Empirical Analysis of Object-Oriented Software Test Suite Evolution

Nada Alsolami[1]
Computer Science Department, Al Imam Mohammad Ibn Saud Islamic University, Riyadh, Saudi Arabia

Qasem Obeidat[2]
Computer Science Department
University of Bahrain
Sakheer, Kingdom of Bahrain

Mamdouh Alenezi[3]
Computer Science Department
Prince Sultan University
Riyadh, Saudi Arabia

***Abstract*—The software system is evolving over the time, thus, the test suite must be repaired according to the changing code. Updating test cases manually is a time-consuming activity, especially for large test suites, which motivate the recent development of automatically repairing test techniques. To develop an effective automatic repair technique that reduces the effort of development and the cost of evolution, the developer should understand how the test suite evolves in practice. This investigation aims to conduct a comprehensive empirical study on eight Java systems with many versions of these systems and their test suites to find out how the test suite is evolving, and to find the relationship between the change in the program and the corresponding evolution in the test suite. This study showed that the test suite size is mostly increased, where the test suite complexity is stabilized. The increase (or decrease) in the code size will mostly increase (or decrease) the test suite size. However, the increasing or decreasing in the code complexity is offset by stabilizing the test suite complexity. Moreover, the percentage of the code coverage tends to be increased more than decreased, but in the mutation coverage, the opposite is true.**

***Keywords*—Software; test; code complexity; code coverage; test evolution**

I. Introduction

Software testing is an important and essential step to identify the correctness and quality of software system. In the software testing process, the tester should write one test case or more to check each function of the system. The test case is the smallest meaningful unit of the tests. The result of each test case is either pass or fail. .If test cases are passed (i.e., the actual results = the expected results), then the functionality of a software system corresponding to these passed test cases is working correctly. The test suite is a collection of test cases to test system functionalities. Any software system (S) is divided into two parts: program (P) and test suite (T). All system test cases (Tc) are stored in (T). These test cases are used to check the correctness of all parts of P. The new version of the software system (S') should have a different program (P') and test suite (T').

Software systems evolve and change during their development and maintenance. Even a little change in the software code can affect a large number of test cases [1]. The software system upgrades are accompanied by code refactoring or code evolution. The code refactoring is a process of improving the internal structure of code without changing the external behavior or system functionality [2]. Refactoring process makes the code more readable, does not contain duplications, easier in maintenance, and increase the quality of the code. On other hand, the code evolution is adding new code to add new functionality to the system or deleting/ modifying the existing parts of the code to edit functionality in the system. Code evolution is a continuous process; it may change the system functionality and external behavior.

Software evolution is one of the essential and normal issues required for most existence software throughout their lifetime. The changes in the code make some of the test cases in the current test suite become out of date for the new version of the software. Therefore, the tester must revise all changes on the code to repair the corresponding test cases in the test suite. While the code evolution may happen frequently, it is very hard for a tester to follow all these code evolutions and make the correct decisions as create, delete, and update test cases. Also, it is time-consuming to repair test cases manually, particularly, the large test suite [1]. This motivates researchers to develop automatic test repair techniques. The basic requirement to automate test repair technique is comprehensive understanding of how test suite evolves in practice. Generally, analysis a test suite evolution can help developers to build effective automated test repair techniques. So, this paper is intended to conduct an empirical study to understand and identify how the test suite evolves during the code changes and to create or build a relationship between code changes and the corresponding changes of the test suite. The main goal of the experiment is to provide answers to the following research questions regarding test suites evolution:

RQ1. Is the test suite size increasing/ decreasing/ stabilizing during a software evolution?

RQ2. What is the relationship between source code size and test suite size during a software evolution?

RQ3. Is the test suite complexity increasing/ decreasing/ stabilizing during a software evolution?

RQ4. What is the relationship between source code complexity and test suite complexity during a software evolution?

RQ5. What is the effectiveness of the test suite during a software evolution?

II. Related Works

This section shows some previous studies on the test suite evolution and the co-evolution between code and test.

A. Test Suite Evolution

Elbaum *et al.* examined the impact of software evolution on code coverage information [3]. This examination showed that a little change in software code can give large impact on code coverage information. This impact increases as the degree of change increases and it could be difficult to predict.

One study presented a technique for studying test-suite evolution [1]. There were 88 program versions studied, 14,312 test cases, and 17,427 test changes (i.e., modification, addition, and deletion). This study provided initial insight on how test cases are added, removed, and modified in practice. It focused on test repair and investigated the characteristics of deleted and added test cases and implemented technique within a tool called TestEvol [4]. TestEvol is a tool which enables the systematic study of test-suite evolution for Java programs and JUnit test cases. In [1] and [4], the researchers showed that the test modifications tend to be complex and hard-to-automate. The most important results were, firstly: the occurring of non-repair test modification nearly four times as frequently as test repairs. In other words, repairing test is a relatively small fraction of the activities performed during test evolution. Secondly: many test cases are not really deleted and added, but rather moved or renamed.

Another approach to study test suite evolution is based on the observation that software developers follow common patterns to identify changes and adapt test cases [5] and [6]. Mirzaaghaei et al. proposed a novel approach for repairing and generating test cases automatically during software evolution. These studies defined a set of algorithms for repairing test cases commonly adopted by software developers and implemented those algorithms on TestCareAssistant (TCA) for evaluation. TCA properly repairs 90% of the compilation errors, where the TCA addressed and generated test cases that cover the same amount of instructions of state of the art techniques.

All aforementioned studies in this section examined the test suite evolution from different aspects, whether from repairing test cases based on following common patterns commonly adopted by software developers or from fixing test oracle or others. However, in this study, we studied the test suite evolution in term of size (RQ1), complexity (RQ3), and effectiveness (RQ5).

B. Co-Evolution between Code and Test

Marsavina *et al.* investigate fine-grained co-evolution patterns of production and test code [7]. This investigation analyzed five open source systems and then generates six patterns. These patterns explain the relationship between the change in code and corresponding test cases in the test suite.

Levin *et al.* have done a large scale study of 61 open source projects to study the relationship between test maintenance and production code maintenance in semantic changes [8]. The most important results were that the test maintenance is individually in each project rather than standardized.

Several researchers studied the nature of the co-evolution between code and test (i.e. synchronously or phased) [9], [10] and [11]. In [9], Lubsen *et al.* used two cases studies: open source system and industrial software system, as they used association rule mining to study the natural of co-evolution. They concluded that within an open source system the development and testing are separate activities, wherein the industrial software system, the developer used test-driven development strategy. In [10] and [11], the researchers proposed three views which are: change history view, growth history view, and test coverage evolution view to study the nature of the co-evolution. In study number [10], the researchers used two open source projects, while in [11], they used two open source projects and one industrial software project. They concluded that the nature of co-evolution depends on the development style that is used to develop a project.

Ens *et al.* create and implement the interactive visual analytics tool for analyzing co-change and co-evolution between code and test [12]. It enables managers and engineers to display 2D and 3D views. In addition, it helps in determining the intensive period of testing and development and determining the development style of the project.

III. Methodology

The main goal of this study is to understand how test suite evolves over the time. Thus, to achieve this goal, several versions of 8 open source Java systems with their test suits were used to investigate different aspects of test-suite evolution. These systems were selected according to many criteria, which are popular, system size, each system has at least 5 versions, and each version has a JUnit test suite. The 8 open source Java systems that used in our empirical study are selected from GitHub (https://github.com/). Table I lists the systems and its versions.

Most researches have studied the relationship between the code and the test suite generally and provided general information about the relationship between the code and the test suite. However, in the current investigation, we studied the relationship between code and test suite in term of size and complexity.

A. Relationship between Code and Test Suite

The test suite is changing and evolving during its lifetime according to the code changes. Therefore, the relationship between the code and its test suite must be investigated. Accordingly, this paper studied the relationship between the code and test suite in terms of size and complexity.

Several metrics are used to determine the size of production code or tests, such as the number of classes, Line of Code (LOC), number of methods, and number of packages. The software complexity focuses on how a piece of code interacts with other pieces. One of the most popular measurements of software complexity is McCabe metric or Cyclomatic complexity metric. The Cyclomatic complexity per method metric is the maximum number of linearly independent paths within method [13].

TABLE. I. SOFTWARE SYSTEMS USED IN THE EMPIRICAL STUDY

Program	Description	Number of versions
Commons-Lang[1]	It is a library which provides extra methods for manipulation of the core classes of the Java API.	11
OGNL[2]	It is an expression language for Java, which using the simpler expression than the Java language.	12
Biojava[3]	It is an open source project for manipulating and processing biological data in Java language.	12
Commons-DBCP[4]	It implements Data Base Connection Pooling service.	5
Californium[5]	It is a Java implementation of the Constrained Application Protocol (CoAP).	9
Assertj[6]	It is a Java library which provides an interface for writing rich and strongly typed assertion to improve maintainability and readability of tests [14].	11
MessagePack[7]	Is a binary serialization (pack) format. This enables a process to exchange data as simple and fast as possible.	10
Aho-Corasick[8]	It is a Java implementation of the Aho-Corasick algorithm.	8

In this paper, we used Eclipse Metrics plug-in 1.3.8 tool to measure the size and complexity metrics because it is one of the most commonly used Java tool in many research either in mobile applications or in other applications [15], [16], [17], and [18]. Moreover, this tool work with the most widely used platforms, such as Windows, Mac, and Linux.

B. Test Suite Effectiveness

The test suite effectiveness (i.e., test suite quality) can be described as the number of bugs in code detected by the test suite. It could be measured in two broad ways: code coverage and mutation testing. The code coverage metric measures the percentage of code covered by the test suite. In this study, we used Eclemma tool because it is giving more accurate results [7] and it is one of the most widely used tools for code coverage [7], [19], [20], [21], [22], and [23].

The mutation testing is a testing technique used to check if the current test cases are able to detect any fault in mutant or the change in software system code [24]. Each modified or mutated version of a program called mutant. In mutation testing, the tool generates multiple mutants of the original program and then executes the test suite on each mutant. If the outputs of the same test case in both, mutant and original program, are different, then the test case detects the fault and the mutant is called killing. However, if the fault is not detected, then the mutant is called surviving. The mutation coverage calculated as a number of detecting mutants over the total number of generating mutants. The detected mutants are killed mutants plus timeout mutants. The mutants called timeout if it causes an infinite loop. The survived mutants are equivalent mutants or not detected mutants. The equivalent mutant is a mutant that acts as original program behavior, and cannot be detected by any test case. The following example explains the equivalent mutants [25]:

Original program:

```
int index = 3;
if (index >= 2)
 return "foo";
```

Mutant program:

```
int index = 3;
if (index > 2)
 return "foo";
```

The effectiveness of the test suite can be measured by the ability of the test suite to detect most mutants. Pitclipse tool is used in this research because it is fast, also it is considered as one of the most popular tools [26]and [27], and it has been used successfully in several studies [28], [29] and [25].

IV. RESULT

In this section, the results of this study will be presented. The results were divided according to the research questions as follows:

A. Code and Test Suite Size Metrics

The size metrics, for both code and test suite, for the eight systems has been illustrated in Fig. 1 according to the number of classes for each version. The number of classes is mostly compatible with the number of line of codes (LOC), so, there is no difference between them.

These results show that the overall percentage of increase in the test suite size within all versions of all systems is equal to 78.6%. This percentage calculated as a number of increased versions over the number of changed versions (i.e. 70 versions). The percentages of stabilizing and decreasing of the test suite size in all versions of all systems are equal 18.5% and 2.9%, respectively. That means the test suite size is mostly increased during its evolution. The percentages of stabilizing and decreasing have been calculated by the same method of calculating the increasing percentage. In all eight systems, there is a harmony which based upon the number of classes' changes between the system code and the test suite as shown in Fig. 1. Here, the compatibility means that any modification to the older version of the system code in terms of increasing or decreasing in the code size is accompanied by the same effect on the test suite size.

In the current investigation, 8 systems with 78 versions have been evaluated. Where, 70 versions have been changed, where the remaining 8 versions are the initial versions of all systems. There are 65 versions out of 70 versions (92.9%) are compatible and most of them have been increased in the size. In the remaining 5 versions (7.1%), there is no compatibility between the changes in code size and test suite size.

B. Code and Test Suite Complexity Metrics

According to the average cyclomatic complexity per method for each version, the complexity metric, for both code and test suite, for the eight systems has been illustrated in Fig. 2. These results show that the overall percentage of

[1] https://github.com/apache/commons-lang
[2] https://github.com/jkuhnert/ognl
[3] https://github.com/biojava/biojava
[4] https://github.com/apache/commons-dbcp
[5] https://github.com/eclipse/californium
[6] https://github.com/joel-costigliola/assertj-core
[7] https://github.com/msgpack/msgpack-java
[8] https://github.com/robert-bor/aho-corasick

stabilizing or increasing in the test suite complexity in all versions of all systems is equal to 94.3% (91.4% stabilizing and 2.9% increasing). On other hand, the percentage of decreasing in the complexity of the test suite in all systems is equal to 5.7%, which means that the test suite complexity is mostly stabilized during software evolution. All previous percentages calculated by the same method used in the previous Section 4.1, besides complexity metric rather than size metric.

In more detail analysis, there are only four versions that increase in the code complexity, where the test suite complexity for one version is decreased and in the other three versions it was stabilized. In addition, there is only one version where the complexity of the code has decreased, but the test suite complexity has stabilized. This means that the increasing or decreasing in code complexity is offset by stabilizing in the test complexity, this relationship achieved by 80%. Furthermore, the code complexity stabilization means stabilization in test complexity, this relationship achieved by 85.7% for all versions of within systems.

C. Test Suite Effectiveness

In this empirical study, the code coverage and mutation coverage was used to predict the quality of the system's test over the time. The code coverage and mutation coverage metrics results are explained in the following subsections, 4.3.1 and 4.3.2.

a) Code Coverage: The results for the code coverage metric for all systems have been illustrated in Fig. 3. The code coverage is increased in 32 versions (45.7%), stabilized in 20 versions (28.6%), and decreased in 18 versions (25.7%). These results show that the code coverage tends to increase more than it is stabilized or decrease during the systems improvement. All previous percentages calculated by the same method used in Section 4.1, besides using code coverage metric rather than size metric.

b) Mutation Coverage: The results for the mutation coverage metric for all systems have been illustrated in Fig. 4. The mutation coverage is increased in 22 versions (31.4%), stabilized and decreased in 24 versions (34.3%). These results show that the mutation coverage tends to stabilize or decrease more than it is increased during the systems improvement. All previous percentages calculated by the same method used in the previous Section 4.1, besides mutation coverage metric rather than size metric.

V. Discussion

In this section, we will discuss the results and we will answer the research questions.

A. Size

The test suite size tends to increase or stabilize in all versions of systems, except for the second version of CommonsDBCP and sixth version of Californium. In these two versions, the test suite size is decreased, this may due to many reasons, such as remove some test cases (i.e., redundant test cases), restructuring the test cases by merge two test cases within one test case, or there are some changes should be considered on the source code. The answer to the first research question (RQ1) of this study, the test suite size is often increased over the time by adding new test cases. This addition caused by the developer who has frequently adding new functionalities to the systems and fixing new critical defects that are found.

As shown in Fig. 1, for all eight systems, there is compatibility between the changes in code size and test suite size. In other words, the increasing, decreasing, or stabilization in the code size leads to an increase, decrease, or stabilize the test suite size, respectively. This relationship achieved in 65 versions (92.9%), while the remaining 5 versions do not satisfy this relationship (7.1%). This is considered an answer to the research question (RQ2). For example, there is an incompatibility between the code size and test suite size for the third and eighth versions of Californium because the test suite may be improved by restructuring test cases or restructuring the source code, where the number of classes of the source code was decreased, while the LOC was increased.

B. Complexity

The complexity results shown in Fig. 2 can be used to answer the research questions (RQ3) and (RQ4). Where, the test suite complexity is mostly stabilized during the software evolution, even as the size of the test suite increases. Moreover, the percentage of decreasing in the test suite complexity was greater than the percentage of increase. Here, the test suite may evolve by adding more methods rather than extend the existing methods (i.e., increase nodes and edges). In most versions, the code complexity was stabilized. The increase, decrease, or stabilization in code complexity is offset by stabilizing in test suite complexity. This is because the test case just calls the code methods, which do not increase the test complexity (i.e. do not increase the number of linearly independent paths within the test method).

In general, the size of the test suite does not increase and evolve by adding new functionalities (methods or classes) to the source code only, but also by improving the current test cases or adding more test cases for the current functionalities. As the first versions of the system always need frequent improvement processes because the developer will understand the functional requirements better by the time, particularly, after the system deployment in real life. In parallel, the improvement may consider the non-functional requirements that effect on the system quality, where the codes for both system and test suite should be written in high quality and in a professional way for the latest versions and it will be more stabilized.

C. Test Suite Effectiveness

In this paper, the test suite effectiveness and quality were measured by two metrics: code coverage and mutation coverage to answer the research question (RQ5).

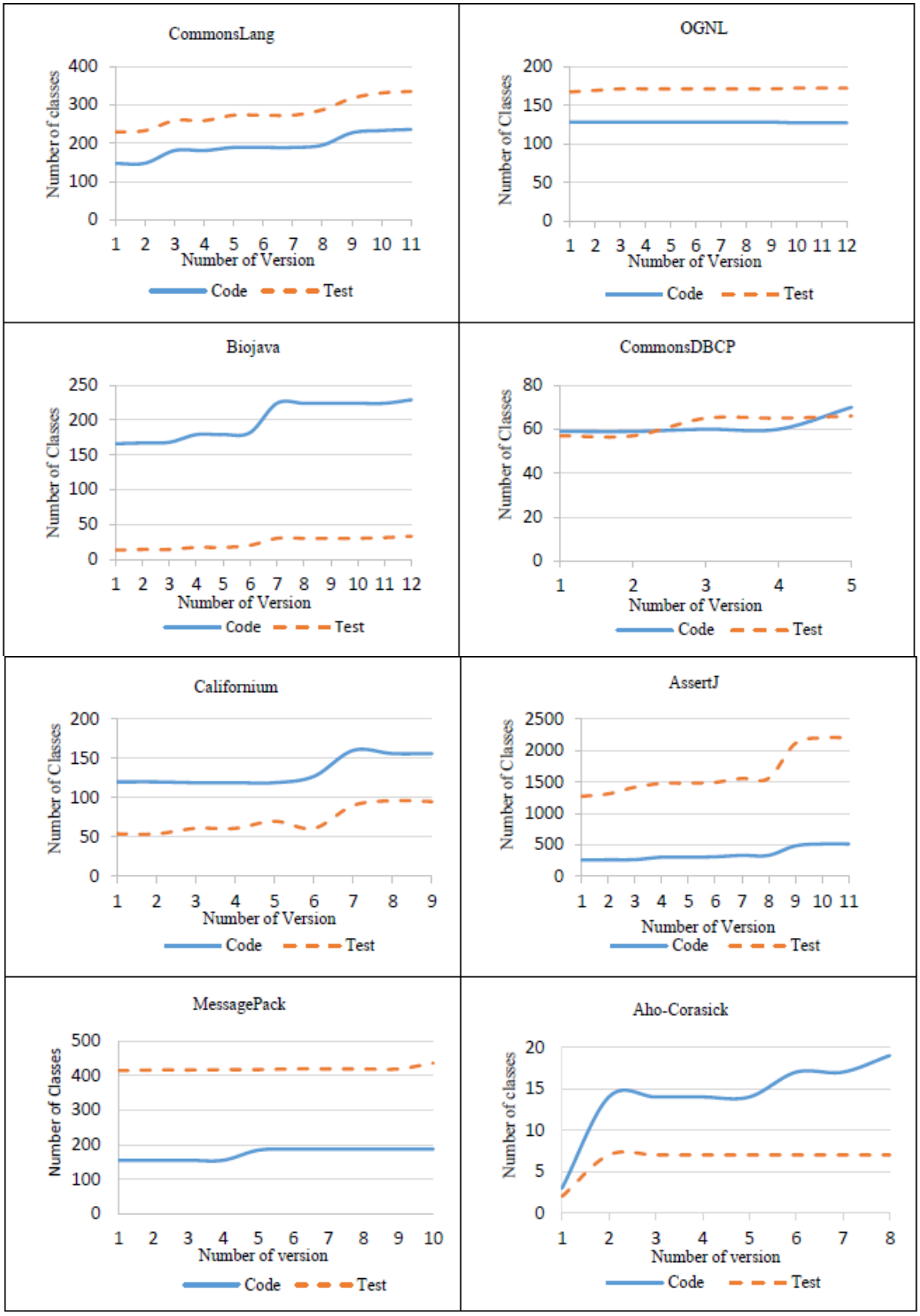

Fig. 1. The Size of the Code and Test Suite for All Systems Versions.

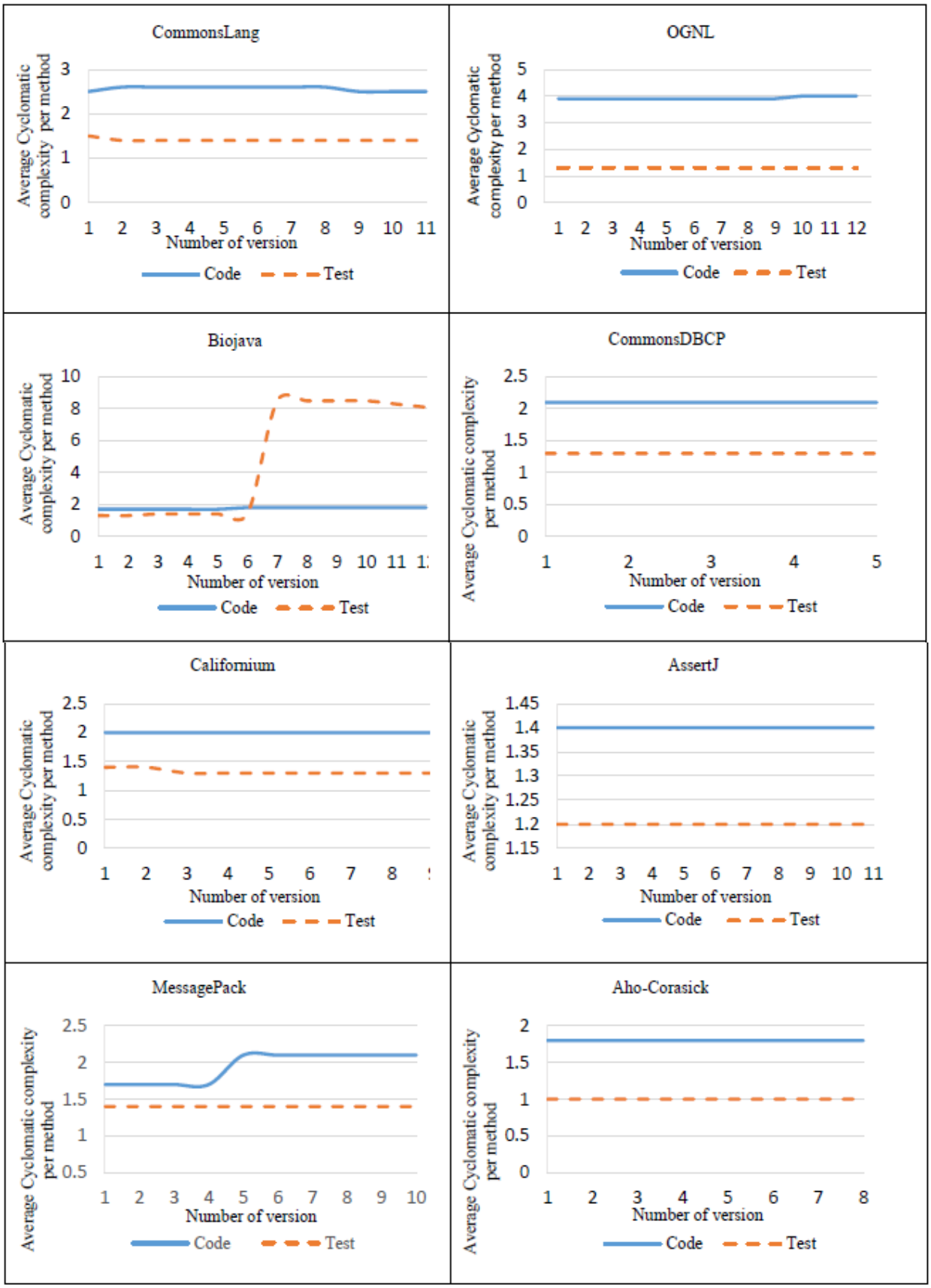

Fig. 2. The Complexity of Code and Test Suite for All Systems Versions.

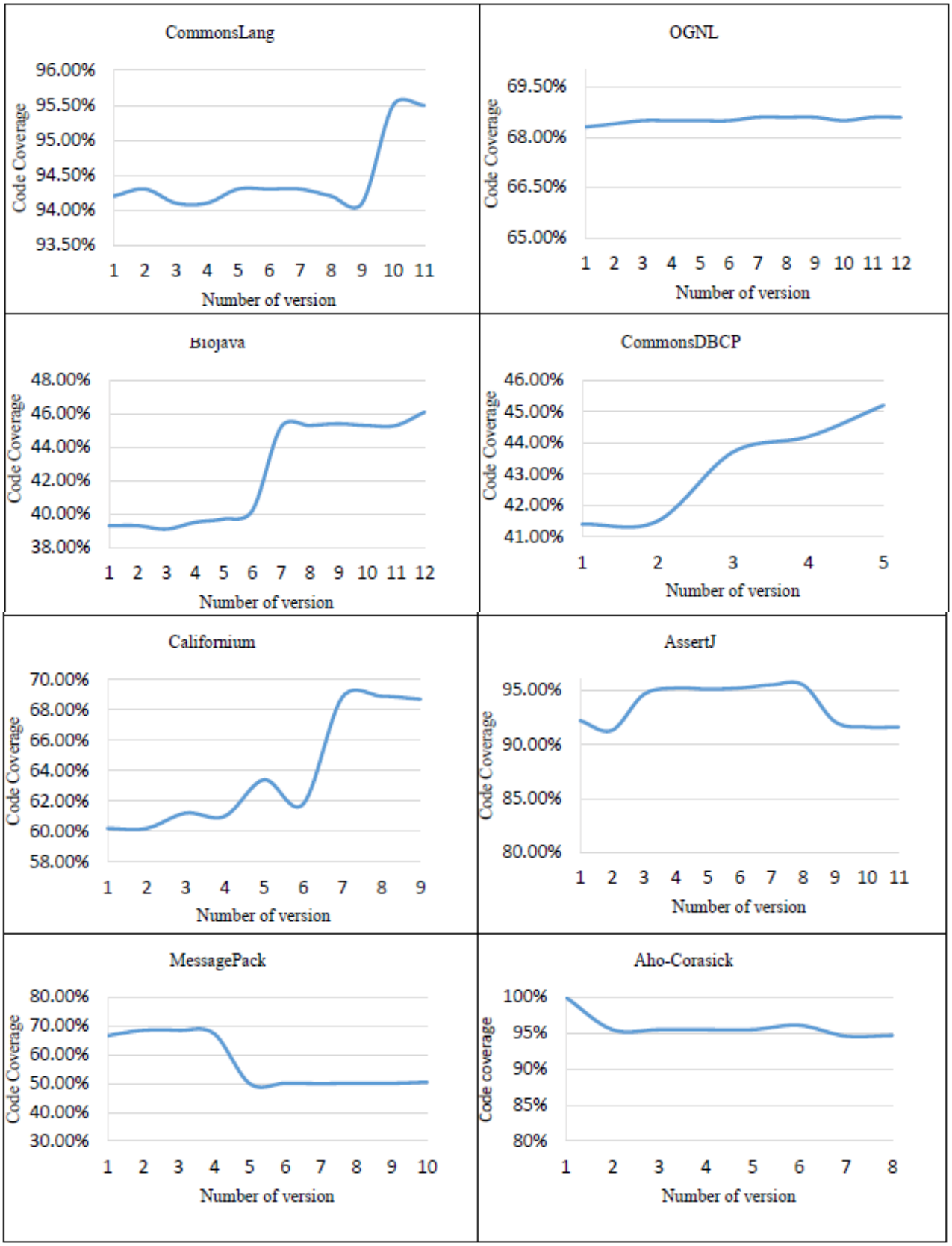

Fig. 3. The Code Coverage for All Systems Versions.

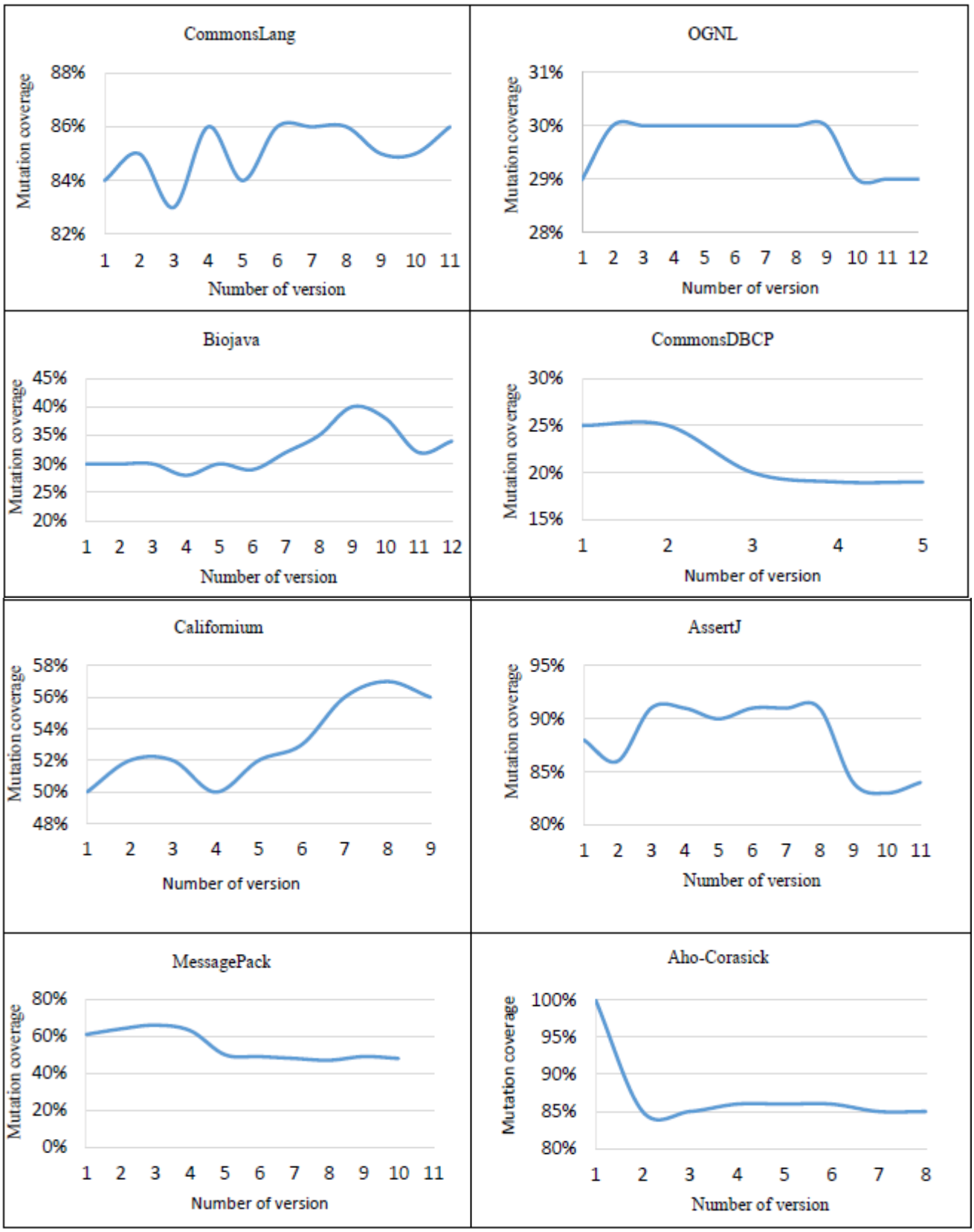

Fig. 4. The Mutation Coverage for All Systems Versions.

The average of the code coverage for all versions in all systems is 71%. This percentage calculated as total code coverage in all versions over the number of versions (i.e. 78 versions). Furthermore, the percentage of the code coverage mostly increase rather than stabilize or decrease within the system versions, as shown in Fig. 5. This indicates that the test suite improved over the time by adding more effective test cases. In a few versions, the code coverage decreased because of some reasons, such as the developer may add new code (i.e., functionality) without adding new test cases. In other words, the code size is increased while the test suite size is stabilized.

When the test suite size is increased more than the code size (i.e., number of classes), the percentage of the code coverage is increased or stabilized by 76.2%. In contrast, when the code size is increased more than the test suite size, then the percentage of the code coverage is decreased by 38.5%. In sum, 76.5% of all versions are increased or stabilized regarding the percentage of their code coverage when increasing the test suites size. This indicates that the percentage of the code coverage can be improved by adding more test cases, to test untested classes and new functionalities added to the source code. All previous percentages have been calculated by the number of versions that satisfied the relationship over the number of versions that satisfied and did not satisfy this relationship.

The average of the mutation coverage of all versions in all systems is 57%. This percentage calculated as total mutation coverage in all versions over the number of versions (i.e. 78 versions). As shown in Fig. 6, the percentage of either decreasing or stabilizing mutation coverage was (34.3%), where it was a little bit greater than the percentage of increasing (31.4%).

The mutation coverage percentage increases as timeout mutants increase, and /or as the test suite size increases to kill more mutants. On the other side, the mutation coverage decreases as survived mutants and equivalent mutants increase. In general, the percentage of increasing and stabilizing for the mutation coverage of all systems versions, at the test suits size increasing or stabilizing was about 66.2%. This percentage has been calculated by the number of versions that satisfied the relationship over the number of versions that satisfied and did not satisfy this relationship. The mutation coverage for the third, fifth, ninth, and tenth versions of CommonsLang and the fourth version of Californium was decreased. This is may be due to the equivalent mutants that affect mutation coverage and causing its decrease. In addition, in CommonsDBCP, MessagePack and Aho-corasick systems the mutation coverage percentage was high but it decreased after a while, this is maybe because the Pitclipse tool wrongly deals with the mutants as a timeout mutants (i.e. infinite loop) which causes increase in mutation percentage in the first versions [25].

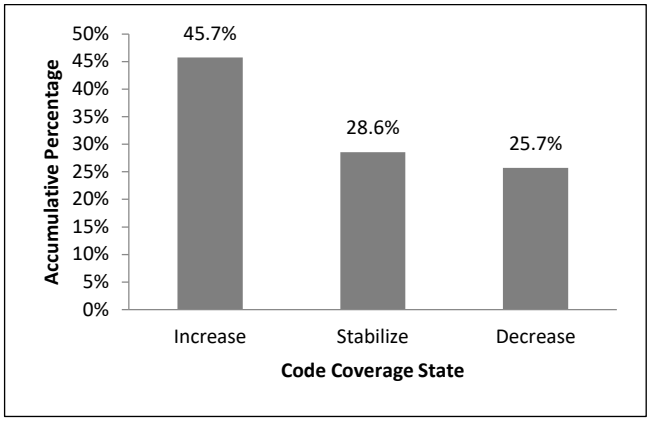

Fig. 5. The Percentage of Accumulative Code Coverage State for All Systems Versions.

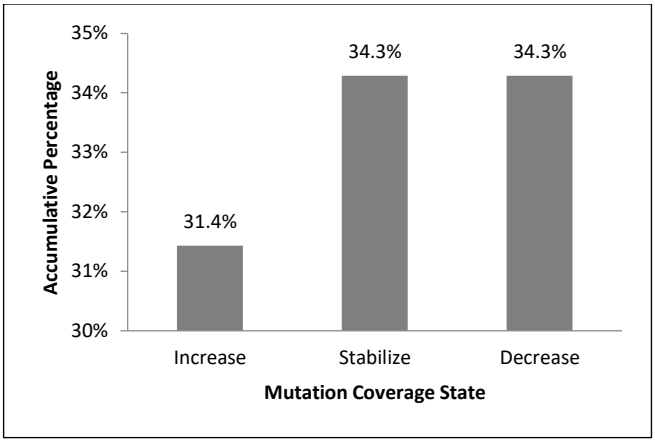

Fig. 6. The Percentage of Accumulative Mutation Coverage State for All Systems Versions.

VI. Conclusion and Future Work

The test suite size mostly increases or stabilizes in program versions. However, the test suite complexity mostly stabilizes and sometimes decreases. The change (i.e. increase or decrease) or stability in the code size is often accompanied by the same change (i.e. increase or decrease) or stability in the test suite size. Often, the complexity of the code and test is stable and does not change between the versions of the system, but in a few cases, it increase or decrease. In these cases, the increase, decrease, or stability in the complexity of the code, offset by stabilization of the complexity of the test.

The code coverage and mutation coverage metrics were measured by using Eclemma and Pitclipse tools, respectively, to evaluate the effectiveness of the test suite. In code coverage, the percentage of increasing the code coverage in program versions is more than the percentage of decreasing. However, in mutation testing, the percentage of decreasing mutation coverage is more than the percentage of increase.

We are planning to extend the current empirical study by engaging and evaluating more well-known Java open-source systems, especially the large ones. Relying on the new results we will develop and build an automated test suite repairing tool. This tool enables a software tester to update the test suite automatically by generating new test cases, deleting, or updating some existence test cases.

REFERENCES

[1] L. S. Pinto, S. Sinha, and A. Orso, "Understanding Myths and Realities of Test-suite Evolution", In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, p. 33. ACM, New York, NY, USA, 2012, pp. 33:1–33:11.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.

[3] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information", in Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, pp. 170–179.

[4] L. S. Pinto, S. Sinha, and A. Orso, "TestEvol: A tool for analyzing test-suite evolution", 35th International Conference on Software Engineering (ICSE), 2013, pp. 1303–1306.

[5] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting Test Suite Evolution through Test Case Adaptation", in Verification and Validation IEEE Fifth International Conference on Software Testing, 2012, pp. 231–240.

[6] M. Mirzaaghaei, "Automatic Test Suite Evolution", in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, New York, NY, USA, 2011, pp. 396–399.

[7] C. Marsavina, D. Romano, and A. Zaidman, "Studying Fine-Grained Co-evolution Patterns of Production and Test Code", in IEEE 14th International Working Conference on Source Code Analysis and Manipulation, 2014, pp. 195–204.

[8] S. Levin and A. Yehudai, "The Co-evolution of Test Maintenance and Code Maintenance through the Lens of Fine-Grained Semantic Changes", in IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 35–46.

[9] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production #x00026; test code", in 6th IEEE International Working Conference on Mining Software Repositories, 2009, pp. 151–154.

[10] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v Deursen, "Mining Software Repositories to Study Co-Evolution of Production #x00026; Test Code", in and Validation 1st International Conference on Software Testing, Verification, 2008, pp. 220–229.

[11] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining", Empir. Softw. Eng., vol. 16, no. 3, pp. 325–364, Jun. 2011.

[12] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "ChronoTwigger: A Visual Analytics Tool for Understanding Source and Test Co-evolution", in Second IEEE Working Conference on Software Visualization, 2014, pp. 117–126.

[13] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity", IEEE Trans. Softw. Eng., vol. 17, no. 12, 1991, pp. 1284–1288.

[14] "AssertJ". [Online]. Available: http://joel-costigliola.github.io/assertj/. [Accessed: 09-Sep-2018].

[15] D. Franke and C. Weise, "Providing a Software Quality Framework for Testing of Mobile Applications", in Verification and Validation 4th IEEE International Conference on Software Testing, 2011, pp. 431–434.

[16] Shalini and S. I. Hassan, "An empirical evaluation of the impact of aspectization of cross-cutting concerns in a Smart-phone based application", in International Conference on Computing for Sustainable Global Development (INDIACom), 2014, pp. 448–454.

[17] J. Hernandez, A. Kubo, H. Washizaki, and F. Yoshiaki, "Selection of metrics for predicting the appropriate application of design patterns", In Proceedings of the 2nd Asian Conference on Pattern Languages of Programs, p. 3. ACM, 2011.

[18] T. Pessoa, F. Brito, M. P. Monteiro, and S. Bryton, "An Eclipse Plugin to Support Code Smells Detection", arXiv preprint arXiv:1204.6492, 2012.

[19] L. Mariani and F. Pastore, "MASH: A tool for end-user plug-in composition", In Proceedings of the 34th International Conference on Software Engineering, pp. 1387-1390. IEEE Press, 2012.

[20] S. Pathy and D. S. Baboo, "Analysis of code coverage metrics using eCobertura and EclEmma: A case study for sorting programs", vol. Volume 4, no. Issue 2, 2016, pp. 121–130.

[21] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report)", In IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 380-389.

[22] P. Dhareula and A. Ganpati, "Open Source Code Coverage Tools for Java: A Comparative Analysis", Indian Journal of Science and Technology, Vol 9(32), 2016, pp. 1–5.

[23] A. Bergel, V. Peña-Araya, and T. Kuhn, "Controlled Experiment to Assess a Test-Coverage Visualization: Lesson Learnt", 2015.

[24] "Mutation Testing: Complete Guide". [Online]. Available: https://www.guru99.com/mutation-testing.html. [Accessed: 17-Apr-2018].

[25] O. Alfsson, "An analysis of Mutation testing and Code coverage during progress of projects", Bachelor's thesis, Umeå University, p. 22.

[26] S. Rani, B. Suri, and S. K. Khatri, "Experimental comparison of automated mutation testing tools for java", in 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 2015, pp. 1–6.

[27] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java (demo)", In Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 449–452.

[28] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness", In Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 435–445.

[29] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive Mutation Testing", IEEE Trans. Softw. Eng., 2018, pp. 1–1.