AUTO TEST CASES GENERATION USING MODEL DRIVEN ARCHITECTURE

Prof. Hirnawale S.B.[1], Bankar Mayur Dada[2], Yadav Sanket Pratap[3],Kamble Rushikesh Kisan[4], Zagade Onkar Arjun[5]

[1] *Professor, Department Of Computer Engineering, HSBPVT's Parikrama College of Engineering Kashti, Maharashtra, India.*
[2] *Student, Department Of Computer Engineering, HSBPVT's Parikrama College of Engineering Kashti, Maharashtra, India.*
[3] *Student, Department Of Computer Engineering, HSBPVT's Parikrama College of Engineering Kashti, Maharashtra, India.*
[4] *Student, Department Of Computer Engineering, HSBPVT's Parikrama College of Engineering Kashti, Maharashtra, India.*

ABSTRACT

Automatic Test Case Generation (ATCG) plays a pivotal role in ensuring software quality and reliability. This paper presents an innovative approach that leverages Model-Driven Architecture (MDA) and Natural Language Processing (NLP) to automate the process of generating test cases. MDA provides a structured foundation for creating models at various levels of abstraction, while NLP facilitates the transformation of natural language requirements into machine-readable models. The proposed method focuses on analyzing textual requirements, extracting key information, and transforming it into models, which are then used to automatically generate comprehensive test cases. This approach aims to enhance test coverage, reduce manual effort, and improve the alignment between test cases and original requirements. We discuss the advantages, challenges, and potential applications of this model-driven, NLP-based ATCG approach, offering a promising direction for advancing software testing practices in the ever-evolving landscape of software development.

Keyword : - *Software Testing , Test Cases, Code Smell, and System Optimization etc….*

1. INTRODUCTION

"Automatic test case generation" is a crucial aspect of software testing, and it plays a vital role in ensuring the quality and reliability of software applications. It involves the automated creation of test cases to evaluate the functionality, performance, and security of software systems. This process can significantly reduce the manual effort required for testing, increase test coverage, and help in identifying defects early in the software development life cycle.

The traditional manual test case creation process is often time-consuming and error-prone, as it relies on human testers to design and execute test cases based on system requirements. In contrast, automatic test case generation leverages various techniques and tools to automate this process.

One approach to automatic test case generation involves Model-Driven Architecture (MDA), which emphasizes the use of models to represent different aspects of a software system. MDA allows for the creation of abstract and platform-independent models that describe the system's structure and behavior. These models can serve as a foundation for automatically generating test cases, ensuring that they cover the essential functionalities and interactions within the system.

Another important component of automatic test case generation is Natural Language Processing (NLP). NLP techniques are employed to extract and understand the requirements and specifications of the software system expressed in natural language. By converting these requirements into machine-readable formats, such as models or specifications, NLP bridges the gap between human-readable descriptions and automated testing.

The combination of MDA and NLP in automatic test case generation offers several advantages, including improved test coverage, early defect detection, reduced manual effort, and better alignment between test cases and requirements. However, this approach is not without its challenges, such as the complexity of NLP processing, the quality of natural language requirements, and the need for manual model refinement.

Overall, automatic test case generation using MDA and NLP is a promising approach to streamline the testing process, enhance the quality of software, and accelerate the software development life cycle. It represents a fusion of cutting-edge technologies that can help organizations improve their testing practices and deliver more reliable and robust software applications to their users.

2. LITURATURE AND SURVEY

Nirpal and Kale et al (2010) described the approach of test data and case generation automatically using genetic algorithm for path testing. Then compared their results with the research of Yong Chen strategy (Chen Yong, 2009) of generation of test data. They proved that their approach performed
Better as compared to Yong Chen approach, as the genetic algorithm reduces the overall time required for long software testing process by generating proper and more suitable test cases for path testing.

Jung sup Oh et al (2011), Genetic algorithm was used by the author in State flow models for
Transition coverage. A tool was introduced to implement this approach of genetic algorithm which was examined on three criteria. When this genetic algorithm was implemented and compared with random search and State Flow which is a commercial tool for model testing, the genetic algorithm achieved better cover

In the study by Zhenzhen Wang, and Qiaolian Liu [6] discuss an improved Particle Swarm Optimization (PSO) algorithm to generate test case automatically. This study evaluated several techniques such as PSO and Genetic algorithm to determine perfomance in term of generating test cases. Based on the result, shows that the improved PSO is effectively reduce run time in generating software test cases automatically compare to conventional PSO

3. SYSTEM IMPLEMENTATION PLAN

1. Model Representation: - Choose a model representation framework or library in Python. One popular choice is the Eclipse Modeling Framework (EMF) for Python, which provides support for modeling languages like UML and allows you to create and manipulate models in Python.

2. Transformation Rules: - Define transformation rules and model-to-code mappings in Python. You can write Python functions or scripts that take elements of your models and generate test cases accordingly. These transformation rules will be a crucial part of your MDA approach.

3. User Interface: - Create a user-friendly interface for users to input their models and specify test case generation preferences. You can use Python GUI libraries like Tkinter or PyQt to build the interface.

4. Model-to-Test Case Generation: - Develop Python-based algorithms for model-to-test case generation. This could involve code generation from models, using Python's code generation libraries, or other techniques, depending on your specific requirements.

5. Testing and Validation: - Implement testing features in Python to validate the generated test cases. This might include running the generated test cases against the software under test and assessing code coverage and compliance with coding standards.

6. Integration: - Integrate the generated test cases into your organization's testing frameworks and pipelines, which might include using Python testing frameworks like pytest.

7. Documentation and Training: - Create documentation using Python documentation tools or libraries, and conduct training sessions for users and maintainers as needed.

8. Deployment: - Deploy your Python-based automated test case generator within your organization. Ensure proper integration with your existing development and testing processes.

9. Maintenance and Updates: - Regularly maintain and update the Python-based system as new requirements or issues arise.

10. Monitoring and Optimization: - Implement monitoring and performance optimization in Python to ensure the generator remains efficient and effective

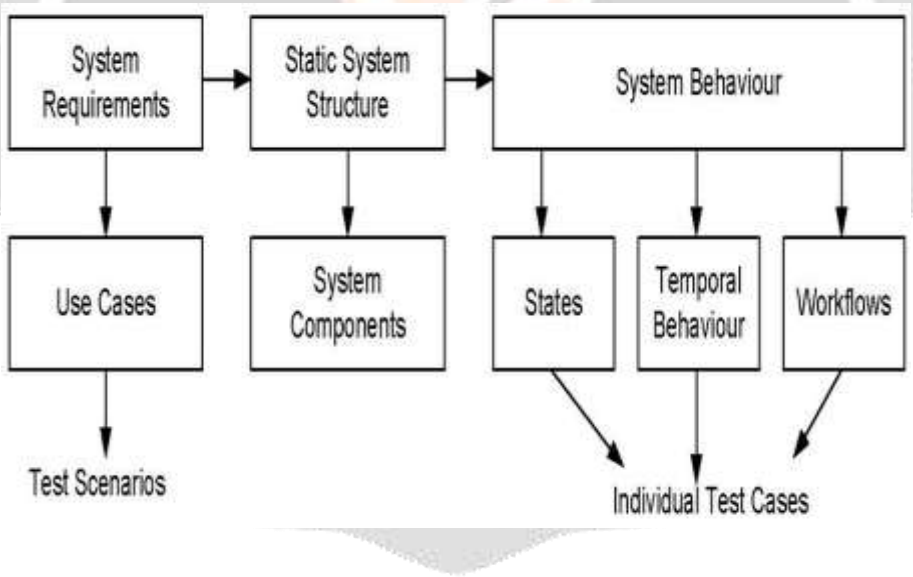

Fig -1 UML DIAGRAM

4. CONCLUSIONS

Automatic test case generation is one way of optimizing the software testing process, which can help in minimizing the required time and effort for testers. In this thesis, an approach for automatic test case generation from natural language test case specifications has been designed, applied, and evaluated. The effectively implemented approach consists of a pipeline of three stages. The first stage of the approach involves parsing and analyzing the test case specification documents using different NLP techniques. Feature vectors, containing keywords representing the

specification are generated during this stage. In the next stage, the feature vectors are mapped to label vectors consisting of C# test scripts file names. The feature and label vectors are used as input and output, respectively, in the last stage in the text classification process. Finally, the proposed approach is applied and evaluated on an industrial testing project at Ericsson AB in Sweden and the approach has been implemented as a tool for aiding testers in the process of test case generation.

5. REFERENCES

[1] [1]. S. Tahvili. "Multi-Criteria Optimization of System Integration Testing". PhD thesis. Malardalen University, Dec. 2018. IsBn: 978-91-7485-414-5.

[2] [2]. S. Tahvili et al. "Cost-Benefit Analysis of Using Dependency Knowledge at Integration Testing". In: The 17th International Conference On Product-Focused Software Process Improvement. Nov. 2016.

[3] [3]. R. P. Verma and M. R. Beg. "Generation of test cases from software requirements using natural language processing". In: 2013 6th International Conference on Emerging Trends in Engineering and Technology. IEEE. 2013, pp. 140–147.

[4] [4]. S. Tahvili et al. "sOrTES: A Supportive Tool for Stochastic Scheduling of Manual Integration Test Cases". In: Journal of IEEE Access 6 (Jan. 2019), pp. 1–19.