running metamorphic testing as component, deployment, and sequence diagrams.

In general, such architecture envisages identification of four individual components:

- Software artifact being tested.
- Input data generator.
- Models of input data, output data, and data to be generated.
- Metamorphic relations.

When transferred to serverless deployment, the fifth component appears – serverless functions (hereafter – metamorphic functions).

However, besides the description of the interaction process between the dedicated components (using the model component) the generic architecture does not provide any recommendations regarding the implementation of components at the class level. Especially it is true for the metamorphic relations component that contains all metamorphic testing logic (that is actually the logic of the serverless application functioning because besides testing logic it contains only metamorphic functions and data generators). Thus, implementation of this component at the class level can be done in various ways and all of them would comply with the generic serverless architecture. Apparently, some of them will be better than others from the standpoint of quality, simplicity and software code maintainability.

Also, in the case of a large number of metamorphic tests another problem occurs with serverless architecture: every metamorphic relation requires existence of a metamorphic function, and these functions are similar to each other. In fact, with such approach metamorphic functions differ only in their names, callable metamorphic relation, and, possibly, data generator. All the other items – function registering as serverless (using the cloud provider library), request parsing, and results return – do not change between functions. That results in duplication of the metamorphic function code as well as possible copy&paste errors. During development of serverless metamorphic tests, it would be desirable to have an opportunity to concentrate on development of the metamorphic relations' logic only, leaving alone the metamorphic functions similarly to how the architecture pattern

Function-as-a-Service allows concentrating on business logic only, leaving alone the whole code for its launch and deployment.

So, the following problems of serverless metamorphic testing can be identified:

- Necessity of decomposing the metamorphic relation so that to avoid extremely coupled or duplicated code.
- Duplication of the metamorphic function code with insignificant differences.

To solve the above problems the new architecture pattern is proposed in the study that would allow concentrating exceptionally on development of the main components of metamorphic tests.

Metamorphic Testing-as-a-Service

For the summary description of the architecture pattern MTaaS developed within these studies let us use the generally accepted pattern description (used for the first time in [18]): “intent” – “motivation” – “applicability” – “structure” – “participants” – “collaborations” – “consequences” – “implementation”.

Intent. Implement metamorphic testing based on serverless computing architecture in the most efficient way according to the code quality metrics by focusing exceptionally on the implementation of metamorphic relations only.

Motivation. Let us consider the ways of solving problems that occur during the implementation of metamorphic testing (including based on serverless architecture) first separately, and then we will demonstrate how their resolution can be combined.

During the metamorphic relation implementation in the software code it is possible to identify the following main components:

- Input data change component that receives a certain type TInput at the input and returns it in a modified form. Hereafter we will call such component an input metamorphosis or input data metamorphosis.
- Corresponding output data change component that in its signature is similar to the input metamorphosis but differs in the TOutput type. We will call such component an output metamorphosis or output data metamorphosis.
- Launch code of the software artifact being tested and being responsible for TInput transformation into TOutput.
- Code for comparison of two TOutput copies.

Actually, the software artifact launch code is permanent for different metamorphic relations. If the software artifact requires different settings for its launch (e.g., configuration file or configuration object providing), such settings must be considered as a part of input data. This approach also allows the creation of metamorphic relations with the involvement of the settings metamorphosis (e.g., during testing of data serializer in text formats it is possible to create a relation that checks that the result did not change during adjustment of an output with indentation).

During implementation of a large quantity of metamorphic relations for one artifact a similar code duplication problem may occur for the input and/or output metamorphoses. For instance, one output metamorphosis may be used in several relations, thus describing that output data may change in a certain way responding to several different changes of input data. The multiplication function of two numbers may be used as an example: the result may change twice (common output metamorphosis) both in case if the first multiplicand doubles (one input metamorphosis) and in case if the second multiplicand doubles (another input metamorphosis).

If a resolution of this problem is considered separately, the most efficient resolution method would consist in the use of the basic design pattern "Strategy". In such a way it would be possible to create one basic class of metamorphic relation that would: receive objects implementing metamorphosis interfaces; contain implementation of launch of the software artifact being tested; contain implementation of calls of metamorphoses, software artifact launch, and comparison. But, as it would be further demonstrated, in combination with resolution of the metamorphic functions duplication problem the metamorphic relation code duplication problem may be resolved in a different way.

As was already mentioned, the main problem during implementation of metamorphic functions is code duplication because inherently the functions are similar to each other. In fact, every metamorphic function always corresponds to one metamorphic relation and serves only as a method of its connection to the external world: receiving input data for generation, call of metamorphic relation, and return of results. According to the authors, the most efficient way to solve this problem is automatic code generation of metamorphic functions based on the identified metamorphic relations. It is proposed to automatically generate for each metamorphic relation implementation of the corresponding metamorphic function inheriting, for instance, its name. In such a way the developer may concentrate on implementation of the main logic – the logic of metamorphic relations – not considering the specific implementation of the corresponding metamorphic functions.

But if we have already decided to use code generation, then having a set of input and output metamorphoses and the software artifact launch code it is, actually, possible to generate metamorphic relations. Thus, every generated metamorphic relation will contain a code similar to the one that would be contained in the above described basic class of relations, but instead of receiving objects implementing metamorphosis interfaces, instances of specific classes corresponding to this relation will be used. In such a case the developer does not need even manual creation of instances of metamorphic relations with required dependencies – during the generation of the metamorphic function code there is a possibility to generate also the creation and call code for the corresponding metamorphic relation.

Applicability. MTaaS pattern may be used in situations when:

- There is a large number of metamorphic relations to be run in the cloud using serverless architecture;
- It is necessary to avoid duplication of the code for some metamorphic relations and/or functions.

Structure. The structural UML diagram of the proposed architecture pattern is shown in Figure 1.

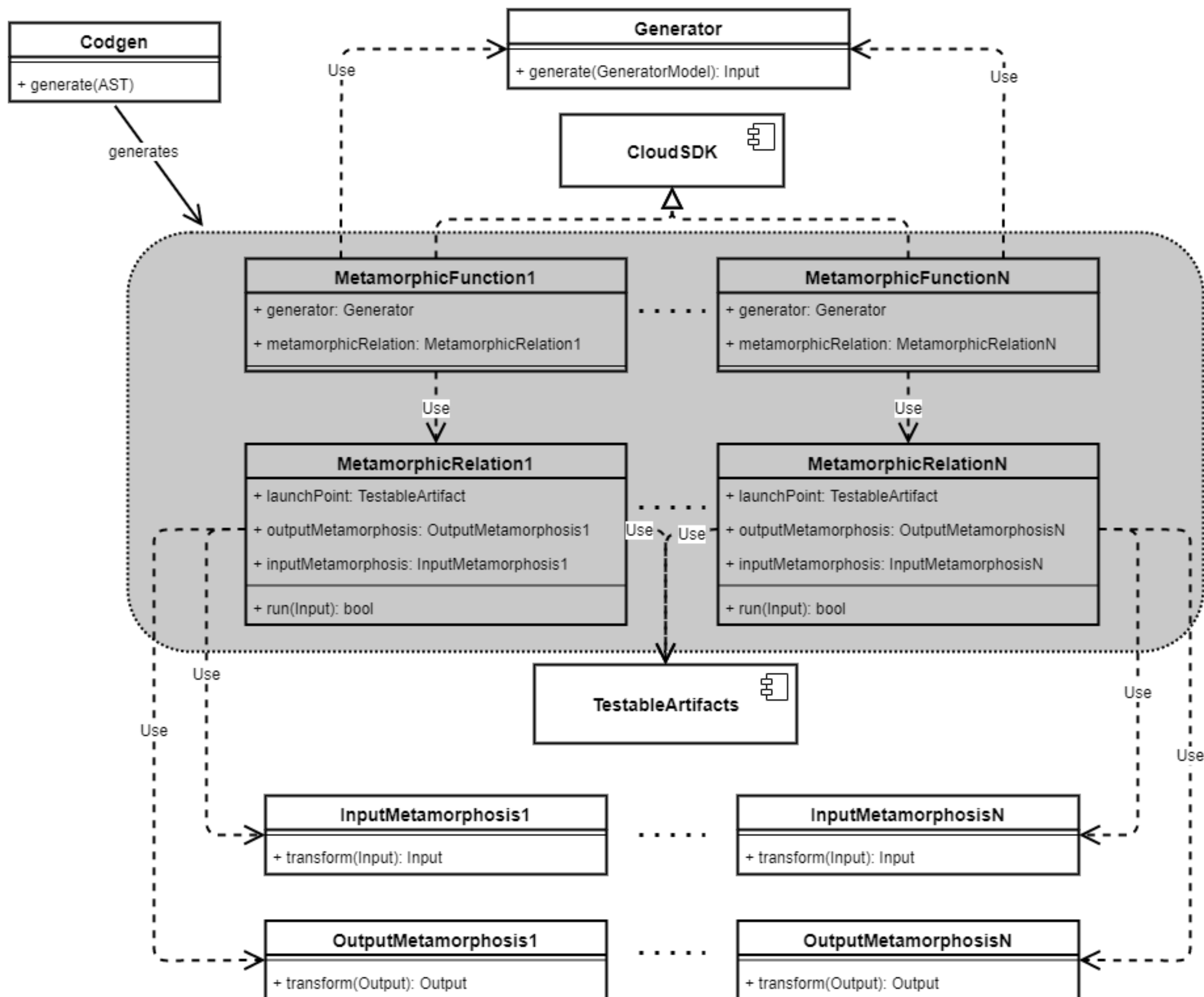

Figure 1. Structural diagram of the proposed MTaaS pattern

Participants.

- InputMetamorphosis1, InputMetamorphosis2, …, InputMetamorphosisN – input metamorphoses provided by the developer;
- OutputMetamorphosis1, OutputMetamorphosis2, …, OutputMetamorphosisN – output metamorphoses provided by the developer;
- TestableArtifact – software artifact being tested and its launch code;
- Generator – input data generator;
- Codgen – code generator;
- CloudSDK – SDK, made available by the cloud provider for implementation of the FaaS pattern;
- MetamorphicRelation1, MetamorphicRelation2, …, MetamorphicRelationN – generated metamorphic relations;
- MetamorphicFunction1, MetamorphicFunction2, …, MetamorphicFunctionN – generated metamorphic functions.

Collaborations.

- The code generator tracks all available input and output metamorphoses – for that purpose various techniques can be used such as annotations, attributes, special contracts, etc.;
- In the same way the code generator finds implementation of the launch of the software artifact being tested;
- The code generator generates metamorphic relations each one containing corresponding input and output metamorphoses and the software artifact being tested;
- The code generator also generates metamorphic functions each one referencing the corresponding metamorphic relation and input data generator;
- Besides, the generated metamorphic functions are also referencing the SDK of the cloud provider thus

implementing corresponding contracts provided by this SDK.

Consequences. Application of the MTaaS pattern has the following consequences:

- Hiding details of metamorphic functions from the developer – thus, the developer concentrates only on implementation of metamorphic relations. This idea is similar to how the architecture pattern FaaS (Function-as-a-Service) hides from the developer details of its code deployment and launching, for that the proposed MTaaS pattern received its name.
- Generated metamorphic relations and metamorphic functions use specific instances of their dependences (metamorphoses for relations, and relations for functions) instead of abstract interfaces thus facilitating the code reading and understanding.
- The code generator may be implemented only once and then used in an unlimited number of projects by connecting as an external library.
- Settings of the software artifact required for its launch are considered as a part of input data that could also participate in input metamorphoses.

The code generator may be extended with various settings, when necessary, or vice versa, simplified. For instance, if it is necessary that a part of metamorphic functions use one type of the Cloud SDK trigger and another part use another trigger, the code generator may be complicated by the corresponding configuration.

Implementation. Specificities of this architecture pattern implementation mainly depend on features of the specific programming language (namely – on specific possibilities of code generation provided by the language: macros, compiler extensions, etc.) and specific cloud provider (what signatures of metamorphic functions must be exported for compliance with FaaS implementation). The next Section discusses reference implementation on the .NET/Azure platform.

Reference Implementation

The reference implementation of the proposed architecture pattern MTaaS was done for the .NET platform and Azure Functions cloud service. The source code of the developed solution is available at https://github.com/yakivyusin/MTaaS.

For the implementation of code generation at the .NET platform starting with release 5.0, the mechanism called source generators is available. The idea of such a mechanism consists in the compiler calling the user code marked with a special attribute. The user code receives at the input the complete project AST and can supplement it with own generated code [19].

In C# language it is possible to identify two idiomatic ways of the proposed architecture pattern implementation – attributive and contract.

In the case of the attributive way, all components of the MTaaS pattern at the user level are marked with attributes provided by MTaaS implementation. Then these attributes are used for the search of all components and their use in generated metamorphic functions and relations.

In the case of the contract way, the MTaaS implementation besides generation of metamorphic relations and functions also generates contracts of components at the user level based on the specified configuration file. Then the developer will need to provide implementation of these contracts. In the C# language the examples of such contracts may include partial classes and methods.

The main advantages of the attributive approach are larger idiomaticity compared to the contract approach (that refers both to compliance with the programming language and the nature of the pattern itself), no need for additional user's configuration files, one component may be marked with several attributes for different metamorphic relations reducing the code duplication. The disadvantage of this approach is a more complicated implementation of the corresponding code generation because it is necessary: to perform the AST analysis for the search of all components marked with attributes; to analyze for every metamorphic relation whether all mandatory components have been marked, and if not – send an error message to the compiler.

Accordingly, the main advantage of the contract approach is simplicity of its implementation: the source generator only generates contracts using the configuration file data, and verification of all implementations may be assigned to a standard compiler (e.g., absence of implementation in a partial method, no value of the type parameter in a generic class, etc.).

The developed reference implementation applies the contract approach but, in the future, it can be extended with the attributive approach providing the user with a choice.

YAML format was used for the configuration file that has the following advantages compared to the JSON format mainly used on the .NET platform: simplified syntax, availability of comments in the format standard. It should be mentioned that whatever JSON document is also a valid YAML document, therefore JSON syntax may be used if desired [20].

An example of the configuration file with comments:

```
  # MR and MF name
- name: MR1
  models:
    # input model according to generic architecture
    input: object[]
    # output model according to generic architecture
    output: System.IO.StringWriter
    # optional – IEqualityComparer for output model
```

```
    output_equality_comparer: Custom.StringWriterComparer
    # generator model according to generic architecture
    generator: MTaaS.Sample.Models.GeneratorModel
  # next MR...
- name: MR2
```

In general, reference implementation consists of five source generators:

- InputMetamorphosis – generates the contract of input metamorphosis for each metamorphic function as partial classes with one partial method (using $.name and $.models.input from the configuration file).
- OutputMetamorphosis – generates the contract of output metamorphosis ($.name and $.models.output).
- InputGeneration – generates the contract for input data generator (also as partial class with partial method) using $.name, $.models.generator and $.models.input.
- MetamorphicRelation – generates the metamorphic relation class; the user needs to provide an implementation of only one partial method – actually, the launch of the artifact being tested.
- MetamorphicFunction – generates an implementation of the metamorphic function.

So far, metamorphic functions are always generated using HttpTrigger (POST method), and it is expected that the input data generator model will be transmitted in the request body in JSON format. It can be extended with further project extension (see Conclusions).

Experiment

To verify the hypothesis that the proposed architecture pattern significantly simplifies implementation of serverless metamorphic testing the same relations as in the study [17] were used in this work.

In the study [17] five metamorphic relations for the program library of tabular collection output were used as an example of implementation of metamorphic testing serverless architecture: increasing and decreasing the collection, field adding and deletion, change of the object order in the collection.

The graphic explanation of the implemented metamorphic relations is provided in Figure 2.

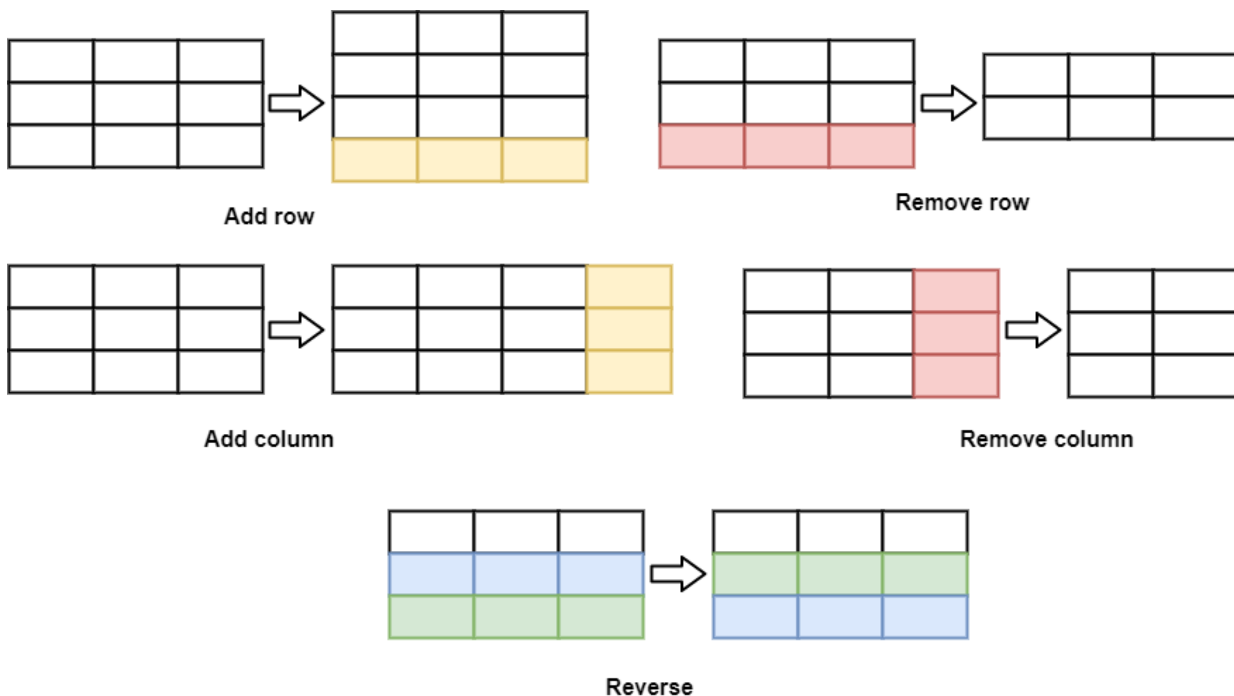

Figure 2. Output metamorphosis for five implemented metamorphic relations

Unlike the software developed in [17], within this experiment rigorous comparison of the objects containing the tabulated output (StringWriter) was implemented according to their content instead of a comparison of individual properties (width, height, individual lines). For this purpose the IEqualityComparer interface was implemented for StringWriter and this implementation was used in the configuration file for every metamorphic relation as $.models.output_equality_comparer.

For comparison of the quality and complexity indicators of the software developed using the MTaaS pattern and software that was implemented just in line with the generic serverless architecture, the following metrics have been used: maintainability index [21], cyclomatic complexity [22], class coupling [23], lines of code.

Results for the specified metrics are provided in Table 1.

Table 1

Code quality metrics of the two developed software

MR	Type	Generator	Input metamorphosis	Output metamorphosis	Relation	Function	Total
Lines of Code (LoC)							
Add row	generic	12/5*	15			8 + 4/5	**26,2**
	MTaaS	4 + 8/5	4	9	7	-	**25,6 (-2%)**
Remove row	generic	12/5	14			8 + 4/5	**25,2**
	MTaaS	4 + 8/5	4	10	7	-	**26,2 (+4%)**
Add column	generic	12/5	14			8 + 4/5	**25,2**
	MTaaS	4 + 8/5	4	11	7	-	**27,6 (+9%)**
Remove column	generic	12/5	13			8 + 4/5	**24,2**
	MTaaS	4 + 8/5	4	10	7	-	**26,6 (+10%)**
Reverse	generic	12/5	18			8 + 4/5	**29,2**
	MTaaS	4 + 8/5	4	11	7	-	**27,6 (-5%)**
Cyclomatic Complexity							
MR	Type	Generator	Input metamorphosis	Output metamorphosis	Relation	Function	**Total**
Add row	generic	3	2			1 + 1	**7**
	MTaaS	1 + 2	1	1	2	-	**7**
Remove row	generic	3	2			1 + 1	**7**
	MTaaS	1 + 2	1	1	2	-	**7**
Add column	generic	3	2			1 + 1	**7**
	MTaaS	1 + 2	1	1	2	-	**7**
Remove column	generic	3	2			1 + 1	**7**
	MTaaS	1 + 2	1	1	2	-	**7**
Reverse	generic	3	3			1 + 1	**8**
	MTaaS	1 + 2	1	1	2	-	**7 (-12,5%)**
Class Coupling							
MR	Type	Generator	Input metamorphosis	Output metamorphosis	Relation	Function	**Total**
Add row	generic	4	8			9	**21**
	MTaaS	3	2	2	3	-	**10 (-52%)**
Remove row	generic	4	8			9	**21**
	MTaaS	3	2	2	3	-	**10 (-52%)**
Add column	generic	4	10			9	**23**
	MTaaS	2	1	2	2	-	**7 (-69%)**
Remove column	generic	4	10			9	**23**
	MTaaS	2	1	2	2	-	**7 (-69%)**
Reverse	generic	4	8			9	**21**
	MTaaS	3	2	2	3	-	**10 (-52%)**
Maintainability Index							
MR	Type	Generator	Input metamorphosis	Output metamorphosis	Relation	Function	**Min**
Add row	generic	75	69			71	**69**
	MTaaS	min(100, 68)	100	100	77	-	**68 (-1%)**
Remove row	generic	75	69			71	**69**
	MTaaS	min(100, 68)	100	100	77	-	**68 (-1%)**
Add column	generic	75	72			71	**71**
	MTaaS	min(100, 68)	100	100	77	-	**68 (-4%)**
Remove column	generic	75	69			71	**69**
	MTaaS	min(100, 68)	100	100	77	-	**68 (-1%)**
Reverse	generic	75	65			71	**65**
	MTaaS	min(100, 68)	100	100	77	-	**68 (+5%)**

The following relationships can be noted based on the data provided:

1) *Lines of code* – depending on metamorphic relation the total number of lines of code may both increase or decrease if the MTaaS pattern is applied. Such a result can be judged as expected because when the MTaaS pattern is used the number of classes the logic is subdivided into increases and each class adds minimum 3 additional lines (namespace declaration, class declaration, and signature of the highlighted method). It should be noted that software developed within the study implements the more complex comparison of output data that impacts the number of lines of code too. Also, during the framework implementation of the attributive approach, this indicator may be improved because this approach is better oriented at reduction of code duplication.

2) *Cyclomatic complexity* – this indicator, actually, is independent of the MTaaS pattern use that is also expected because decomposition improvement of the metamorphic relation code and automatic generation of metamorphic functions do not much influence the program control dataflow graph.

3) *Class coupling* – this metric improves when the proposed pattern is applied, mainly owing to the automatic generation of a part of the code. Besides automatic generation, the subdivision of one class of metamorphic relation into three classes (metamorphoses and the relation itself) also positively influences this metric because in practice it is easier to operate three classes with CC=3 than one class with CC = 6 (despite the fact that the total CC indicator of the subdivided classes may be larger).

4) *Maintainability index* – despite the fact that formally for four relations out of five the minimum value of this indicator has worsened, it is necessary to pay attention to the distribution of values between classes. When simply generic architecture was used, all three components have quite coinciding MI values; and when the MTaaS pattern was used large dispersion of values was observed from the best value equal to 100 to the minimum obtained value. So, each metamorphosis class received the best possible value for the MI metric (100); metamorphic relation classes improved their values compared to the generic architecture; the minimum value was determined by the input data generator. Thus, it can be concluded that application of the MTaaS pattern improves the MI metric.

So, application of the proposed architecture pattern MTaaS improves the key metrics of the code quality and, accordingly, the hypothesis that this pattern simplifies the implementation of the serverless metamorphic testing can be considered as confirmed.

Conclusions

Conducted analysis of the problems occurring during implementation of metamorphic testing using serverless computing (following the generic architecture) demonstrated expediency of developing additional architecture patterns application of those promotes efficient resolution of the identified problems. In the study the MTaaS (Metamorphic Testing-as-a-Service) architecture pattern was proposed that is based on the idea of metamorphic relation decomposition into individual parts together with automatic code generation of the relations' and functions' bodies.

The proposed architecture pattern envisages:

- Subdivision of the metamorphic relation code into separate input and output metamorphoses (done by the user); launch component if the software artifact being tested (done by the user); and the body of metamorphic relation using them (code generation).
- Code generation for metamorphic function.

So, application of the MTaaS pattern simplifies the implementation of serverless metamorphic testing (owing to code generation and ready-to-use decomposition scheme) and improves the metrics of the software code quality.

The reference implementation of the proposed architecture pattern together with the code generation was developed on the .NET platform for the Azure cloud provider. Using such implementation, comparison of the key quality metrics of the software code with the MTaaS pattern application and without was performed. The conducted experiment has demonstrated improvement of the class coupling and maintainability index metrics without worsening other metrics that confirmed the effectiveness of the proposed pattern.

Further development of this study is primarily in the technical field: implementation of an alternative, attributive approach; application of the reference implementation of the proposed pattern for other programming languages and cloud provides, etc.

References

1. Capgemini, Sogeti, Microfocus, “World Quality Report 2021”. URL: https://www.capgemini.com/research/world-quality-report-wqr-2021-22/.
2. Weyuker E., “The oracle assumption of program testing,” in Proc. of 13th International Conference on System Sciences, pp. 44-49 (1980).
3. Barr T., Harman M., McMinn P., Shahbaz M., Yoo S., “The oracle problem in software testing: a survey,” IEEE Transactions on Software Engineering Vol. 41 (5), pp. 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785.
4. 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 University of Science and Technology, Hong Kong (1998).
5. Zhou Z.Q., Zhang S., Hagenbuchner M., Tse T.H., Kuo F.-C., Chen T.Y., “Automated functional testing of online search services” Software Testing, Verification and Reliability Journal Vol. 22 (4), pp. 221–243 (2012). https://doi.org/10.1002/stvr.437.
6. Zhou Z.G., Tse T.H., Kuo F.-C., Chen T.Y. “Automated functional testing of web search engines in the absence of an oracle” Technical Report TR-2007-06, Department of Computer Science, The University of Hong Kong, Hong Kong (2007).
7. Zhou Z.Q., Xiang S., Chen T.Y. “Metamorphic testing for software quality assessment: A study of search engines” IEEE Transactions on Software Engineering Vol. 42 (3), pp. 264–284 (2016). https://doi.org/10.1109/TSE.2015.2478001.
8. Tao Q., Wu W., Zhao C., Shen W. “An automatic testing approach for compiler based on metamorphic testing technique” in Proc. of 17th Asia Pacific Software Engineering Conference (APSEC), pp. 270–279 (2010). https://doi.org/10.1109/APSEC.2010.39.
9. Le V., Afshari M., Su Z. “Compiler validation via equivalence modulo inputs” in Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 216–226 (2014). https://doi.org/10.1145/2666356.2594334.
10. Kuo F.-C., Liu S., Chen T.Y. “Testing a binary space partitioning algorithm with metamorphic testing” in Proc. of the 2011 ACM Symposium on Applied Computing, pp. 1482–1489 (2011). http://dx.doi.org/10.1145/1982185.1982502.
11. Jameel T., Mengxiang L., Liu C. “Test oracles based on metamorphic relations for image processing applications” in Proc. of 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1–6 (2015). https://doi.org/10.1109/SNPD.2015.7176238.
12. Pullum L.L., Ozmen O. “Early results from metamorphic testing of epidemiological models” in Proc. of ASE/IEEE International Conference on BioMedical Computing (BioMedCom), pp. 62–67 (2012). https://doi.org/10.1109/BioMedCom.2012.17.
13. Ramanathan A., Steed C.A., Pullum L.L. “Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics” in Proc. of ASE/IEEE International Conference on BioMedical Computing (BioMedCom), pp. 68–73 (2012). https://doi.org/10.1109/BioMedCom.2012.18.
14. Segura S., Fraser G., Sanchez A.B., Ruiz-Cortes A. “A survey on metamorphic testing” IEEE Transactions on Software Engineering Vol. 42 (9), pp. 805-824 (2016). https://doi.org/10.1109/TSE.2016.2532875.
15. Chen T.Y., Kuo F.-C., Liu H., Poon P.-L., Towey D., Tse T.H., Zhou Z.Q. “Metamorphic testing: A review of challenges and opportunities” ACM Computing Surveys Vol. 51 (1), pp. 4:1-4:27 (2018). https://doi.org/10.1145/3143561.

16. Troup M., Yang A., Kamali A.H., Giannoulatou E., Chen T.Y., Joshua W. K. "A cloud-based framework for applying metamorphic testing to a bioinformatics pipeline" In Proc. of the 1st International Workshop on Metamorphic Testing, pp. 33–36 (2016). https://doi.org/10.1109/MET.2016.014.

17. Yusyn Y., Zabolotnia T. "Metamorphic Testing and Serverless Computing: A Basic Architecture" preprint, [Online]. Available: https://yakivyusin.github.io/preprints/Serverless.pdf.

18. Gamma E., Helm R., Johnson R., Vlissides J. "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley (1994).

19. Microsoft, "Source Generators". URL: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview.

20. YAML Language Development Team, "YAML Ain't Markup Language (YAML™) version 1.2". Available: https://yaml.org/spec/1.2.2/.

21. Microsoft, "Code metrics – Maintainability index range and meaning" URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning.

22. Microsoft, "Code metrics – Cyclomatic complexity" URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity.

23. Microsoft, "Code metrics – Class coupling" URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-class-coupling.

Рецензія/Peer review : 28.01.2022 р. Надрукована/Printed : 27.02.2022 р.