2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI) | 979-8-3503-9523-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICMCSI61536.2024.00063

Software Quality Testing Framework based on Machine Learning Analysis

Rongrong Li
Nanyang Technological University, 50 Nanyang Ave, Singapore, 639798
lirongronglee@proton.me

***Abstract*— This research work presents a software quality testing framework based on machine learning analysis. The framework utilizes dynamic symbol technology and integration testing methods to analyze different execution paths, thereby establishing a comprehensive integration testing framework. Technologies such as robot framework, exploration-driven test data generation, and software reliability coupling measurement are employed to improve testing efficiency and ensure thorough verification of software functions and performance. The research demonstrates the application of reinforcement learning to test case sequencing, using Q-learning to optimize API functional test case generation. The proposed methodology involves the integration of machine learning analysis into three aspects: information handling, procedure formulation, and execution flow. The paper explores regression testing, test case prioritization technology (TCP), and reinforcement learning for efficient test case ordering. A comprehensive simulation of 500 software reliability testing use cases shows significant improvements in test efficiency by reducing redundant instances. The research concludes with a discussion of the application of Q-Learning in continuous integration testing, emphasizing the need for flexible memory representations to handle complex states and action sets. The proposed framework effectively addresses the challenges posed by scale expansion in software development, thereby improving the accuracy and efficiency of software testing.**

Keywords*— *Machine learning analysis; software quality; testing framework; robustness testing

I. INTRODUCTION

The primary function of the software dependability [1]-[5] examination scenarios in software creation is to assess the trustworthiness of the software and the pinpoint potential susceptibilities promptly for timely rectification. With the expansion of software dimensions, the quantity of examination scenarios escalates, heightening the intricacy of software dependability appraisal. To enhance precision and efficacy in examination, a meticulous scrutiny of requisites for software dependability assessment is imperative, accompanied by a reduction in examination scenarios through streamlining and enhancement. The formulation of dependability examination scenarios should encompass all facets of modern software, encompassing utility, effectiveness, safeguarding, etc.

By employing adept examination scenario formulation, diverse plausible operational scenarios of the software can be comprehensively covered, offering a more faithful emulation of conceivable issues during the practical application.

Nevertheless, as software size burgeons, the number of the examination scenarios burgeons exponentially, intensifying the intricacy of the evaluation. Overcoming this obstacle mandates a meticulous exploration of software dependability assessment prerequisites. Identification of pivotal examination scenarios and central functionalities enables the optimization of examination scenario formulation to assure comprehensive coverage of the pivotal software sections. Concurrently, diminution methods are applied to curtail the number of the examination scenarios through the elimination of redundant instances, thereby refining examination efficiency.

Meticulously crafting and refining software dependability examination scenarios facilitates the more efficacious detection of the potential software glitches, enabling the expeditious resolution of issues. Such an approach not only elevates the precision of software dependability assessment but also adeptly manages challenges posed by the expansion of the scale throughout the software development trajectory. Among all testing scenarios, the XML abstracting [6]-[8] is the key step and is presented in the Figure 1.

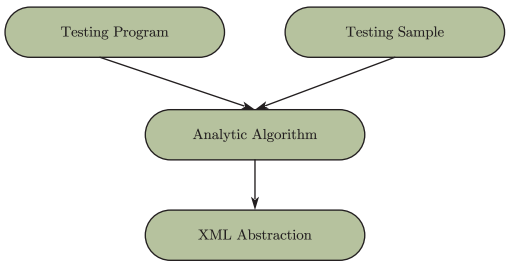

Fig. 1. The XML Abstracting for Testing

XML is the concise language for storing data that has unique properties that transcend operating systems and programming language development platforms. This allows for streamlined data interaction across disparate systems. As a meta-markup language, XML uses a set of tags to describe data elements and facilitates data communication through structured and self-explanatory tags, providing unparalleled flexibility in the data structuring. Software developers can significantly improve code maintainability by using custom tags to precisely articulate data content. The enumerated characteristics of the XML language enable seamless data exchange between the disparate systems, platforms, and applications, solidifying its status as a ubiquitous standard. Its power lies in its ability to provide comprehensive solutions for some diverse applications, simplifying data exchange and communication. In essence, as a universal standard, XML's

979-8-3503-9523-5/24/$31.00 ©2024 IEEE
DOI 10.1109/ICMCSI61536.2024.00063

adaptability and extensibility brings considerable convenience to the field of software development. Then, in the rest of the paper, the software quality testing framework based on the machine learning analysis is studied.

II. Literature Review

It's common knowledge that the dependability of the program is directly linked to the expenses and excellence of developing software. Verification stands out as a pivotal element within the software advancement journey. Its primary objective is to unearth and identify plausible imperfections in the software, ensuring the ultimate product's excellence. Through a methodical verification procedure, diverse software anomalies can be adeptly brought to light. Within the verification domain, the critical undertaking is devising a method to proficiently generate test facts that align with predefined criteria. This operation not only aids in affirming the constancy and trustworthiness of the software under varied scenarios but also facilitates the early identification of latent issues. Consequently, prompt rectifications can be executed before the software is officially unveiled, guaranteeing that the end product adheres to elevated quality benchmarks.

Hence, the verification process assumes a pivotal role across whole life-cycle of software development, rendering indispensable backing for upholding software reliability and quality. Exploration into Machine Learning (ML) testing has garnered substantial interest, with Zhang and colleagues (2020) providing an exhaustive examination encompassing diverse facets of ML testing. Covering 144 papers, this review spans properties, components, workflows, and application scenarios, offering valuable insights into emerging trends, persistent challenges, and promising directions within the realm of ML testing [9]. In the domain of software engineering (SE) literature reviews, Garousi et al. (2019) furnish specialized guidelines tailored for multivocal literature reviews (MLRs), underscoring the importance of integrating gray literature. Their guidelines intricately detail the planning, execution, and reporting phases, amalgamating insights from established SE guidelines and their experiential wisdom [10].

Within the sphere of deep learning (DL) system testing, Kim et al. (2019) introduce an innovative test adequacy criterion labeled Surprise Adequacy for Deep Learning Systems (SADL). Founded on the behavioral nuances of DL systems concerning training data, this criterion places emphasis on the unexpectedness of inputs. Empirical validations showcase its efficacy in enhancing classification accuracy, particularly in the face of adversarial examples [11]. Confronting the pivotal facet of statistical model assessment, Lüdecke et al. (2021) unveil the "performance" R package, furnishing utilities to appraise model quality and fit, along with tools to navigate diverse challenges during the analytical process [12].

The intricacies of class imbalance in Software Defect Prediction (SDP) datasets are tackled by Feng et al. (2021) through the introduction of Complexity-based OverSampling TEchnique (COSTE). COSTE engineers synthetic instances by amalgamating pairs of flawed instances with the akin complexity, achieving equilibrium between the over-generalization and diversity. Experimental findings underscore COSTE's efficacy in augmenting diversity without compromising the detection of defects, positioning it as a commendable alternative to redress the class imbalance challenges encountered in SDP [13].

III. Proposed Methodology

A. The Dynamic Symbol Technology, Integration Testing Overview

Dynamic symbol technology [14]-[15] is a technology that performs the actual execution and symbolic execution simultaneously. In a program, actual execution is used for operations or variables that are not associated with symbols, while symbolic execution is used for operations or variables that are associated with symbols. Symbolic execution treats each input variable in the program as a symbol and uses the input symbols to execute constraints on different paths. This technique effectively analyzes the different execution paths of a program by simulating program execution and collecting constraint information on all the paths. In the Figure 2, the program code conversion symbolic execution flow chart is illustrated.

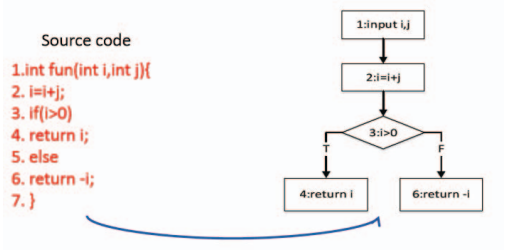

Fig. 2. The Program Code Conversion Symbolic Execution Flow Chart

Integration testing techniques can be categorized into the progressive and non-progressive methods. In the progressive testing approach, there are three methods: upward progression, downward progression, and layered integration. The upward progression method can be further classified into in-depth upward progression and breadth-first upward progression.

These techniques offer distinct advantages during the integration phase of software development, allowing efficient identification of system defects and systematic establishment of a comprehensive integrated testing framework.

The selection of appropriate integration testing approaches is critical to improving the testing efficiency and ensuring thorough validation of software functionality and performance across different levels and aspects. Breadth-first top-down integration testing is the considered scheme and presented in the Figure 3.

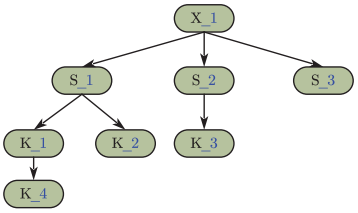

Fig. 3. The Breadth-first Top-down Integration Testing Scheme

B. The Initial Software Quality Testing Framework

The initial testing framework contains 3 aspects:

1. Robot Framework automated testing framework;
2. Software test data generation method;
3. Software reliability coupling metrics.

Robot Framework [16]-[17], as an open source general automated testing framework, is characterized by keyword drive and has the ability to integrate with the other tools to provide users with powerful and flexible automated testing solutions. The framework not only provides a rich base library and advanced interfaces, but also meets the testing needs of various functional types. Through its flexibility and also integration, Robot Framework has become an important tool in the field of automated testing, providing users with a wide range of application possibilities.

In the realm of source code-focused automated software test data [18]-[20] creation, exploration-driven testing approaches achieve the automatic generation of test data through the application of enhancement techniques. This sector has garnered extensive attention and thorough exploration by numerous scholars globally. The fundamental objective of automated testing is to formulate the test cases aligning with the specified coverage criteria, typically encompassing conditional inclusion, statement integration, branch incorporation, and path incorporation. The efficacy of test data creation relies on the standards taken into account, and program testing methods grounded in branch and path inclusion are commonplace preferences. Nevertheless, due to the relatively diminished potency of the branch inclusion, heightened program intricacy results in a proliferation of path inclusion. Consequently, generating adequately potent test data to meet path inclusion has emerged as a contemporary research focal point. Investigating technology for test data generation oriented towards path inclusion standards is pivotal for adeptly generating test data spanning diverse program pathways. Relative to manually composed test scenarios, the exploration-driven test data generation method adeptly pinpoints unidentified errors while enhancing code coverage, thereby significantly amplifying the efficiency of test data generation. Within this domain, numerous enhancement algorithms, including genetic algorithms, particle swarm optimization algorithms, artificial bee colony algorithms, hill-climbing algorithms, differential evolution algorithms, etc., have gained widespread application. These enhancement algorithms aim to explore test data within the program input domain that satisfies the given requirements, furnishing robust backing for automated test data generation.

In the streamlined optimization of software reliability test cases, K evaluation indicators are selected as decision targets to reduce the number of software reliability test cases during optimization process. This means that during the optimization process, the system considers K different criteria to achieve more effective and efficient software reliability testing. Such a multi-index optimization method helps to comprehensively consider different aspects of the requirements, and thus more comprehensively evaluate and improve software reliability.

First, we calculate the weight corresponding to the core decision-making objectives, and then rank the decision-making objectives according to the calculation results as:

$$H = (h_1, h_2, h_3, \ldots, h_k) \tag{1}$$

The established optimization objective function is defined as follows:

$$\begin{cases} H_1(t_i) = \underset{t_i \in T}{optimization} h_1(t_i) \\ H_2(t_i) = \underset{t_i \in T}{optimization} h_2(t_i) \\ \vdots \\ H_k(t_i) = \underset{t_i \in T}{optimization} h_k(t_i) \end{cases} \tag{2}$$

Then, through the integration of machine learning, the optimization work of solution can be achieved.

C. The Testing Framework based on Machine Learning Analysis

For achieving the machine learning based testing, the 3 aspects should be considered:

(1) Information handling. Input information for automated knowledge models is generally gathered or further generated manually and is susceptible to disturbances and inaccuracies. Therefore, it is essential to organize the model information, rectify anomalies, and execute pre-processing algorithms such as normalization, information enrichment, and characteristic extraction. The objective is to eliminate disturbances and incorrect information from the initial input while extracting meaningful data characteristics.

(2) Procedure formulation. Differing from customary programs, the procedure of the automated knowledge model does not manually construct the logic of input and output. Conversely, the procedural process is strategized by crafting model operators, formulating model structure, and creating neural network calculation graphs to guide the model in accomplishing learning tasks such as clustering, classification, and regression based on provided data.

(3) Execution progression. Leveraging the pre-processed information and model calculation graph, the automated knowledge model employs tools like calculation libraries to bring to life the operational functions of each operator within the model. Throughout the training iteration, the weights of the model operators are adjusted, ultimately fulfilling the designated automated knowledge task. This progression relies on the computational libraries like automated knowledge frameworks to guarantee the model's ability to assimilate and comprehend patterns within the information.

During software testing, regression testing must be performed because code changes may introduce new bugs or cause other module problems. Regression testing is an important part of the software development cycle. Test Case Prioritization Technology (TCP) is an important regression testing method that aims to find the best test sequence that can quickly detect defects through specific sequencing criteria to improve software testing efficiency. We have a formal description of the TCP problem and what follows is definition:

T: testing samples.

PT: permutation set of the execution order of all test cases.

f: a function that maps PT to real numbers.

The problem to be solved in test case prioritization is to find an $T' \subseteq PT$ such that satisfies:

$$\left[f(T') \geq f(T^n)\right]\left(\forall T^n\right)\left(T^n \subseteq PT\right)\left(T^n / T' \neq 1\right) \quad (3)$$

In reinforcement learning [21]-[24] test case ordering, the reward function plays a key role in guiding the effectiveness of test case ordering. A good reward function can effectively guide the agent to gradually optimize the order of test cases, quickly detect defects, and improve testing efficiency. The design of the reward function is related to the direction of the agent's learning. With the right reward signal, the agent can learn a more sensitive software defect sorting method. This method uses reinforcement learning to enable test cases to be executed in a more efficient order, thereby detecting potential problems earlier. In test case sorting, careful design of the reward function is a key factor in ensuring that the agent can learn efficiently and improve the efficiency of software testing.

Reinforcement learning differs from traditional machine learning in that it does not directly tell the agent what to do, but rather optimizes behavior through continuous trial and error. The agent acts as the "brain" in reinforcement learning and performs iterative optimization through the feedback information after receiving external signals. Figure 4 shows the basic structure of the reinforcement learning. The agent continues trial and error in the environment and pursues the maximum expected benefit by obtaining environmental rewards. That is, the agent makes decisions based on feedback from the environment to achieve maximum benefit. The characteristic of reinforcement learning is that it relies on the agent's ability to continuously adjust its strategy through interaction with the environment to achieve the best decision. In this way, the agent can learn through the experience and gradually improve its behavior to achieve maximum benefit. This has significant advantages for solving the problems that require continuous optimization and adaptation.

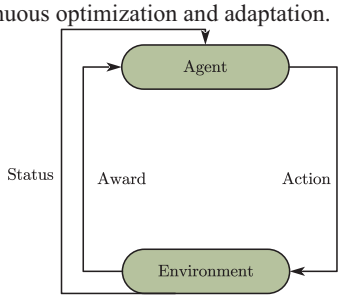

Fig. 4. The Reinforcement Learning Scheme

In reinforcement learning, the agent's memory typically uses a tabular representation method to store and update experience in the form of tables. Although this direct table representation allows the agent to extract experience and learn quickly, it stores each state-action pair separately, requiring states and actions to be bounded and discrete. As the dimensions of the state and action sets become more complex, table-based agents face challenges in learning efficiency. In the test optimization problem for continuous integration, which involves a continuous decision-making process, each continuous integration cycle requires regression testing of test cases. Therefore, it is difficult for the existing table-based reinforcement learning methods to meet the requirements of continuous integration testing. For test optimization problems, memory representations that are more flexible and adaptable to complex states and action sets are needed to improve learning efficiency and adaptability. This study uses the Q-Learning reinforcement learning method to solve the problem of API functional test case generation. It is necessary to construct a state-action Q-table to represent the test agent Xbot. The Q-table adopts the table form shown in Figure 5, where the value corresponding to each (s, a) represents the Q-value of selecting action a in state s. In a state, the larger the Q-value of the corresponding action, the greater the possibility of selecting that action. Although the Q-table is a simple representation method, its states and actions must be discrete sets of finite size because each state-action pair is stored separately. For the API interface of web applications, the number of existing states is limited, so the Q-table is still applicable. This representation method allows the Xbot to continuously update the Q-value through learning experience, thereby guiding test case generation and improving the agent's performance in API functional testing.

	Action1	Action2	Action3	...	Action N
State1	0.00	0.00	0.00	...	0.00
State2	0.00	0.00	0.00	...	0.00
State3	0.00	0.00	0.00	...	0.00
...	0.00	0.00	0.00	...	0.00
StateN	0.00	0.00	0.00	...	0.00

Fig. 5. The Q-table

Int he Q-table, it represents the Q value corresponding to the state action in reinforcement learning. Each row represents a state, each column represents an action, and each element in the table represents the Q-value of selecting the corresponding action in a given state. Initially, all Q values are 0.00, meaning that the agent's initial knowledge of each state-action pair is zero. These Q values are constantly updated as the agent interacts and learns from the environment.

As the Q value increases, the benefit of choosing the corresponding action in the corresponding state increases, reflecting the optimal decision strategy that the agent is gradually learning. Through this basic Q-table, the agent can optimize test case generation in the process of continuous trial and learning to improve test efficiency and performance. Then, the testing work can be finalized through the calculation of the Q status.

IV. Simulation

Now set the number of use cases required in the software reliability testing process to 500, and the total number of test cases to 600 as the simulation sets. F represents the running cost of the reduction use case, and its calculation formula is as follows:

$$F = \frac{|V| - |V'|}{|V|} \times 100\% \quad (4)$$

|V'| represents the running cost of the reduced use case in the software reliability testing process; |V| represents the running cost of the original use case in the software reliability testing process. In the Table 1, the tests under 10 sets of experiment are demonstrated.

Table 1. The F Value Tests under 10 Sets of Experiment

Set	*F Value(%)*
1	95.73
2	95.26
3	95.82
4	95.63
5	95.45
6	95.12
7	95.22
8	95.62
9	95.63
10	95.26

Table 1 shows the results of the F value test conducted in 10 sets of experiments. Each set of experiments corresponds to an F-value percentage that reflects the specific data or phenomenon under that experimental condition. For example, the F value of the first set of experiments is 95.73%, the second set is 95.26%, and so on. These data can be used to analyze differences or trends between experiments and help understand the consistency and reliability of the experimental results. The test path generation and assembly is performed using the constructed and initialized Q-table type test agent Xbot. Based on a specific algorithm, Xbot continuously generates test paths with the highest risk priority in the current situation. First, starting from the given initial state, Xbot selects executable actions according to the ε-greedy state transition strategy and adds the next arriving state to the test path. As the test path is gradually generated, Xbot accurately builds a path by continuously evaluating risk priorities. If the generated test path satisfies the termination conditions, Xbot successfully generates a test path with the highest risk priority in the current situation. This process helps to systematically test the software and ensures that the most critical test paths are prioritized in the face of potential risks. After the combination is completed, each test case will generate a fixed-format JSON file that can be executed. Specific request headers are added to the JSON file, such as Cookie, Content-Type and other parameters. The Figure 6 demonstrates the JSON file.


```
{
{
"matched_method":"GET",
"request":{
"method":"GET",
"requestHeaders":[
{
"name":"Accept",
"value":"application/json, text/javascript, */*; q=0.01"
},
{
"name":"Cookie",
"value":"JSESSIONID=464FD5F83E905BE772BAF82FE268B0B4"
}
],
"url":"http:xxxxxxx"
},
```


Fig. 6. The Test Case of JSON File

V. CONCLUSION

The proposed research experiment has used 500 use cases for software reliability testing and set the total number of test cases to 600. The results show that by reducing redundant instances, our approach significantly improves testing efficiency while accurately assessing software reliability. In addition, we demonstrate the application of reinforcement learning to test case sequencing and design an effective reward function to guide the agent to optimize the test case sequence to quickly detect defects and improve testing efficiency. In conclusion, our research shows that a software quality testing framework based on machine learning analysis can effectively manage and cope with the challenges posed by scale expansion in the software development process, thereby improving the accuracy and efficiency of software testing.

REFERENCES

[1] Longo, Francesco, Salvatore Distefano, Dario Bruneo, and Marco Scarpa. "Dependability modeling of software defined networking." Computer Networks 83 (2015): 280-296.

[2] Natella, Roberto, Domenico Cotroneo, and Henrique S. Madeira. "Assessing dependability with software fault injection: A survey." ACM Computing Surveys (CSUR) 48, no. 3 (2016): 1-55.

[3] Khrennikov, A. Y., Aleksandrov, N. M., & Radin, P. S. (2020). Dependability of service of substation electrical equipment: Estimation of the technical condition state with the use of software and information tools. In Engineering in Dependability of Computer Systems and Networks: Proceedings of the Fourteenth International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, July 1–5, 2019, Brunów, Poland (pp. 274-283). Springer International Publishing.

[4] Bernardi, Simona, José Merseguer, and Dorina C. Petriu. "Dependability modeling and analysis of software systems specified with UML." ACM Computing Surveys (CSUR) 45, no. 1 (2012): 1-48.

[5] Chatterjee, Subhashis, and Bappa Maji. "A fuzzy logic-based model for classifying software modules in order to achieve dependable software." International Journal of Service Science, Management, Engineering, and Technology (IJSSMET) 11, no. 4 (2020): 45-57.

[6] Nečaský, Martin, Jakub Klímek, Jakub Malý, and Irena Mlýnková. "Evolution and change management of XML-based systems." Journal of Systems and Software 85, no. 3 (2012): 683-707.

[7] Bischof, Stefan, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. "Mapping between RDF and XML with XSPARQL." Journal on Data Semantics 1 (2012): 147-185.

[8] van Gompel, Maarten, and Martin Reynaert. "FoLiA: A practical XML format for linguistic annotation–a descriptive and comparative study." Computational Linguistics in the Netherlands Journal 3 (2013): 63-81.

[9] Zhang, Jie M., Mark Harman, Lei Ma, and Yang Liu. "Machine learning testing: Survey, landscapes and horizons." IEEE Transactions on Software Engineering 48, no. 1 (2020): 1-36.

[10] Garousi, Vahid, Michael Felderer, and Mika V. Mäntylä. "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering." Information and software technology 106 (2019): 101-121.

[11] Kim, Jinhan, Robert Feldt, and Shin Yoo. "Guiding deep learning system testing using surprise adequacy." In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1039-1049. IEEE, 2019.

[12] Lüdecke, Daniel, Mattan S. Ben-Shachar, Indrajeet Patil, Philip Waggoner, and Dominique Makowski. "performance: An R package for assessment, comparison and testing of statistical models." Journal of Open Source Software 6, no. 60 (2021).

[13] Feng, Shuo, Jacky Keung, Xiao Yu, Yan Xiao, Kwabena Ebo Bennin, Md Alamgir Kabir, and Miao Zhang. "COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction." Information and Software Technology 129 (2021): 106432.

[14] Mossberg, Mark, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts." In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1186-1189. IEEE, 2019.

[15] Li, Dongcheng, W. Eric Wong, Shenglong Li, and Matthew Chau. "Improving search-based test case generation with local search using adaptive simulated annealing and dynamic symbolic execution." In 2022 9th International Conference on Dependable Systems and Their Applications (DSA), pp. 290-301. IEEE, 2022.

[16] Coronado, Enrique, Fulvio Mastrogiovanni, and Gentiane Venture. "Design of a human-centered robot framework for end-user programming and applications." In ROMANSY 22–Robot Design, Dynamics and Control: Proceedings of the 22nd CISM IFToMM Symposium, June 25-28, 2018, Rennes, France, pp. 450-457. Springer International Publishing, 2019.

[17] Stresnjak, Stanislav, and Zeljko Hocenski. "Usage of robot framework in automation of functional test regression." In Proc. 6th Int. Conf. Softw. Eng. Adv.(ICSEA), pp. 30-34. 2011.

[18] Li, Yichen, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R. Lyu. "Exploring the effectiveness of llms in automated logging generation: An empirical study." arXiv preprint arXiv:2307.05950 (2023).

[19] Arcuri, Andrea. "RESTful API automated test case generation with EvoMaster." ACM Transactions on Software Engineering and Methodology (TOSEM) 28, no. 1 (2019): 1-37.

[20] Gambi, Alessio, Marc Mueller, and Gordon Fraser. "Automatically testing self-driving cars with search-based procedural content generation." In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 318-328. 2019.

[21] Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu. "Offline reinforcement learning: Tutorial, review, and perspectives on open problems." arXiv preprint arXiv:2005.01643 (2020).

[22] Brunke, Lukas, Melissa Greeff, Adam W. Hall, Zhaocong Yuan, Siqi Zhou, Jacopo Panerati, and Angela P. Schoellig. "Safe learning in robotics: From learning-based control to safe reinforcement learning." Annual Review of Control, Robotics, and Autonomous Systems 5 (2022): 411-444.

[23] Moerland, Thomas M., Joost Broekens, Aske Plaat, and Catholijn M. Jonker. "Model-based reinforcement learning: A survey." Foundations and Trends® in Machine Learning 16, no. 1 (2023): 1-118.

[24] Shinn, Noah, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. "Reflexion: Language agents with verbal reinforcement learning." In Thirty-seventh Conference on Neural Information Processing Systems. 2023.