A Novel Approach to Automated Test Script Generation using Large Language Models for Domain-Specific Languages

Jianing Sun*, Jiahui Wang, Yuyan Zhu, Xingyu Li, Ying Xie and Jiaxin Chen

Chongqing University, 400044 Chongqing, China

Abstract

This paper presents a novel method for generating automated test scripts for Domain-Specific Languages (DSLs) in software testing, particularly for the automotive industry. It emphasizes the growing importance of software testing in ensuring product quality amid IT advancements. The paper reviews software testing's evolution, modern processes, and the role of Large Language Models (LLMs). It highlights DSLs' significance and uses the automotive sector to show how LLMs can automate test script generation. Tests indicate that in cases with a small sample size, the effectiveness of prompt engineering is superior to model fine-tuning. The proposed method thus relies on well-designed prompts to direct LLMs to produce accurate scripts. The generation system's overview is discussed, along with an evaluation of the scripts' quality using metrics like Levenshtein Distance. Results indicate that LLMs boost test automation, defect detection, and software reliability. Future work will optimize these tools for higher testing automation levels.

Keywords

software testing, domain-specific languages, large language models, Levenshtein Distance

1. Introduction

Software testing is a key component in ensuring the quality and reliability of software products. In the rapidly developing information technology era, software has become an indispensable part of our daily life and work. With the increasing complexity and diversification of software functions, the importance of software testing has become increasingly prominent. Software testing is a series of processes designed to check that a software product meets specified requirements and ensures its quality. It not only helps developers to find and fix defects, but also greatly enhances system security, especially in fields with high software safety requirements such as automotive and aviation [1].

1.1. A Brief History of Software Testing

The origins of software testing date back to the 1950s, focusing initially on debugging to identify and rectify software faults [2][3][4]. As software complexity grew, the need for independent testing organizations became apparent. In 1957, Charles Baker first defined program testing, in his review of the book Digital Computer Programming by Dan McCracken, separating it from debugging. Bill Hetzel formalized software testing as a concept at the University of North Carolina in 1972, establishing it to ensure a program performs as intended [5]. Glenford J. Myers further refined this in 1979, describing testing as executing a program to uncover errors [6].

By 1983, IEEE had standardized software testing, defining it as a process -manual or automated- to verify system requirements [7]. The 1990s brought agile methodologies, integrating testing and development and encouraging tester involvement from the earliest development stages [8]. In the 21st century, testing has advanced, with a focus on exploratory testing that highlights the tester initiative. The era of AI and big data has intensified scrutiny of software testing. Despite still leveraging 20th-century methods, the field anticipates future innovations, potentially revolutionizing testing practices [9].

1.2. Modern Approaches

The modern software testing process is crucial for ensuring software quality and functionality. It starts with requirement analysis, followed by developing a test plan, designing test cases, and preparing test data (**Figure 1**). The test environment is set up, tests are executed and recorded, and defects are tracked. Regression and performance testing are conducted,

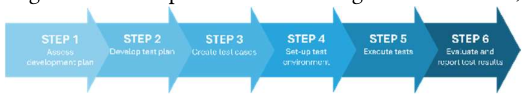

Figure 1: Modern software testing process.

QuASoQ 2024: 12th International Workshop on Quantitative Approaches to Software Quality, 3rd December 2024, Chongqing, China,

*Corresponding author.

j.sun@cqu.edu.cn (J. Sun)

0009-0001-4943-6038 (J. Sun); 0009-0000-0107-0666 (J. Wang); 0009-0008-1670-4607 (Y. Zhu); 0009-0009-4526-007X (X. Li); 0009-0003-8939-7832 (Y. Xie); 0009-0002-4492-4734 (J. Chen)

along with security and system testing. Acceptance testing confirms business requirements are met. Test reports summarize results, and evaluations identify process improvements.

Techniques like automated testing, Continuous Integration (CI), and Continuous Delivery (CD) enhance testing efficiency. Agile testing fosters collaboration between testers and developers. Performance, security, and mobile application testing ensure software reliability across different aspects. Cloud testing leverages cloud resources for extensive testing.

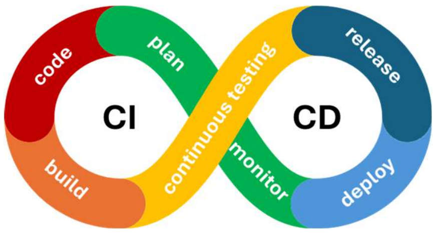

Figure 2: CI/CD is a software development practice in which code changes are automatically integrated, built, and tested, with successful builds being deployed to production.

AI-based testing leverages machine learning to automate software testing processes, enhancing efficiency and accuracy. It encompasses exploratory testing to identify issues without fixed test cases, ensuring broader coverage. Model-driven testing and testability design optimize test case generation and software sustainability. Additionally, managing test data and implementing strategies such as Test Left Shift and Test Right Shift further refine the development cycle. These dynamic approaches adapt to different methodologies, ensuring consistent software quality throughout the testing process.

1.3. Large Language Models

Large Language Models (LLMs) are cutting-edge AI specialized in natural language understanding and generation. Trained on extensive datasets and employing neural networks like Transformers, they capture linguistic subtleties and perform a variety of language tasks such as categorization, analysis, translation, and Q&A systems. They discern nuances, generate realistic text, and continuously adapt to linguistic evolution, raising concerns over data privacy and ethics.

LLMs have significantly impacted sectors like smart offices, travel, e-commerce, and government by enhancing efficiency and personalization. In software development, LLMs are revolutionizing the field. They aid in document summarization, provide travel advice, and improve user engagement. Tools like GitHub Copilot demonstrate their advantage by assisting in coding tasks [10].

Also, LLMs boost software testing by automating tasks, detecting defects, and ensuring reliability. They improve fuzz and unit testing, creating test cases, and suggesting fixes. Research shows their significant benefits in expanding test coverage and error detection. Future efforts will focus on optimizing testing tools and techniques.

2. Domain-Specific Software Testing

2.1. Domain-Specific Languages

Domain-specific languages (DSLs) are specialized languages designed for particular domains or tasks, offering simplified syntax for ease of use by domain experts [11]. They can be integrated with general-purpose languages (GPL) like Java and C++, enhancing development efficiency through tool support such as analyzers and compilers. DSLs are crucial in various industries, for example, HTML in web development and SQL in databases. They automate tasks like API documentation and legal document generation, improving efficiency and reducing errors. DSLs also facilitate team collaboration by allowing non-technical members to express requirements in a natural language-like format.

For software development, DSLs boost efficiency by simplifying complex representations and promoting code reusability. They accelerate prototyping and iteration, integrating seamlessly into existing tools and workflows. Testing DSL-developed programs requires a detailed plan with automated test scripts for regression testing. Test cases must be readable, and test data should reflect the domain specifics to ensure comprehensive coverage and identify defects. Maintainability of test cases and DSL is essential for ongoing development success.

2.2. Software Testing using DSL from the Automobile Industry

This section addresses the critical need for rigorous testing in passenger car product development, ensuring quality and performance meet standards.

Traditionally, automotive testing relies on manual, labor-intensive translation of requirements into test cases and scripts, causing significant strain on resources. To streamline this process and integrate Continuous Testing/Continuous Delivery

Table 1
A segment of the test data, describing the test cases, pre-conditions, and the desired test script DSL to be generated.

Test case	Pre-condition	Test script

Forward gear is activated when the car is moving forward	Configuration.Gears.Drive.is_activated()	Signals.Check(signals=[Gears_GearsStatus], values=[Gears_GearsStatus_shift], waiting_time=100)
		Gear.Shift('drive')

Reverse gear is activated when the car is being driven reversely	Configuration.Gears.Reverse.is_activated()	Signals.Check(signals=[Gears_GearsStatus], values=[Gears_GearsStatus_shift], waiting_time=100)
		Gear.Shift('reverse')

At driving P gear is deactivated	Configuration.Gears.Park.is_deactivated()	current_speed = car.get_speed()
		current_gear = car.get_gear_status()
		Self.assertNotEqual(current_gear, 'park')
		car.stop()
		car.shift_gear('park')
		current_gear = car.get_gear_status()

(CT/CD) pipelines, the industry is moving towards automated test development.

The automotive sector is in pursuit of an AI-driven solution to streamline the automation of test script generation for its proprietary DSLs, which are integral to the testing of a spectrum of automotive systems. The prevailing manual methodology is marred by inefficiencies, susceptibility to errors, and variability in code quality, alongside insufficient test coverage. By harnessing the capabilities of large language models, an AI-powered tool has the potential to orchestrate this process, amplifying efficiency, curbing errors, and upholding uniformity in code excellence, thereby conquering existing challenges and invigorating the software development lifecycle.

2.3. Sample Data

A total of 51 data samples (**Table 1**), each representing a true mapping from a test case to a test script in a particular DSL format.

For privacy protection purposes, all information, program code and data in this paper have been anonymized.

3. Approach

Broadly speaking, two dominant strategies have emerged for augmenting the knowledge base: the art of prompt engineering, which is particularly effective for modest datasets, and the process of model fine-tuning, which is best suited for addressing more substantial volumes of data. Considering the current data landscape, characterized by a dearth of samples and inherent uncertainties, a comprehensive evaluation was undertaken to compare the merits of both prompt engineering and LLM fine-tuning. This analysis has demonstrated that, under the present circumstances of limited data, prompt engineering emerges as the slightly superior approach.

Consequently, we have intentionally opted to employ the finesse of prompt engineering for the automated crafting of test scripts. This strategic choice is rooted in its proven ability to deliver optimized outcomes, even within the confines of our data scarcity. By leveraging the finesse of prompt engineering, we aim to transcend the limitations imposed by scant data availability, thereby enhancing the overall performance and reliability of our test script generation process.

An integral element of our approach is the selection of the foundational Large Language Model. To this end, we have undertaken a model selection process, meticulously assessing ChatGLM3, Llama3, and Qwen2. Following an exhaustive comparison, we determined that Llama3's generative capabilities align more closely with our requirements. Hence, we have chosen Llama3 to serve as the underlying LLM for this study.

3.1. Prompt to Make Precise Test Script Generation

Through a meticulous process of refinement, we've perfected our prompt for generating test scripts, as shown in the example. This fine-tuning ensures our AI model produces outputs that are both accurate and meet our objectives.

Our prompt is divided into four key components (**Table 2**): First, an exhaustive list of potential samples, excluding the current focus, provides a comprehensive training context. Second, we concentrate on the specific test purpose to create targeted, efficient test scripts. Third, we provide clear instructions in natural English for the LLM to follow, ensuring a seamless and accurate generation process. Lastly, we impose constraints to optimize the generation process, enabling our LLM model to autonomously produce precise and relevant test scripts without excessive input.

Table 2
Pseudo code of prompt design.

Prompt for test script generation (pseudo code)
dataFrame = All sample mappings except the one which is being generated
testCase = one test case which is being processed
instruction = "Above is a list of test cases and corresponding test scripts, assembled in Json format. Please generate test script for the following test case:"
condition = "Please export generated test script only, no leading text, no leading new lines."
prompt = dataFrame + CRLF + instruction + testCase + condition

This structured approach not only boosts script accuracy but also enhances the efficiency of our testing process, bringing us closer to our goal of fully automated AI-driven test script generation.

3.2. Test Script Generation System Overview

The test script generation system, illustrated in **Figure 3**, converts input test cases into executable scripts in the partner's DSL language, verifying product functionality. It uses outlined methodologies, and scripts are evaluated by experts for accuracy and reliability, with corrections made as needed.

Validated scripts are executed and stored, informing future prompts and enhancing script generation over time. This cycle of evaluation and learning improves script quality and reduces manual creation, aiming for an automated, self-improving system that streamlines software testing. As data storage grows, prompts become more complex, reflecting deeper learning and improved autonomy in script generation, ultimately advancing AI in software testing.

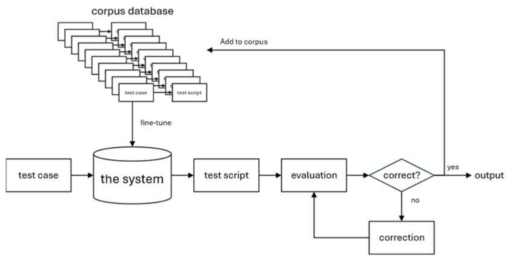

Figure 3: Generation System Overview.

4. Result and Evaluation

4.1. Evaluation Metric

This paper employs the Levenshtein Distance [12] to evaluate the textual accuracy of our language model, providing an objective measure of how closely generated text matches the ground truth. This edit distance metric, devised by Vladimir Levenshtein, quantifies the minimum number of single-character edits required to transform one string into another, offering insights into model performance. It plays a crucial role in fields like Natural Language Processing, where it assesses text similarity, and Bioinformatics, where it indicates genetic relatedness. Despite its higher computational demands for longer strings, our use of dynamic programming makes it an efficient tool for our analysis. The Levenshtein Distance aids in refining our model, ensuring that the text generation is both accurate and reliable.

The formal definition of Levenshtein Distance between two arbitrary strings a and b with length of $|a|$ and $|b|$ respectively is given by

$$\mathbf{lev}(a,b) = \begin{cases} |a|, & \text{if } |b| = 0, \\ |b|, & \text{if } |a| = 0, \\ \mathbf{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big), & \text{if } \operatorname{head}(a) = \operatorname{head}(b), \\ 1 + \min \begin{cases} \mathbf{lev}(\operatorname{tail}(a), b) \\ \mathbf{lev}\big(a, \operatorname{tail}(b)\big) \\ \mathbf{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) \end{cases}, & \text{otherwise.} \end{cases}$$

where $\operatorname{tail}(x)$ of any string x of length n is a substring of x without the first character, i.e. $\operatorname{tail}(x) = \operatorname{tail}(x_0x_1 \cdots x_{n-1}) = x_1x_2 \cdots x_{n-1}$ and $\operatorname{head}(x)$ of any string x of length n is a substring of x without the last character, i.e. $\operatorname{head}(x) = \operatorname{head}(x_0x_1 \cdots x_{n-1}) = x_0x_1 \cdots x_{n-2}$.

4.2. Result and Discussion

In our comprehensive analysis, we have utilized the Levenshtein Distance alongside the test script generation methodologies previously discussed to assess the output across all 51 data samples. It is crucial to highlight the exceptional stability achieved with the prompts we've designed, particularly when employing

Llama3 as our LLM. The consistency of Llama3 is noteworthy; for a given data sample, or in other words, with the same prompt, the model reliably produces identical results in each test scenario. This uniformity is a testament to the robustness of our prompt engineering and the model's ability to deliver reliable outcomes. This level of consistency is not only a significant advantage in the context of test script generation but also a key factor in ensuring the reproducibility of our experiments. It allows us to confidently attribute any variations in the output to changes in the input data or to the model's fine-tuning, rather than to the inherent instability of the model itself. By achieving such a high degree of stability, we pave the way for more accurate and meaningful evaluations of our model's performance, which in turn, informs our continuous efforts to enhance its capabilities. Moreover, this stability ensures that our test script generation process is not only efficient but also dependable, providing our partners and users with a tool that they can trust to deliver consistent results.

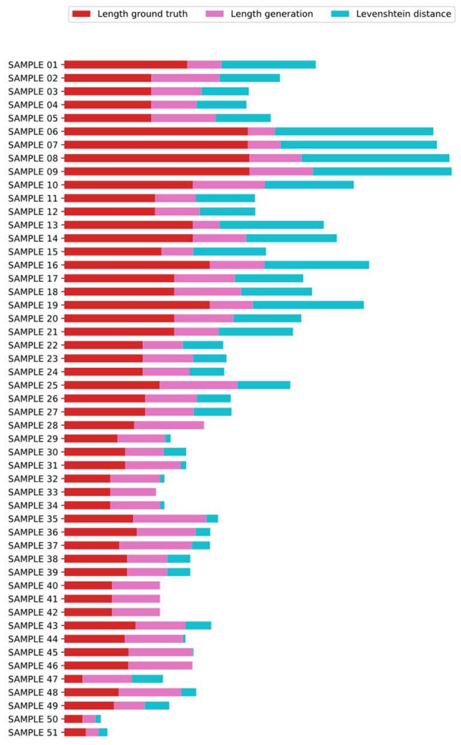

Figure 4: Discrete distribution as a horizontal bar chart to illustrate the result evaluation.

The test results for all 51 samples are displayed in the horizontal bar chart in **Figure 4**, offering a clear visual representation of our system's performance. The red bars in the chart signify the text lengths of the ground truth test scripts, serving as a benchmark for comparison. It represents the ideal output, against which the effectiveness of our system is measured. The pink bars, on the other hand, denote the lengths of the test scripts generated by our system. This provides insight into the output of our AI-driven script generation process, highlighting the efficiency and effectiveness with which our system translates prompts into executable test scripts. Most importantly, the blue bars in the chart represent the Levenshtein Distances for each sample, a critical metric that quantifies the difference between the generated test scripts and the ground truth. This distance is calculated based on the minimum number of single-character edits required to transform the generated test scripts into the ground truth test scripts. In this context, a shorter blue bar indicates a higher degree of similarity, suggesting that the generated script closely mirrors the ground truth, which is the goal of our system.

As observed from **Error! Reference source not found.**, it is evident that the system currently exhibits a noticeable margin of error. This finding is further accentuated and clarified in the subsequent statistical box plot in **Figure 5**, which provides a more detailed visualization of the distribution of errors across our dataset. It is apparent that our dataset, comprising a mere 51 samples, is significantly limited for a deep learning initiative. The consensus in the field is that a larger dataset is often necessary to train models to achieve higher accuracy and reliability.

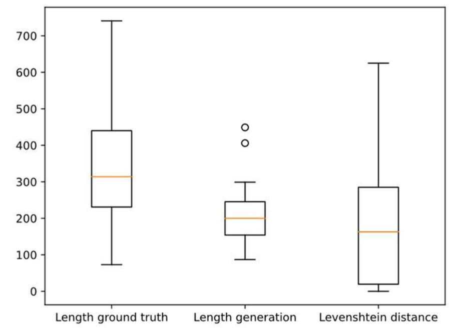

Figure 5: Box plot display of the generated test results.

However, it is remarkable to note that despite this constraint, our system has produced flawless results in six instances where the generated test scripts matched the ground truth perfectly. This achievement is particularly impressive given the small sample size and serves as a testament to the potential of our approach using prompt engineering with large language models. The fact that our system was able to generate scripts indistinguishable from the ground truth in these cases suggests that with further optimization and a more extensive dataset, we could see a substantial improvement in the system's overall performance.

This early success with a limited dataset is not just encouraging; it also validates the feasibility of our methodological approach. It indicates that our system has the innate capacity to learn and produce high-quality outputs, even when faced with data scarcity. As we continue to expand our dataset and refine our models, we are confident that the performance will see a marked enhancement, further solidifying the effectiveness of our AI-driven test script generation system in the field of software testing.

5. Conclusion and Future Work

This research highlights the significant impact of LLMs on enhancing software testing efficiency, particularly in the automotive sector. Our findings underscore the superiority of prompt engineering over model fine-tuning, especially with smaller datasets. The Levenshtein Distance proved a reliable metric for script accuracy. Notably, LLMs, such as Llama3, demonstrated remarkable consistency, indicating the robustness of our framework. Even with a limited dataset, our system achieved high accuracy, showcasing LLMs' potential in software testing.

Our study introduces a novel approach to DSL testing, with a user-friendly web application for our test script generation system, enhancing accessibility and testing efficiency. Future work includes expanding our dataset to improve script performance and integrating the system into CI/CD pipelines for real-time testing. Ethical considerations and model transparency will also be prioritized. In conclusion, our research establishes LLMs as a viable solution for automating DSL test script generation, laying the groundwork for future advancements in AI-assisted software testing.

References

[1] Awedikian, Roy, and Bernard Yannou. "Design of a Validation Test Process of an Automotive Software." International Journal on Interactive Design and Manufacturing (IJIDeM) 4, no. 4 (November 1, 2010): 259–68. https://doi.org/10.1007/s12008-010-0108-2.

[2] Campbell, Robert V. D. "Evolution of Automatic Computation." In Proceedings of the 1952 ACM National Meeting (Pittsburgh), 29–32. ACM '52. New York, NY, USA: Association for Computing Machinery, 1952. https://doi.org/10.1145/609784.609786.

[3] Orden, Alex. "Solution of Systems of Linear Inequalities on a Digital Computer." In Proceedings of the 1952 ACM National Meeting (Pittsburgh), 91–95. ACM '52. New York, NY, USA: Association for Computing Machinery, 1952. https://doi.org/10.1145/609784.609793.

[4] Demuth, Howard B., John B. Jackson, Edmund Klein, N. Metropolis, Walter Orvedahl, and James H. Richardson. "MANIAC." In Proceedings of the 1952 ACM National Meeting (Toronto), 13–16. ACM '52. New York, NY, USA: Association for Computing Machinery, 1952. https://doi.org/10.1145/800259.808982.

[5] Hetzel, William C. Program Test Methods. Prentice-Hall, 1973.

[6] Myers, Glenford J., Corey Sandler, and Tom Badgett. The Art of Software Testing. John Wiley & Sons, 2011.

[7] "IEEE Standard for Software Test Documentation." Accessed September 17, 2024. https://standards.ieee.org/ieee/829/1217/.

[8] Martin, James. Rapid Application Development. Macmillan Publishing Company, 1991.

[9] Khaliq, Zubair, Sheikh Umar Farooq, and Dawood Ashraf Khan. "Artificial Intelligence in Software Testing: Impact, Problems, Challenges and Prospect." arXiv, January 14, 2022. https://doi.org/10.48550/arXiv.2201.05371.

[10] Schäfer, Max, Sarah Nadi, Aryaz Eghbali, and Frank Tip. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation." IEEE Transactions on Software Engineering 50, no. 1 (January 2024): 85–105. https://doi.org/10.1109/TSE.2023.3334955.

[11] "Domain Specific Languages." Accessed September 17, 2024. https://martinfowler.com/books/dsl.html.

[12] Levenshtein, Vladimir I. "Двоичные Коды с Исправлением Выпадений, Вставок и Замещений Символов [Binary Codes Capable of Correcting Deletions, Insertions, and Reversals]." Soviet Physics Doklady 163, no. 4 (February 1966): 845–48.