Sena TLS-Parser: A Software Testing Tool for Generating Test Cases

Rosziati Ibrahim[1]
Department of Software Engineering
Universiti Tun Hussein Onn Malaysia
Parit Raja, Malaysia

Sapiee Jamel[3]
Department of Information Security
Universiti Tun Hussein Onn Malaysia
Parit Raja, Malaysia

Samah W.G. AbuSalim[2]
Department of Computer Information Sciences
Universiti Teknologi PETRONAS (UTP)
Perak, Malaysia

Jahari Abdul Wahab[4]
Engineering R&D Department
SENA Traffic Systems Sdn. Bhd.
Kuala Lumpur, Malaysia

***Abstract*—Currently, software complexity and size has been steadily growing, while the variety of testing has also been increased as well. The quality of software testing must be improved to meet deadlines and reduce development testing costs. Testing software manually is time consuming, while automation saves time and money as well as increasing test coverage and accuracy. Over the last several years, many approaches to automate test case creation have been proposed. Model-based testing (MBT) is a test design technique that supports the automation of software testing processes by generating test artefacts based on a system model that represents the system under test's (SUT) behavioral aspects. The optimization technique for automatically generating test cases using Sena TLS-Parser is discussed in this paper. Sena TLS-Parser is developed as a Plug-in Tool to generate test cases automatically and reduce the time spent manually creating test cases. The process of generating test cases automatically by Sena TLS-Parser is be presented through several case studies. Experimental results on six publicly available java applications show that the proposed framework for Sena TLS-Parser outperforms other automated test case generation frameworks. Sena TLS-Parser has been shown to solve the problem of software testers manually creating test cases, while able to complete optimization in a shorter period of time.**

***Keywords*—Software testing; schema parser; software under test (SUT); model based testing (MBT); java applications**

I. Introduction

Before a software can be released to consumers, it needs to pass the software testing phase. Software testing covers the aspect of testing the software to meet its functional requirements as well as discovering errors before the software is released. Two main factors are usually used to determine whether tests will show failures: test inputs and test oracles [1]. A statement in JUnit test is an example of a test oracle. Software testing is important not only for the software company, but also for consumers. Many consumers are currently worried about how software companies ensure software quality, the mechanisms used to do so, and so on. Although the types, frequency and activities of tests vary from program to program, most of the common activities used in each test cycle are: requirements testing, test planning, writing test cases, test execution, testing feedback and defect testing.

The development of test cases is a difficult aspect of software testing [2]. Creating test cases manually is time consuming. Creating test cases manually should address the aspects of the test objective. Therefore, creating test cases automatically is more efficient and consumes less time. The techniques for automated test case generation aim to efficiently identify a limited number of cases that satisfy an adequacy criterion, reducing the cost and resulting in more effective software product testing. One of the well-known techniques for software testing is Model-based testing (MBT). MBT is a testing technique that creates test cases automatically from models derived from existing application artifacts [3]. MBT is a promising approach for automatic testing to increase testing performance and effectiveness [4]. MBT can perform and complete test tasks in a more cost-effective and reliable manner than conventional test methods. A description of the MBT method is presented in [5]. This paper addresses the problem of manually creating test cases that consume more time. By introducing Sena TLSParser, test cases can be automatically created and generated. Sena TLS Parser can reduce time in generating test cases manually.

The next section will discuss related works followed by details of the proposed Sena TLS-Parser Framework. Subsequently, the implementation of Sena TLS-Parser is discussed followed by the comparison of the proposed framework with other frameworks.

II. Related Work

With the development of model-based engineering technology [6], MBT has attracted more and more interest in research. In the past few years, several MBT tools have been developed to support MBT activities [7]. Li et al. [4] proposed and applied a set of test case generation criteria, as well as surveying new methods that have not been used in previous research or have not been analyzed using test case generation criteria. From 2000 to 2018, a review study on requirement-based test case generation was presented in [8]. The study was

conducted to gain information in the areas of requirements-based test case generation and future studies. In addition, authors in [9] present a systematic mapping study (SMS) by analyzing 87 studies in this field. They discovered that the majority of the studies were devoted to test generation activities. Utting et al. [10] presented model-based research literature over the last ten years including MBT methodology and the industry's current level of MBT adoption. The Unified Modeling Language (UML) is a diagrammatical modeling language that enables developers to identify, visualize, create, analyze and document system features. It is the most popular and widely accepted language in the software industry, to the point that its spread and popularity may have reached a point where it is impossible to imagine a software working without it. There are many UML diagrams used for this purpose such as class diagram, use case diagram, activity diagram and others. One of these diagrams, the UML activity diagram is used to describe behavior while modeling the sequence of activities in the system. It is closely related to use case diagram, which shows the sequence of steps the system performs in order to carry out a use case. By using it, any software process is simplified and improved by identifying complex use cases. Therefore, many research works concentrate on generating test cases from UML [11 - 16].

Many researchers have created a variety of tools for test case generation, but the features of these tools vary greatly. This makes it difficult for the user to identify the right tool for the testing process. TCG, an open-source LoTuS modelling tool plugin was developed by Muniz et al. [17] to generate test cases. EvoSuite [18, 19] is a tool that generates test suites automatically for Java programs with high code coverage and assertions. EvoSuite employs a number of innovative techniques that result in increased structural coverage and efficient assertion selection based on seeded defects, both of which are important features that other Java tools are lacking. Another tool called EPiT was developed by Ibrahim et al. [20], which is shown to be effective in generating test cases automatically. With the increase in Android mobile devices, there is growing interest in automated testing for Android applications. GUI testing is one of the most used techniques for detecting errors in mobile applications and for testing app functionality and usability. Salihu et al. [21] proposed AMOGA which is an alternative model-based testing approach for mobile apps. Their proposed method uses a combination of the UI element's event list and each event to dynamically exercise event ordering at run time. Another tool called APE is presented by Gu et al. [22] for Android apps testing. PLATOOL [23] is another tool that has been proven effective in creating useful functional tests to deal with events involved in mobile applications during the automatic testing phase. More details about various software testing techniques applied for testing mobile applications are shown in [24].

The Synchronized Depth First Search (SDFS) introduced by Pinkal and Niggemann [25] to automate test case generation is proven to efficiently execute testing with less effort and time compared to other techniques. Based on research, it is possible to generate test cases automatically using Timed Synchonizable I/O Automation. Genetic algorithms have been used successfully in software testing. Mishra et al. [26, 27] shows how genetic algorithms are used for software testing in generating random test cases. Du et al. [28] presents a combination of genetic algorithms with mutation testing to increase coverage and mutation score within test cases. To assess output in terms of generating test cases, the proposed algorithm by Wang and Liu [29] shows that it is capable of achieving both high performance and low time cost in the automated generation of software test cases.

One significant approach is the generation of test cases from UML models. Shin and Lim [30] propose an approach in reducing time and resources required for testing embedded software. Ma and Provost [31] suggest a testing process that ensures that a system's nominal behavior is fully covered while still allowing for the consideration of defective behavior. Elqortobi et al. [32] describe the components of an automated Modified Condition/Decision Coverage MC/DC Test Generation Tool (TGT) for avionics software test sequence generation. Their method incorporates three coverage parameters to increase the performance and error detection capacity of the derived tests. The criteria are selected to satisfy the industrial needs for avionics software certification.

III. SENA TLS-PARSER

Sena TLS-Parser consists of four main steps. Fig. 1 shows the flowchart for Sena TLS-Parser.

Based on Fig. 1, Sena TLS-Parser can be used to generate test cases automatically in Eclipse Integrated Development Environment (IDE). The source code is the input for Sena TLS-Parser. The schema parser will read the source codes line by line. The token will be used to detect classes and methods using code smell. The algorithm for code smell is discussed in [33]. MBT is used for generating the test cases. The algorithm for generating the test cases is discussed in [20]. However, for the time being, Sena TLS-Parser can only generate test cases for Java applications only. The output for Sena TLS-Parser is the generated test cases as shown in Fig. 2.

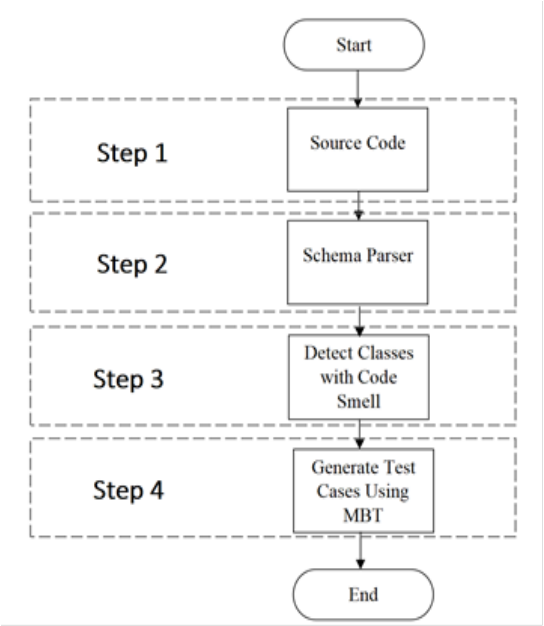

Fig. 1. The Flowchart for Sena TLS-Parser.

```
Console SenaTLSParser
Start Time: 2022/06/22 14:24:58
### Working in project linux-core-service-5 ###

Package

Package CVS

Package com

Package com.CVS

Package com.senatraffic

-----------------------------------------------------
Source file Constants.java
Has number of lines: 154

-----------------------------------------------------
Source file ItrafficService.java
Has number of lines: 63

Method name :main
Signature :([QString;)V
Return Type :V
Input variable :
args
Generate Test Cases:
Test Case 1 : valid [args]are input with :1
Test Case 2 : invalid [args]are input with :-1
Test Case 3 : null [args]are input with :null
```

Fig. 2. Test Cases Generated from Sena TLS-Parser.

IV. Implementation of Sena TLS-Parser

The tool is executed in four steps and this section discusses the steps. The first step is importing the java project to an Eclipse environment as a Plug-in Tool. Fig. 3 shows an example of how Sena TLS-Parser is imported as a Plug-in Tool. By clicking the in-help tab menu, Sena TLS-Parser can be installed as a new software option in Eclipse environment. Sena TLS-Parser will then analyze the codes by right clicking the project and selecting the SenaTLSParser-2.0 as shown in Fig. 4.

Based on Table II, Sena TLS-Parser successfully generated test cases automatically for each application. The duration also depends on the line of code (LOC) for the application. If the LOC is smaller, less time is taken to generate the test cases; for example the Calculator application. If, however, the LOC is much bigger, more time is required to generate the test cases. However, the generation of the test cases also depends on the complexity of the algorithm and the number of methods the application has; for example the Elevator application.

Fig. 3. Installation of Sena TLS-Parser as a Plug-in Tool.

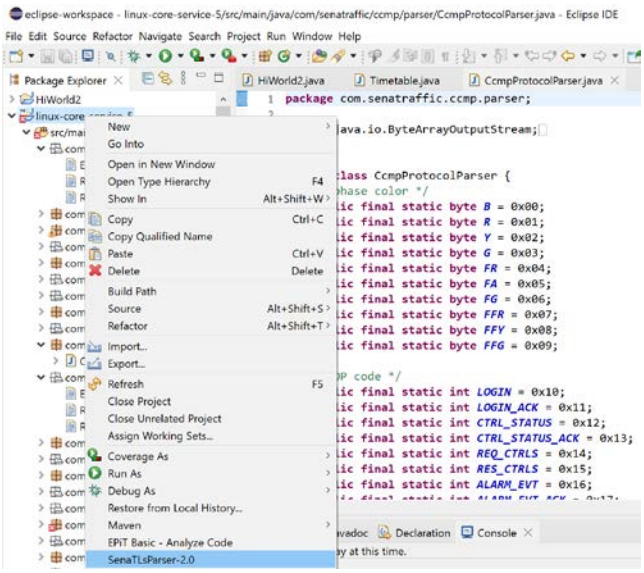

Fig. 4. Sena TLS-Parser as a Plug-in Tool.

Fig. 4 shows the second step for selecting Sena TLS-Parser to analyze source code. The third step is for Sena TLS-Parser to begin analyzing and detecting all of the classes in the source code. The class will be detected by the parser node by node. Sena TLS-Parser then identifies a method within each node. Sena TLS-Parser will save all detected method classes in a variable. Finally, for the fourth step, test cases are generated based on the identified attributes. When Sena TLS-Parser has finished analyzing the source codes, a success popup menu will appear. Fig. 5 shows the popup menu appearing, which indicates the time used to analyze the source code.

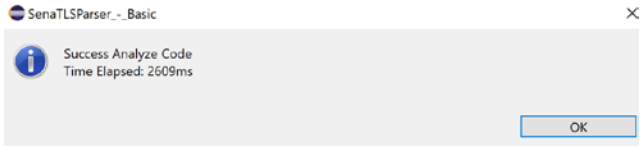

Fig. 5. Time Taken to Generate Test Cases from Sena TLS-Parser.

After the code analysis process is complete, the results will be shown in the Sena TLS-Parser console. The console contains all project information such as project name, test cases created for each class, and the time and date of when the test was conducted. Fig. 2 shows the Sena TLS-Parser console.

V. Results and Discussion

Case studies are used for testing the performance of Sena TLS-Parser. The case studies are completed by testing Java applications only by developing test cases for each case study to evaluate performance in execution time. For the case studies, performance evaluation regarding Java applications is tested using Eclipse IDE Version 4.5. The six case studies were selected and downloaded from GitHub website [34]. The six case studies are Calculator [35], ATM Machine [36], BlackJack [37], Traffic Light Simulation [38], Airline Reservation [39] and Elevator [40]. Table I describes the case studies used in this paper by providing a summary of the number of classes for each application and a description for each case study.

TABLE I. THE CASE STUDIES

Application	Number classes	Description
Calculator [35]	1	The application contains mathematical methods like add, subtract, multiply, divide methods.
ATM Machine [36]	5	The application has financial transactions such as cash deposits, withdrawals and transfer funds.
BlackJack [37]	5	Simple Blackjack implemented in Java. Creates a random deck of cards, or takes in a file with a list of cards, and plays a round. Prints the winner and the resulting hands of the player and the dealer.
Traffic Light Simulation [38]	5	A traffic light has three lenses, green, orange and red mounted in a panel. Simulates a set of traffic lights (N, S), (E, W) at an intersection.
Airline Reservation [39]	7	Provide online ticket and seat booking for national and international flights as well as flight departure information.
Elevator [40]	9	A small Java project to simulate the evolution of an elevator.

The source codes from all applications are uploaded in Eclipse environment. Subsequently, Sena TLSParser is used as a Plug-in Tool to generate test cases for each application listed in Table I. The results for each case study are shown in Table II. Sena TLSParser shows the duration taken to analyze the source code and generate test cases based on the number of classes detected in the LOC for each application. The results show that the Calculator application is the fastest with 1ms. Meanwhile, the slowest is the ATM Machine application, which is 89ms. The comparison is based on the time required to generate test cases using Sena TLS-Parser.

Based on Table II, Sena TLS-Parser successfully generated test cases automatically for each application. The duration taken also depends on the LOC for each application. If the LOC is smaller, less time is required to generate the test cases; for example the Calculator application. If, however, the LOC is much bigger, more time is required to generate the test cases. However, the generation of the test cases will also depend on the complexity of the algorithm and the number of methods in the application; for example the Elevator application.

TABLE II. THE RESULTS USING SENA TLS-PARSER

APPLICATION	Number classes	LOC	Duration Time (ms)
Calculator	1	63	1
ATM Machine	5	635	89
BlackJack	5	458	51
Traffic Light Simulation	5	185	46
Airline Reservation	7	154	15
Elevator	9	1150	75

Testing the software manually requires effort and is time consuming. Automation saves time and money while also increasing test coverage and accuracy, which is beneficial to both developers and testers. Choosing the right automation framework is critical to assist with various types of testing such as unit, functional, and regression testing. For comparison purposes, three automated testing frameworks are reviewed compared to the proposed framework, Sena TLS-Parser. These frameworks are JUnit [41], TestNG [42] and Epit [20], which are widely used in the generation of test cases.

A. Junit Testing Framework

JUnit [41] is a well-known Java unit testing framework. It is easy to understand, simple to integrate, and best of all, it is open-sourced. For writing test cases, JUnit employs annotations and assertions. It includes a test-runner for identifying and running all test methods in a project. The JUnit process is done through setting fixed states for objects and running tests by using Fixtures, Test suites, Test runners and JUnit classes are the main features offered by JUnit. The work of these Fixtures aims to provide a good environment for the conduct and implementation of the test. Test suites are a collection of unit test cases that are compiled together. Before testing a code, annotations are used in order to run the test suite. Test runners are used to carry out test cases while JUnit classes are used for testing and writing JUnits, with assert, test case, and test result.

B. TestNG Framework

Cédric Beust created TestNG [42], an open-source test automation framework inspired by JUnit and NUnit for the Java language. The goal of TestNG's design is to provide more powerful and easy-to-use functionalities for a broad range of test categories such as unit, functional, end-to-end, integration, and so on. TestNG's advanced and useful features make it a more robust framework than its competitors. In this framework, the executing of the methods is determined by a set of codes called annotations. Using these annotations demonstrates the usage of Java language new features in a real-world production environment.

C. EPiT Plug-in

EPiT [20] was created to reduce the time spent manually generating test cases by utilizing code smell technique for automated test case generation. EPiT begins by reading the code line by line before applying code smell technique to detect all classes in the Java application. Following that the tool determines the method's name, input parameter, and return type and stores them in variables for use in generating test cases. EPiT has demonstrated its ability to optimize automated test case generation using the code smell technique in a short period of time and with high efficiency.

D. Comparisons of Results

Each framework is used to test the generation of test cases for the case studies. Table III presents the result comparison of the test cases generation framework for each case study based on execution time.

A JUnit test is a method in a class that is only used for testing. This is known as a Test class. To indicate that a method is a test method, @Test annotation is used. This function executes the code being tested. An assert method is used which is provided by Junit.

TestNG covers all categories of tests such as unit, functional, and integration testing. In this research, TestNG has

been integrated with eclipse to generate the test report and execute multiple test cases in parallel. TestNG uses various Annotations for test cases generation, such as @BeforeSuite, @AfterSuite, @BeforeTest, and @AfterTest. Annotations in TestNG are lines of code that can control how the method below them will be executed. Annotations are preceded by "@" symbol.

TABLE III. THE RESULTS FOR DIFFERENT FRAMEWORKS

Application	JUnit	TestNG	EPiT	Sena TLS-Parser
Calculator	32	9	8	1
Airline Reservation	96	57	59	15
Traffic Light Simulation	4201	4101	91	46
BlackJack	93	60	111	51
ATM Machine	179	76	94	89
Elevator	339	67	95	75

The comparison between the testing frameworks has been done based on duration required to generate test cases for each application. The results demonstrate the effectiveness of the proposed tool (Sena TLS-Parser) in automatically and quickly producing test cases. Oshin et al. [43] compared JUnit framework with TestNG framework. Based on Table III, TestNG gives better results as compared with JUnit. The results shown are consistent with the comparison done by [43]. The results show that the Calculator application is the fastest with 32ms for JUnit, 9ms for TestNG, 8ms for EPiT and 1ms using Sena TLS-Parser. For Calculator application, Sena TLS-Parser gives the best result compared to JUnit, TestNG and EPiT. Meanwhile, for the Traffic Light simulation, both JUnit and TestNG have more execution time as compared to EPiT and Sena TLS-Parser. Sena TLS-Parser is an automation testing tool dedicated for Sena Traffic Light System (TLS) with MBT embedded in its algorithm. Therefore, the results show it has better performance. Meanwhile, EPiT with its technique for code smell [33] improved the generation of test cases compared to the JUnit and TestNG frameworks.

For more clarification, Fig. 6 shows the graph for the applications and duration time for each of the frameworks. It is noticeable that the time taken by each framework to generate test cases depends on the complexity of the algorithm and the number of methods of the application. For example, the traffic light simulation application has more complexity for the algorithm as compared with the Elevator application. Therefore, it takes a lot of time to generate test cases using JUnit and TestNG frameworks. Comparing with EPiT and Sena TLS-Parser, traffic light simulation application takes less time than Elevator application in generating test cases. In addition, the line of codes for the ATM Machine application is less than the Elevator application, however it takes more time using EPiT framework when applying code smell algorithm for the purpose of reducing the redundancy of test cases generation.

To summarize, there are numerous factors that contribute to inconsistent results, including project code complexity, CPU usage, and memory usage. Despite inconsistencies in the results, the results of the case studies demonstrated that Sena TLS-Parser is faster than conventional manually generated test cases and other testing frameworks.

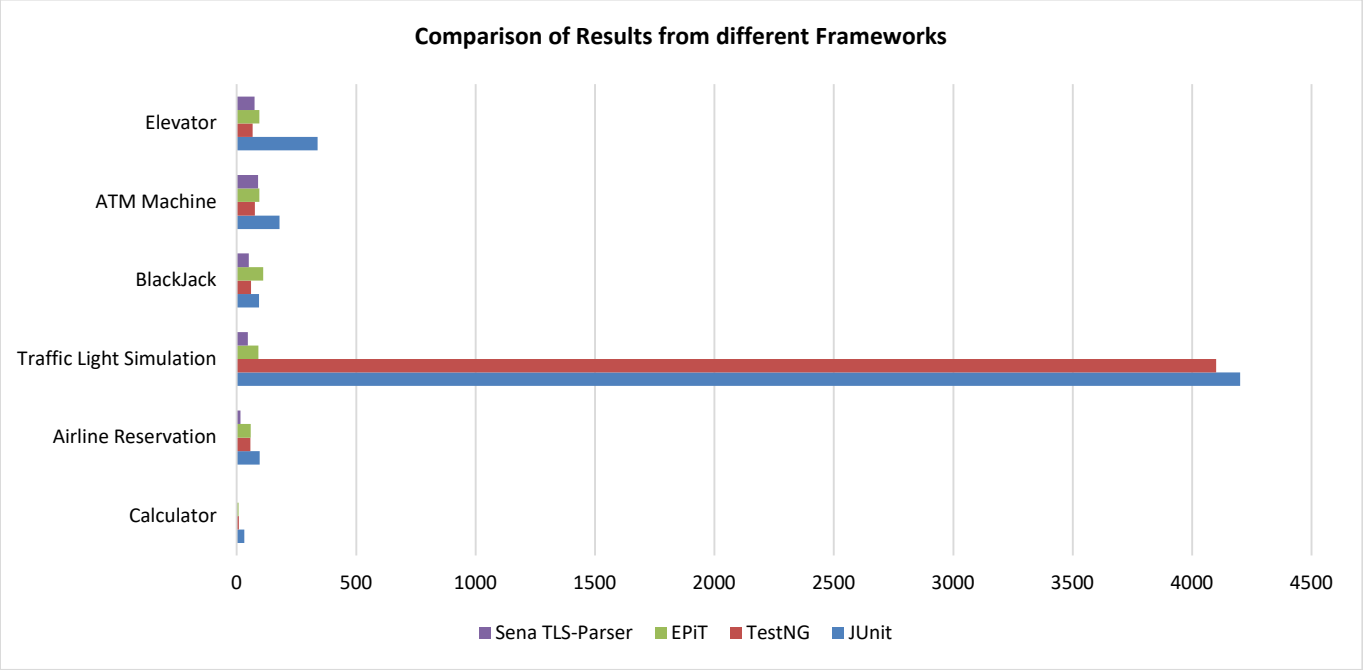

Fig. 6. Comparison of Results from different Frameworks.

VI. Conclusion and Future Work

During the software testing process, automatic test data generation is critical. Unit-level testing is more successful because test cases cover all the essential paths of the software being tested. The process of creating test cases automatically by Sena TLS-Parser in an Eclipse setting was discussed in this paper. Based on the results presented in this paper, it is shown that Sena TLS-Parser has successfully generated test cases automatically and has a faster response time than traditional manual testing as well as JUnit testing and TestNG. Also, using the MBT technique to create test cases is a very powerful way to do so. For future work related to this research, Sena TLS-Parser framework can be extended to work with other programming languages such as C and C++. Adding other features would also be an interesting direction for future work using Sena TLS-Parser. Sena TLS-Parser can also be generalized to cover mobile applications for software testing. A convertor can be used to convert the source codes of mobile application in ".apk" format into source codes of java programming in ".java" format. By having the convertor, Sena TLSParser will also be able to generate test cases for mobile applications.

Acknowledgment

The authors would like to thank Universiti Tun Hussein Onn Malaysia (UTHM) for supporting this research. The authors received funding for this study from Industry Grant under Grant Vote No M081.

References

[1] N. Li and J. Offutt, "Test Oracle Strategies for Model-Based Testing," *IEEE Transactions on Software Engineering*, 43(4), 372–395. doi:10.1109/tse.2016.2597136, 2017.

[2] M. Keyvanpour, H. Homayouni, and H. Shirazee, "Automatic Software Test Case Generation: An Analytical Classification Framework," *International Journal of Software Engineering and Its Applications* Vol. 6, No. 4, October, 2012.

[3] G. De Cleva Farto and A. T. Endo, "Evaluating the Model-Based Testing Approach in the Context of Mobile Applications," *Electronic Notes in Theoretical Computer Science*, 314, 3–21. doi:10.1016/j.entcs.2015.05.002, 2015.

[4] W. Li, F. Le Gall and N. Spaseski, "A Survey on Model-Based Testing Tools for Test Case Generation," *In: Itsykson V., Scedrov A., Zakharov V. (eds) Tools and Methods of Program Analysis.* TMPA 2017. Communications in Computer and Information Science, vol 779. Springer, Cham. https://doi.org/10.1007/978-3-319-71734-0_7.

[5] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux and A. Vernotte, "Recent Advances in Model-Based Testing," *Advances in Computers*, 53–120, January 2016. doi:10.1016/bs.adcom.2015.11.004.

[6] Model-Based Engineering Forum. [Online]. Available: http://modelbasedengineering.com/.

[7] R. Marinescu, C. Seceleanu, H. Le Guen, P. Pettersson, "A research overview of tool supported model-based testing of requirements-based designs," *In: Advances in Computers*, pp. 89–140. Elsevier (2015).

[8] A. Mustafa, W. M. Wan-Kadir, N. Ibrahim, M. A. Shah, M. Younas et al., "Automated test case generation from requirements: a systematic literature review," *Computers, Materials & Continua*, vol. 67, no.2, pp. 1819–1833, 2021.

[9] M. Bernardino, E. M. Rodrigues, A. F. Zorzo and L. Marchezan, "Systematic mapping study on MBT: tools and models," *IET Software*, 11(4), 141–155. doi:10.1049/iet-sen.2015.0154, 2017.

[10] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, "Chapter two - recent advances in model-based testing," *ser. Advances in Computers, A. Memon, Ed. Elsevier*, 2016, vol. 101, pp. 53 – 120.

[11] F. G. C. Ribeiro, C. E. Pereira, A. Rettberg, M. S. Soares, "Model-based requirements specification of real-time systems with UML, SysML, and MARTE," *Software & Systems Modeling*, vol. 17, no. 1, pp. 343-361, 2018.

[12] M. L. van Eck, N. Sidorova, W. M. van der Aalst, "Guided interaction exploration and performance analysis in artifact-centric process models," *IEEE 19th Conference on Business & Information Systems Engineering*, pp. 1-5, 2018.

[13] N. Khurana, R. S. Chhillar, U. A. Chhillar, "A novel technique for generation and optimization of test cases using use case, sequence, activity diagram and genetic algorithm," *Journal of Software*, vol. 11, no. 3, pp. 242-250, 2016.

[14] T. A. Alrawashed, A. Almomani, A. Althunibat, A.Tamimi, "An Automated Approach to Generate Test Cases from Use Case Description Model," *CMES-Computer Modeling in Engineering & Sciences*, 119(3), 409–425, 2019.

[15] Meiliana, I. Septian, R. Alianto, Daniel, & F. Gaol, "Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm," *ICCSCI*, 2017.

[16] M., I. Septian, R. S. Alianto, D. and F. L. Gaol, "Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm," *Procedia Computer Science*, 116, 629–637. doi:10.1016/j.procs.2017.10.029.

[17] L.L. Muniz, U.C. Nett and P. Maia, "TCG - A Model-based Testing Tool for Functional and Statistical Testing," *ICEIS*, 2015.

[18] J. Campos, A. Panichella and G. Fraser, "EvoSuite at the SBST 2019 Tool Competition," *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, Montreal, QC, Canada, 2019, pp. 29-32, doi: 10.1109/SBST.2019.00017.

[19] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," *In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011.

[20] R. Ibrahim, AAB. Amin, S. Jamel, J. Abdul Wahab, "EPiT: A Software Testing Tool for Generation of Test Cases Automatically," *International Journal of Engineering Trends and Technology, 68(7),8-12, 2020.*

[21] I.A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli and A. Usman, "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing," *IEEE Access*, 1–1. doi:10.1109/access.2019.2895504, 2019.

[22] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, … Z. Su, "Practical GUI Testing of Android Applications Via Model Abstraction and Refinement," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. doi:10.1109/icse.2019.00042.

[23] E. H. Marinho and E. Figueiredo, "PLATOOL: A Functional Test Generation Tool for Mobile Applications," *In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20).* Association for Computing Machinery, New York, NY, USA, 548–553. DOI:https://doi.org/10.1145/3422392.3422508, 2020.

[24] S.W.G. AbuSalim, R. Ibrahim, J. Abdul Wahab, "Comparative Analysis of Software Testing Techniques for Mobile Applications," In: *Phys.: Conf.* Ser. 1793 012036, 2020.

[25] K. Pinkal and O. Niggemann, "A new approach to model-based test case generation for industrial automation systems," *2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Emden*, 2017, pp. 53-58, doi: 10.1109/INDIN.2017.8104746.

[26] D. Mishra, S. Bilgaiyan, R. Mishra, A. A. Acharya, … S. Mishra, "A Review of Random Test Case Generation using Genetic Algorithm," *Indian Journal of Science and Technology*, 10(30), 1–7. doi:10.17485/ijst/2017/v10i30/107654, 2017.

[27] D. Mishra, R. Mishra, K. Das and A. Acharya, "Test Case Generation and Optimization for Critical Path Testing Using Genetic Algorithm," *SocProS*, 2017.

[28] Y. Du, Y. Pan, H. Ao, N. Ottinah Alexander and Y. Fan, "Automatic Test Case Generation and Optimization Based on Mutation Testing," *2019 IEEE 19th International Conference on Software Quality,*

Reliability and Security Companion (QRS-C). doi:10.1109/qrs-c.2019.00105, 2019.

[29] Z. Wang and Q. Liu, "A Software Test Case Automatic Generation Technology Based on the Modified Particle Swarm Optimization Algorithm," *2018 International Conference on Virtual Reality and Intelligent Systems (ICVRIS).* doi:10.1109/icvris.2018.00045, 2018.

[30] K.-W. Shin and D.-J. Lim, "Model-based automatic test case generation for automotive embedded software testing," *International Journal of Automotive Technology*, 19(1), 107–119. doi:10.1007/s12239-018-0011-6, 2017.

[31] C. Ma and J. Provost, "A model-based testing framework with reduced set of test cases for programmable controllers," *2017 13th IEEE Conference on Automation Science and Engineering (CASE).* doi:10.1109/coase.2017.8256225.

[32] M. Elqortobi, A. Rahj, J. Bentahar, and R. Dssouli, "Test Generation Tool for Modified Condition/Decision Coverage: Model Based Testing," *In Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications (SITA'20).* Association for Computing Machinery, New York, NY, USA, Article 38, 1–6. DOI:https://doi.org/10.1145/3419604.3419628, 2020.

[33] R. Ibrahim, M. Ahmed, R. Nayak and S. Jamel, "Reducing Redundancy of Test Cases Generation using Code Smell Detection and Refactoring". *Journal of King Saud University - Computer and Information Science,* Volume 32, Issue 3, March 2020.

[34] GitHub. *GitHub Repository*. [Online]. Available: https://github.com/

[35] Клуб анонимных айтишников, *GitHub repository for calculator-unit-test-example-java.* [Online]. Available: https://github.com/kranonit/calculator-unit-test-example-java.

[36] Samah AbuSalim. *GitHub repository for ATMMachine.* [Online]. Available: https://github.com/samahwaleed/ATMMachine.

[37] Danish Mohd, *GitHub repository for Blackjack-game-in-java*, https://github.com/DanisHack/Blackjack-game-in-java.

[38] Yvan Martin. *GitHub Repository for traffic-light-simulation.* [Online]. Available: https://github.com/ymartin/traffic-light-simulation.

[39] Annu Dalal. *GitHub Repository for Airline-Reservation-System.* [Online]. Available: https://github.com/annudalal/Airline-Reservation-System.

[40] Khesualdo Condori. *GitHub Repository for Elevator-Scheduling-Simulator*. [Online]. Available: https://github.com/00111000/Elevator-Scheduling-Simulator.

[41] JUnit. *JUnit4.* [Online]. Available: https://junit.org/junit4/.

[42] Cédric Beust and Hani Suleiman. "Next Generation Java Testing: TestNG and Advanced Concepts." Addison-Wesley Professional, 2007.

[43] Oshin and V. Chaudhary, "Comparison Analysis of TestNG and JUnit frameworks for Automation with Java," *Journal of Emerging Technologies and Innovative Research (JETIR)*, June 2018, Vol 5. Issue 6.