An Approach to Testing Banking Software Using Metamorphic Relations

Karishma Rahman
Gianforte School of Computing
Montana State University
Bozeman, MT, USA
karishma.rahman@student.montana.edu

Clemente Izurieta
Gianforte School of Computing
Montana State University
Bozeman, MT, USA
clemente.izurieta@montana.edu

***Abstract*—Software systems used for banking are crucial for daily operations and are considered to be part of critical infrastructure; however, testing the functions of these highly reusable systems can be difficult due to the project's complexity and the absence of a reliable oracle. In software testing, the Oracle problem directs to the difficulty of deciding whether the software's observed behavior is correct. To address this issue, we suggest utilizing metamorphic testing (MT), which tests the banking system's functionalities based on their properties. Metamorphic testing is a software testing technique where multiple inputs are generated for a program, then those inputs are transformed based on a pre-defined set of rules. The resulting outputs are then compared to the original outputs to verify that the program works correctly. Metamorphic relations (MRs) are a fundamental concept in metamorphic testing. They define the relationships between the input and output of a system under test and specify how they should change in response to input transformations. Through a case study, we introduce new metamorphic relations to test banking functions and demonstrate the effectiveness of using these MRs. The study results indicate that this is a feasible and efficient approach using an alternative to a test oracle when testing complex E-type (i.e., real-world) software.**

***Index Terms*—Metamorphic Testing, Oracle Problem, Banking Software, Mutation Testing**

I. Introduction

Advances in science and technology have led to significant evolution in the software industry in recent years [1]. With the increasing demand for E-type software in modern society, most applications involve complicated scientific calculations and data processing, necessitating software engineers to ensure that they meet all the requirements for reliability [2]. E-type systems refer to real-world systems [3]. One of Lehman's [2] Laws states that E-type systems are constantly evolving, and their complexity is continuously increasing unless something is purposely done to minimize it. Banking systems are highly reusable E-type systems and their functionality directly affects the growth of the commercial banking industry. Therefore, developing and testing banking software are crucial phases in the software lifecycle for the growth of the industry, and efficient validation and verification techniques are necessary for maintaining the reusability of these banking applications. Consumer demands for banking software have risen, leading to more complex projects and further research into various aspects of banking software [1]. Software testing is a continuous process throughout the banking system project lifecycle and is essential for its progress. The testing of banking systems requires high complexity, security, and accuracy, and customers expect tools for easy transactions and access to financial organizations' services [4]. Banking software has complex designs and multi-layered workflows and offers various features and functions, handling sensitive data like customers' financial and personal information [5]. Therefore, software testing for banking applications must be precise, as any lack of test coverage can lead to data breaches, loss of funds, banking fraud, and other criminal activities [4].

Testing is a crucial aspect of the development process of banking software to ensure that the system behaves correctly. Testing bears more than 50% of the total software development costs, as it is an expensive, time-consuming, and complex activity [6]. The process of testing is often prone to human error. Creating dependable software systems remains an ongoing challenge, and researchers and practitioners continuously explore more efficient methods to test software [1]. Test cases are performed on the system under test during the testing process. A test oracle, either automated or manual, is then used to determine whether it acted as anticipated [1]. In either case, the actual output is compared with the expected outcome.

The challenge of Test Oracle arises when testing complex software. It occurs when it is difficult to determine whether the program outputs on test cases are correct [7]. Banking software often has intricate functionality, which can exacerbate the Oracle problem in their system. For instance, with an electronic payment service, there may be situations where the consumer needs clarification on how much should be charged for a given input. The challenge becomes even more pronounced when the payment concerns transfer charges among different bank accounts or currency exchange applications.

Metamorphic Testing (MT) is a technique that has proven helpful in certain circumstances to address the challenge of the oracle problem [8]. The principle behind Metamorphic Testing (MT) [9] is that it might be easier to analyze the relations between the results of multiple test executions, which are referred to as metamorphic relations (MRs), rather than specifying the input-output behavior of a system [9]. MT employs MRs to determine system properties, which automatically alter the initial test input into follow-up test input [9]. If the system

fails to meet the MRs when tested with the initial and follow-up input, it is inferred that it is defective [11].

A significant amount of study has focused on creating Metamorphic Testing (MT) methods for specific areas such as computer graphics, web services, and embedded systems [13]. Our research aims to apply MT to tackle the test oracle problem in banking software. We aim to systematically define metamorphic relations that capture banking function properties (i.e., characteristics that are compromised when the system is at risk) and automate testing using these metamorphic relations.

We examine how MT applies to banking software testing and convey a case study. This paper delivers the following contributions:

- An approach to investigate and uncover the essential points when utilizing MT to test banking software.
- A list of new MRs for banking functions.
- To show the applicability of the proposed MRs, we conduct a case study on bank software functionalities. The study indicates the relevance of using MT to test banking functions. We also employed mutation analysis to assess the efficiency of the MT approach.

The rest of the paper is organized as follows. Section II introduces underlying concepts related to MT and mutation analysis. Section III presents a framework of MT for banking software and reports on a case study where MT tests major banking functions. Section IV discusses the results of the case study. The next Section V discusses the threats to validity of the experiment. Section VI discusses the related works done in this area. Lastly, Section VII concludes the paper by pointing out potential future work.

II. Background

This section introduces relevant literature and concepts related to Banking application testing, MT, and mutation analysis.

A. Testing Bank Application

The banking industry has changed remarkably due to rapidly growing and innovative technology. Due to the complicated features integrated into banking software, it is regarded as one of the most sophisticated and complex enterprise solutions [4]. The daily transactions carried out through the banking system require accurate data, high scalability, and reliability. Therefore, testing this software under various conditions ensures its efficiency. Moreover, the banking sector requires robust reporting mechanisms to record and instantly monitor transactions and user interactions [5]. Testing is essential to ensure that banking software functions well and effectively. Functional testing of banking software is distinct from standard software testing as these applications handle customers' financial data and money, making it necessary to conduct thorough testing [4]. No critical business scenario should be overlooked during testing.

B. Metamorphic Testing

Metamorphic testing is a technique that can help solve the well-known test oracle problem. It is developed by Chen et al. [9] to check if a program satisfies a set of previously defined properties known as Metamorphic Relations. It determines how input changes should affect a program's output. If the program fails to meet these expected relations, it could indicate the presence of faults. To use metamorphic testing, a suitable set of MRs should be identified, and a set of initial test cases created. Input changes defined by the MRs are then applied to develop follow-up test cases. The initial and follow-up test cases are then executed, and a fault may exist if the output does not behave according to the predicted MR. Metamorphic testing helps identify defects in programs without test oracles because it examines the input and output relationship between multiple program executions, even when the correct result of each execution is unknown [13]. For example, the SINE function $y = sin(x)$ can be tested using MT by using the property that adding 2π to the input angle does not change the output (i.e., $sin(x) = sin(x+2\pi)$). If this property is violated, it indicates a failure in the function's implementation.

C. Mutation Testing

Mutation testing is a typically used practice for evaluating the efficacy of testing strategies and adequacy of test suites [14]. This approach involves applying mutation operators to the tested program, which introduces various faults and generates a set of mutant variants. A test case is considered to "kill" a mutant if it causes the mutant to exhibit behavior different from the original program [14]. The number of killed mutants is used to calculate the mutation score (MS), which measures the thoroughness of a test suite in killing mutants [14]. The MS is calculated using the following formula:

$$MS = \frac{M_k}{M_t - M_e}$$

where the number of killed mutants is denoted as M_k, the total number of mutants is denoted as M_t, and the number of equivalent mutants is denoted as M_e (i.e., mutants that always behave the same way). Automatically generated mutants are thought to be more similar to real-life faults than manually seeded ones. Therefore, the mutation score effectively indicates the testing technique's effectiveness. In this study, the mutation analysis technique is used to assess the efficacy of our testing method.

III. Metamorphic Testing for Banking applications

A. Approach

In developing software systems for business purposes, the end-users must be confident that the application is performing as expected. However, testing every possible usage scenario can be challenging for developers. As a result, it is necessary to employ a testing technique that allows for the revision of

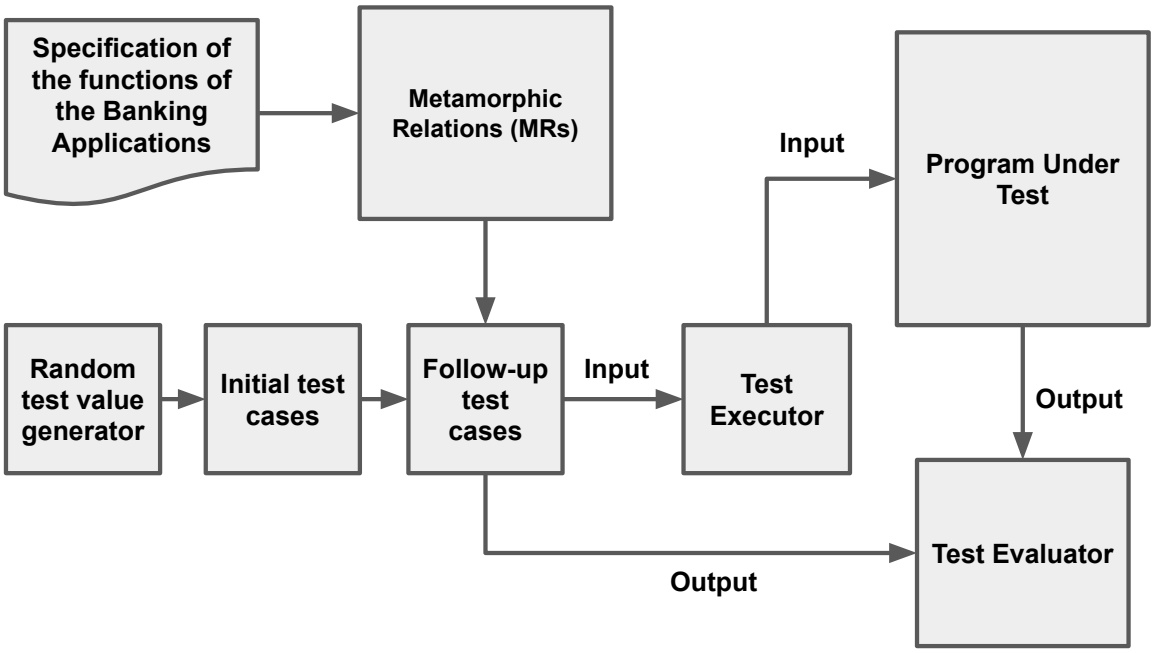

Fig. 1. A diagram that portraits the framework of MT to test a system under test (i.e. Banking Software). This framework uses MRs to generate follow-up test cases. MT process is used as the test evaluators to check the output. Here, the arrows represent the flows of information.

executed tests. MT offers an appropriate solution for testing applications without requiring oracles.

Figure 1 illustrates the MT framework for testing banking application functions. Metamorphic relationships (MRs) are a critical component of the framework since they determine the generation of test cases and the evaluation of results. When used to test banking applications, we first extract metamorphic property specifications from the function's description and identify the MRs. We then use the identified MRs to generate the follow-up test cases from the initial test cases. A test executor executes the follow-up test cases on the system under test. The outputs produced by the program being tested is compared with the follow-up test cases using a test evaluator to establish whether the MR has been satisfied or violated. If any MR is violated, we can say it has detected a fault in the system.

This section outlines the approach employed that can validate the MT framework's effectiveness in testing banking functions. The study focuses on key banking functionalities and identifies corresponding MRs. The effectiveness of MT is assessed through mutation analysis. The findings indicate that MT is capable of detecting approximately 75% of mutants (i.e., faulty programs), thereby demonstrating its efficacy as a testing technique.

B. Subject Program

This study focuses on testing the functionality of banking software by examining three commonly used features: Deposit, Withdrawal, and Transfer. The Deposit function is relatively simple and involves adding money to an existing account. On the other hand, the Withdrawal function is more complex and consists in withdrawing cash from an existing account. It only performs the withdraw function if the account has enough balance. Lastly, the Transfer function transfers money from the current account to other accounts with a commission fee associated with the transfer type. These functions are generated using the guidelines for general banking software [12] and they are all implemented in Java.

C. Metamorphic Relations

Choosing appropriate MRs is crucial when testing an application using MT. There are some guidelines that are available for determining MRs based on the program's specifications. Chen et al. [9] suggested that the MRs affecting the significant functionalities' performance are more effective. They also recommended using MRs that can produce diverse program executions. Based on these guidelines, we identified a set of MRs for each function, which are listed in Table I. Each function has some MRs derived from the function's specification. Table I contains the MR names and the relationship between initial and follow-up test inputs. Here, the initial input is (I_i) and output is (O_i). The follow-up input is (I_f) and output is (O_f). In Table I, MR1 Addition tests the Deposit function. This MR says that for the follow-up input, if we add a credit C, where the credit is greater than 0 to the initial input, i.e.,

TABLE I
METAMORPHIC RELATIONS (MRS) FOR BANKING FUNCTIONS AND THEIR ASSOCIATED DESCRIPTIONS

Metamorphic Relations (MRs)	Description
Deposit (Input is the amount to deposit, and the output is the total balance in the account.)	
MR1- Addition	This MR says that for the follow-up input (I_f), if we add a credit (C, where $C > 0$) to the initial input (I_i), i.e., $I_f = I_i + C$, the follow-up output (O_f) will increase accordingly from the initial output (O_i), i.e., $O_f > O_i$.
MR2- Subtraction	This MR says that for the follow-up input (I_f), if we subtract a credit (C, where $0 <= C <= I_i$) from the initial input (I_i), i.e., $I_f = I_i - C$, the follow-up output (O_f) will increase or remain the same accordingly from the initial output (O_i), i.e., $O_f > O_i$ or $O_f = O_i$.
MR3- Multiplication	This MR says that for the follow-up input (I_f), if we multiply a credit (C, where $C > 0$) with the initial input (I_i), i.e., $I_f = I_i * C$, the follow-up output (O_f) will increase or remain same accordingly from the initial output (O_i), i.e., $O_f > O_i$ or $O_f = O_i$.
MR4- Division	This MR says that for the follow-up input (I_f), if we divide a credit (C, where $C > 0$) by the initial input (I_i), i.e., $I_f = I_i/C$, the follow-up output (O_f) will increase or remain same accordingly from the initial output (O_i), i.e., $O_f > O_i$ or $O_f = O_i$.
MR5- Negative	This MR says that for the follow-up input (I_f), if we convert the credit to a negative value (C, where $C < 0$) from the initial input (I_i), i.e., $I_f = -(I_i)$, the follow-up output (O_f) will remain the same as the initial output and an error message will be displayed (O_i), i.e., $O_f = O_i$.
Withdrawal & Transfer (Input is the amount to withdraw or transfer, and the output is the total balance in the account.)	
MR1- Addition	This MR says that for the follow-up input (I_f), if we add a credit (C, where $0 < C < (total\_balance - I_i)$) to the initial input ($I_i$), i.e., $I_f = I_i + C$, the follow-up output (O_f) will decrease accordingly from the initial output (O_i), i.e., $O_f < O_i$.
MR2- Subtraction	This MR says that for the follow-up input (I_f), if we subtract a credit (C, where $0 <= C <= (total\_balance - I_i)$) from the initial input (I_i), i.e., $I_f = I_i - C$, the follow-up output (O_f) will decrease or remain same with error message accordingly from the initial output (O_i), i.e., $O_f < O_i$ or $O_f = O_i$.
MR3- Multiplication	This MR says that for the follow-up input (I_f), if we multiply a credit (C, where $C > 0$) with the initial input (I_i), i.e., $I_f = I_i * C$, the follow-up output (O_f) will decrease or remain same with an error message accordingly from the initial output (O_i), i.e., $O_f < O_i$ or $O_f = O_i$.
MR4- Division	This MR says that for the follow-up input (I_f), if we divide a credit (C, where $C > 0$) by the initial input (I_i), i.e., $I_f = I_i/C$, the follow-up output (O_f) will decrease or remain same accordingly from the initial output (O_i), i.e., $O_f < O_i$ or $O_f = O_i$.
MR5- Negative	This MR says that for the follow-up input (I_f), if we convert the credit to a negative value (C, where $C < 0$) to the initial input (I_i), i.e., $I_f = -(I_i)$, the follow-up output (O_f) will remain same to the initial output and an error message will be displayed (O_i), i.e., $O_f = O_i$.
MR6- Add insufficient fund	This MR says that for the follow-up input (I_f), if we add a balance greater than the total balance ($total\_balance + C$, where $C >= 0$) to the initial input (I_i), i.e., $I_f = I_i + total\_balance + C$, the follow-up output ($O_f$) will remain same to the initial output and an error message will be displayed (O_i), i.e., $O_f = O_i$.

$I_f = I_i + C$, the follow-up output will increase accordingly from the initial output, i.e., $O_f > O_i$.

D. Test Case Generation

To carry out MT, test cases are created based on the MRs. Several methods can be used to generate the initial test cases, such as generating specific test values, random test values, or iterative test values. Among these methods, random test value generation is preferred for MT as it is cost-efficient and unbiased [10]. Hence, this study used random test value generation to produce the source test cases. Follow-up test cases are then developed using the MRs described in Table I.

E. Evaluation

In order to assess the efficacy of MT, we utilize mutation analysis. We create Junit test cases for the functions based on the MRs. Then we use mutation operators to introduce faults into the implementation of the functions automatically using the PIT mutation testing tool [15]. This resulted in 16 mutants. Usually, equivalent mutants are excluded from experiments. However, this mutation testing tool does not create equivalent mutants. Test suites are generated using the MRs outlined in Table I, which is then used to test the subject programs. We use the mutation score (MS) metric to measure the effectiveness of MT. When a mutant is killed, it is considered a detected fault.

IV. RESULTS

In this section, we present the results of our experiment, where we utilized mutation analysis to assess the efficacy of our MT framework. We used the PIT mutation tool to generate the mutators listed in Table II. In Table III, we summarize the test efficiency of MT mainly by measuring its Mutation Score (MS) and Test Strength. The following describes the information listed in Table III.

- Line coverage shows the percentage of lines covered by the tests.
- Mutation coverage shows how many mutants are killed from the total mutants.
- The MS measures the percentage of mutants killed out of all mutants (i.e., excluding the equivalent mutants) created without test coverage [15].
- Test Strength measures the ratio of mutants killed out of all mutants with test coverage [15]. The Test Strength metric does not include mutants that survive due to a lack of coverage [15]. This metric is considered a better metric than MS when validating builds. It is also shown as a percentage.

TABLE II
ACTIVE MUTATORS USED FOR MUTATION ANLYSIS

Active Mutators
BOOLEAN_FALSE_RETURN
BOOLEAN_TRUE_RETURN
CONDITIONALS_BOUNDARY_MUTATOR
EMPTY_RETURN_VALUES
INCREMENTS_MUTATOR
INVERT_NEGS_MUTATOR
MATH_MUTATOR
NEGATE_CONDITIONALS_MUTATOR
NULL_RETURN_VALUES
PRIMITIVE_RETURN_VALS_MUTATOR
VOID_METHOD_CALL_MUTATOR

TABLE III
RESULTS OF MUTATION ANALYSIS BASED ON MUTATION SCORE AND TEST STRENGTH

MRs	Line Coverage (%)	Mutation Coverage (Killed mutant/Used mutant)	Mutation Score (MS) (%)	Test Strength (Killed mutant/Used mutant)	Test Strength (%)
Deposit					
MR1	93%	3/6	50%	3/6	50%
MR2	93%	3/6	50%	3/6	50%
MR3	93%	3/6	50%	3/6	50%
MR4	93%	3/6	50%	3/6	50%
MR5	64%	2/6	33%	2/5	40%
All	100%	4/6	67%	4/6	67%
Withdrawal					
MR1	82%	4/9	44%	4/9	44%
MR2	82%	4/9	44%	4/9	44%
MR3	82%	4/9	44%	4/9	44%
MR4	82%	4/9	44%	4/9	44%
MR5	53%	2/9	22%	2/5	40%
MR6	71%	3/9	33%	3/8	38%
All	100%	6/9	67%	6/9	67%
Transfer					
MR1	82%	4/9	44%	4/9	50%
MR2	82%	4/9	44%	4/9	50%
MR3	82%	4/9	44%	4/9	50%
MR4	82%	4/9	44%	4/9	50%
MR5	53%	2/9	22%	2/5	40%
MR6	71%	3/9	33%	3/8	38%
All	100%	6/9	67%	6/9	67%
Total	100%	12/16	75%	12/16	75%

For each function, the overall performance of MT is shown. Moreover, the last row of the table presents the overall performance of MT when we consider the testing results of all functions and all MRs together.

The effectiveness of each MR can be compared using their MS and test strength for each function. Among all the MRs, for the deposit function, MR1, MR2, MR3, and MR4 are more effective than MR5. For example, MR1, MR2, MR3, and MR4 have an MS of 50% with 93% line coverage, while MR5 has an MS of 33% with 64% line coverage. When all 5 MRs are combined, the MS increases to 67% with 100% line coverage. However, when test strength is used, the effectiveness of MR5 increases to 40%, as mutants that survive due to lack of coverage are not counted.

With the withdrawal function and considering all MRs, MR1, MR2, MR3, and MR4 are more effective, while MR5 and MR6 are less effective. This can be seen in the mutation score (MS) of each MR, where MR1, MR2, MR3, and MR4 have an MS of 44% with 82% line coverage, while MR5 has an MS of 22% with 53% line coverage, and MR6 has an MS of 33% with 71% line coverage. When all six MRs are considered, the MS increases to 67% with 100% line coverage. However, when using the test strength metric, it can be seen that MR5 becomes more effective with a score of 40% and MR6 with a score of 38% since the mutants that survive due to lack of coverage are not counted.

Regarding the transfer function and all the generated MRs, MR1, MR2, MR3, and MR4 are more effective than MR5 and MR6. Specifically, MR1, MR2, MR3, and MR4 have a higher MS of 44% with 82% line coverage compared to MR5's MS of 22% with 53% line coverage. When all 6 MRs are taken into account, the MS increases to 67% with 100% line coverage.

However, by using test strength, it becomes apparent that the effectiveness of MR5 increases to 40% and MR6 to 38% since the mutants that survive due to lack of coverage are not taken into consideration.

Finally, if all 17 MRs are utilized collectively, they can eliminate up to 75% of all mutants, and the combined value of both MS and test strength is greater than that of any individual MR. This implies that, as long as there are no concerns about testing expenses, many MRs as possible should be employed to create test suites.

V. Threats to validity

Possible risks exist when validating the examined MT framework in this study. Two types of threats are discussed: external and internal threats [16]. One of the significant external threats to validation is the ability to generalize the study results to other cases. The study utilized a demo banking application with mathematical functions that perform standard calculations and a relatively small data set. Therefore, the results may not explicitly confirm that this approach will work for industrial-sized software.

The PIT mutation testing tool is used to produce mutation analysis for the experiments. The method utilizes a third-party tool, which may contain potential faults which could threaten the proposed method. This type of threat could occur concerning internal validity.

VI. Related Work

When an oracle is absent, the effectiveness of testing techniques is limited. The use of metamorphic testing (MT) in this paper is effective in conducting testing without requiring an oracle. MT has been utilized to address the Oracle problem in fault-based testing and symbolic execution. Several studies have examined the oracle problem in various fields. However, there needs to be research on the functional testing of banking software using MT. In one case study, Chen et al. [17] injected a seeded fault into a program that implemented partial differential equations. They compared the efficacy of special test cases versus MT in catching defects. While special test cases missed seeing the fault, MT could identify it by employing only one metamorphic relation. Aruna and Prasad [18] suggested multiple MRs for multi-precision arithmetic software, which is evaluated with four mathematical projects and mutation testing. The banking functions share similarities with these mathematical processes. Therefore, we attempted to assess these functions using MT.

VII. Conclusion and Future Work

This study confirms that the MT framework is suitable for evaluating banking functions and that MT is an efficient and effective testing technique even without an oracle. MT can test a banking application's deposit, withdrawal, and transfer functions without needing oracles, which is a significant advantage for testing complex banking functions. The experimental results demonstrate that MT can detect up to 75% of mutants, indicating its high fault detection capability.

For future work, we plan to extend the testing of the banking software beyond the function level and assess the security aspects of the banking application using MT. We also aim to improve the automation capability of MT by predicting MRs based on program execution flow. This will help the developers so they don't have to define the MRs manually. Additionally, we intend to conduct further empirical studies to evaluate the effectiveness of MT for banking software with more complicated functions and deploy the framework to test various financial and payment applications.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507-525, 1 May 2015, doi: 10.1109/TSE.2014.2372785.

[2] M. M. Lehman and L. A. Belady. 1985. Program evolution: processes of software change. Academic Press Professional, Inc., USA.

[3] Type, P-type, S-type systems. E. (n.d.). Retrieved May 2, 2023, from https://denrox.com/post/e-type-p-type-s-type-systems.

[4] F. Molu, "Software testing practices in critical financial systems transformation," 2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), Konya, Turkey, 2013, pp. 394-399, doi: 10.1109/TAEECE.2013.6557307

[5] X. Xie, Z. Yang, J. Yu and W. Zhang, "Design and implementation of bank financial business automation testing framework based on QTP," 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), Changchun, China, 2016, pp. 143-147, doi: 10.1109/ICCSNT.2016.8070136.

[6] L. Padgham, Z. Zhang, J. Thangarajah and T. Miller, "Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems," in IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1230-1244, Sept. 2013, doi: 10.1109/TSE.2013.10.

[7] E. J. Weyuker, "On testing non-testable programs", Comput. J., vol. 25, no. 4, pp. 465-470, 1982.

[8] H. Liu, F.-C. Kuo, D. Towey and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?", IEEE Trans. Softw. Eng., vol. 40, no. 1, pp. 4-22, Jan. 2014.

[9] T. Y. Chen, S. C. Cheung and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases", 1998.

[10] T.Y. Chen, F.C. Kuo, Y. Liu, A. Tang, "Metamorphic testing and testing with special values", Proceedings of SNPD2004, 2004, pp128-134.

[11] T. Y. Chen, F.-C. Kuo, T. H. Tse and Z. Q. Zhou, "Metamorphic testing and beyond", Proc. 11th Annu. Int. Workshop Softw. Technol. Eng. Practice, pp. 94-100, Sep. 2003.

[12] C. A. Gumussoy. 2016. Usability guideline for banking software design. Comput. Hum. Behav. 62, C (September 2016), 277–285. https://doi.org/10.1016/j.chb.2016.04.001

[13] S. Segura, G. Fraser, A. B. Sanchez and A. Ruiz-Cortés, "A Survey on Metamorphic Testing," in IEEE Transactions on Software Engineering, vol. 42, no. 9, pp. 805-824, 1 Sept. 2016, doi: 10.1109/TSE.2016.2532875.

[14] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011, doi: 10.1109/TSE.2010.62.

[15] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. 2016. "PIT: a practical mutation testing tool for Java (demo)". In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[16] X. Zhou, Y. Jin, H. Zhang, S. Li and X. Huang, "A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering," 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, New Zealand, 2016, pp. 153-160, doi: 10.1109/APSEC.2016.031.

[17] T. Y. Chen, J. Feng and T. H. Tse, "Metamorphic testing of programs on partial differential equations: A case study", Proc. 26th Int. Comput. Softw. Appl. Conf. Prolonging Softw. Life: Develop. Redevelop., pp. 327-333, 2002.

[18] C. Aruna and R. S. R. Prasad, "Metamorphic relations to improve the test accuracy of multi precision arithmetic software applications", Proc. Int. Conf. Adv. Comput. Commun. Informat., pp. 2244-2248, Sep. 2014.