UnitTenX: Generating Tests for Legacy Packages with AI Agents Powered by Formal Verification

Yiannis Charalambous[1,*], Claudionor N. Coelho Jr[2], Luis Lamb[3] and Lucas C. Cordeiro[1,4]

[1]The University of Manchester, UK
[2]ECE Department, Santa Clara University, US
[3]UFRGS, Brazil
[4]UFAM, Brazil

Abstract

This paper introduces *UnitTenX*, a state-of-the-art open-source AI multi-agent system designed to generate unit tests for legacy code, enhancing test coverage and critical value testing. *UnitTenX* leverages a combination of AI agents, formal methods, and Large Language Models (LLMs) to automate test generation, addressing the challenges posed by complex and legacy codebases. Despite the limitations of LLMs in bug detection, *UnitTenX* offers a robust framework for improving software reliability and maintainability. Our results demonstrate the effectiveness of this approach in generating high-quality tests and identifying potential issues. Additionally, our approach enhances the readability and documentation of legacy code.

Keywords

Artificial Intelligence, Large Language Models, Formal Methods, Software Verification, Software Engineering

1. Introduction

Software testing is an important element of the software engineering life-cycle that ensures the software is correct [1]. In the past, writing tests for software was neglected as it was not deemed as important as it is in recent times [2]. Legacy software is defined as software that uses outdated technologies with source code that is not actively maintained, but is actively used in production [3]. According to [4], legacy code may contain little or no tests. This neglect is oftentimes reflected in modern software due to deadline restrictions [5]. In many cases, this lack of tests can manifest as bugs and security vulnerabilities in legacy and newly built software [6, 7]. Contemporary examples of high-profile bugs in complex modern systems are the Heartbleed bug [8] and the CrowdStrike [9] outage. These incidents underscore the importance of thoroughly testing legacy codebases and ensuring the correctness of software systems.

For example, consider a sequence of 1000 independent and uncorrelated *if-then-else* statements, such as those found in network-based policy devices [10]; this scenario alone presents around 2^{1000} possible states to verify. Legacy code is often highly complex and lacks documentation, which makes bug detection challenging. According to [11], such conditions can result in up to 15 times more defects and extend the time required to develop new features by as much as 124%. The increase of bugs is reflected by the reduced predictability inherent in poorly maintained codebases.

To address this problem, we propose the use of AI Agents, which are software entities that have access to a limited environment where they can perform actions that change the environment autonomously [12]. In the software testing field, AI code agents can be used to automate test creation by

GeCoIn 2025: Generative Code Intelligence Workshop, co-located with the 28th European Conference on Artificial Intelligence (ECAI-2025), October 26, 2025 — Bologna, Italy
*Corresponding author.
yiannis.charalambous-4@postgrad.manchester.ac.uk (Y. Charalambous); claudionor.coelho@alumni.stanford.edu (C. N. C. Jr); lislamb@acm.org (L. Lamb); lucas.cordeiro@manchester.ac.uk (L. C. Cordeiro)
https://yiannis.site (Y. Charalambous); https://www.linkedin.com/in/claudionor-coelho-jr-b156b01/ (C. N. C. Jr); linkedin.com/in/luis-lamb-131394 (L. Lamb); https://ssvlab.github.io/lucasccordeiro/ (L. C. Cordeiro)
0009-0000-5755-5099 (Y. Charalambous); 0000-0001-9637-1890 (C. N. C. Jr); 000-0003-1571-165X (L. Lamb); 0000-0002-6235-4272 (L. C. Cordeiro)

overseeing the execution of code to detect patterns, predict defects, and optimize test cases based on software requirements and past results [13]. This auto-regressive stochastic behavior allows them to improve the accuracy and efficiency of the tests over time. The simplicity of use, efficiency, stability, and scalability make AI agents a powerful tool in software testing.

In this paper, we introduce *UnitTenX*, an AI Agent that uses formal verification to identify and create unit tests for documenting the interfaces of legacy code, thereby uncovering conditions that can cause the software to crash. By extension, the unit tests created also act as regression tests since they are designed to test the software and achieve max coverage. This paper aims to answer the following Research Questions (RQs):

1. How effectively does *UnitTenX* generate unit tests that increase code coverage for legacy C codebases?
2. How does *UnitTenX* handle compilation errors, runtime exceptions (segmentation faults), and timeouts during automated test generation?
3. How does the reflection and feedback loop in *UnitTenX* contribute to iterative improvement of generated test suites?

This paper advances the field through the following contributions:

- Introduces UnitTenX, a formal verification driven AI agent tool that automatically generates unit tests for code-bases with no tests, addressing a critical challenge in maintaining and modernizing complex software systems such as legacy software.
- Combines symbolic analysis from tools like ESBMC with large language models to identify edge cases, crash conditions, and maximize code coverage in legacy C modules.
- Automates the creation of code mockups, enabling end-to-end test suite generation and integration with existing legacy infrastructure.
- Implements a reflection and feedback loop where a language model evaluates test outcomes, recommends improvements, and iteratively increases both the quality and coverage of generated test suites.
- Demonstrates robustness by recovering from common errors such as compilation faults and segmentation violations, resulting in production-ready regression suites on real-world legacy software.

This paper is organized as follows: In Section 2 we describe the background theory that *UnitTenX* uses and also work analogous to *UnitTenX* and distinguish the areas in which it innovates. in Section 3, we present a motivating example for using *UnitTenX*, and discuss the limitations of AI in bug detection, highlighting the challenges LLMs face in identifying bugs. In Section 4, we describe *UnitTenX* itself and how it generates unit tests for legacy code. Section 5 presents the results of our study, demonstrating the effectiveness of our approach. Finally, Section 6 discusses threats to validity and concludes the paper, discussing the limitations of our approach and potential areas for future research.

2. Preliminaries

The following section covers the intersection of the theories that *UnitTenX* utilizes, including unit test generation (the core background theory), formal methods (used to identify crash states in the legacy codebase), and large language models (used as the code generation element of *UnitTenX*). Lastly, previous related works that are in the field are covered.

2.1. Regression Testing

Regression Testing is the process of testing software to detect any behavioral changes that may have appeared when the program is modified [14, 15]. This is usually done by running tests and checking if the

Program Under Test (PUT) in each test has changed, this is used to verify that changes haven't introduced new bugs or altered existing functionality [16]. Regression testing research can be split into Regression Test Selection and Optimization research [16] and regression test generation research [17, 18, 19]. The main purpose of Regression Test Selection and Optimization research is to maximize the subset of test cases affected by the code changes. This is usually done using various static and dynamic analysis techniques. The selection process is primarily guided by code coverage, to maximize coverage and fault detection while minimizing cost and execution redundancy [16]. In Regression Test Generation research, the primary purpose is to generate new test cases to test parts of the program that are not currently being tested. This is done through constraint solving at branches, using heuristics, templates, or using AI-based methods (which includes LLMs in recent years) [17, 18, 19].

2.2. Formal Verification

Formal Verification is the process of encoding a program into an abstract representation to verify that it does not violate any predefined properties. In our experiments, we use the Efficient SMT-based Bounded Model Checker (ESBMC) to identify invalid states (through counterexamples) and generate test cases for the legacy software. ESBMC is a Bounded Model Checker (BMC) that encodes the PUT into an SMT formula and uses an SMT backend to find any violated program states. As detailed in [20, 21, 22], ESBMC converts the program into a Control-Flow Graph (CFG), then extracts a state transition system $M = (S, T, s_0)$ where S is the set of states and T represents the transitions between the states $T \subseteq S \times S$ and s_0 represents the initial state of M. Let I be a predicate evaluating the set of initial states of M. Given the state transition system M, a bound k, and a property ϕ, the BMC process unrolls the system k times and translates it into a Verification Condition (VC) ψ, where ψ is satisfiable iff ϕ has a counterexample (CE) of length less than or equal to k. More formally, $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ is the executions of M of length j and the formula can be satisfied iff there exists a state at step j where ϕ is violated. The VC is calculated from the following formula:

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i) \tag{1}$$

2.3. Large Language Models

Large Language Models are deep neural networks based on the Transformer architecture [23]. The transformer architecture is composed of layers that each include a self-attention mechanism, allowing the model to relate different positions within the input sequence. This allows the model to compute relationships between all input tokens in a sequence. Self-attention enables the model to weigh the importance of different input tokens relative to each other. It does so by using $Q, K, V \in \mathbb{R}^{n \times d}$, where n is the sequence length and d is the embedding dimension. The matrices are constructed using learned linear projections of the input embeddings. In recent years, LLMs have been utilized for code generation tasks due to their ability to generate code from textual prompts with high performance.

2.4. Related Work

Automatic regression test generation now combines classic symbolic-execution techniques with emerging LLM-based methods. Traditional methods remain influential, such as TracerX, which mitigates path explosion using interpolation and lazy annotations [24], or eXpress, which applies dynamic symbolic execution to concentrate on regression-relevant paths [25]. Other symbolic execution–based tools include MutSyn for mutation-driven testing [26] and map2check, which leverages program analysis for error detection and test generation [27].

More recently, LLM-driven approaches have emerged. CoverUp integrates coverage feedback to guide test generation for Python programs [28], while SymPrompt encodes execution constraints into prompts for systematic test creation [29]. Cleverest instead focuses on zero-shot prompt generation of failing regression tests, targeting structured input programs [30].

UnitTenX approaches the problem of unit test-generation by using formal verification to extract sensitization conditions to create unit that cause crashes. *UnitTenX* is primarily aimed at legacy code bases with undocumented interfaces. By generating unit tests, we can resolve bugs and document unknown interfaces, thereby increasing their understanding.

3. Motivating Example

AI Can Fix Bugs, But Can't Find Them [31]: Finding bugs corresponds to solving the reachability problem in sequential programs, which asks if an error state is attainable from initial conditions [32]. Depending on variable domains and program complexity, this problem ranges from NP-complete implying high computational difficulty, to undecidable, where no algorithm guarantees a solution. Decoder-only LLMs generate sequences through a forward token generation process, and can at best enumerate solutions in token space, in an exponentially large search space that minimally requires backtracking [33]. This limitation highlights the challenges faced by AI in identifying bugs, as the search space for potential solutions is vast and complex.

Consider the source code in Listing 3.1, which is an extracted function from `djbdns` [10], a DNS server implementation using the C programming language. In large software codebases, it can be challenging to ensure that the code behaves as expected, especially in scenarios where it relies on results from other independent parts of the software. In the example shown, it can be hard to assert with confidence that the code cannot enter a crash state. It can be just as hard to determine if the return statements in lines 7 and 10 are the only possible way to exit the function.

Listing 3.1: Function from a DNS server implementation

```
1  int socket_recv4(int s, char *buf, int len, char ip[4], uint16 *port)
2  {
3    struct sockaddr_in sa;
4    int dummy = sizeof sa;
5    int r;
6    r = recvfrom(s, buf, len, 0, (struct sockaddr *)&sa, &dummy);
7    if (r == -1) return -1;
8    byte_copy(ip,4,(char *) &sa.sin_addr);
9    uint16_unpack_big((char *) &sa.sin_port,port);
10   return r;
11 }
```

To overcome this problem, we combine the use of LLMs with formal verification tools that can analyze the source code and provide a counterexample based on predefined properties that may have been violated [34, 35, 36]. A counterexample is a description of the state of a program that has violated a given property [37, 22]. ESBMC [1] [37], a formal verification tool that automatically encodes safety properties such as integer arithmetic errors and buffer overflows, and checks for any possible violations, can be used to identify situations that may lead to crashes or other undesirable behavior [36], which is common in legacy software. We use ESBMC [37] to analyze C code for coverage and extract sensitization conditions. This approach complements the capabilities of LLMs, providing a more formal solution for bug detection and software testing.

Legacy code bases often lack comprehensive tests, making them difficult to maintain and modernize [4]. Formal verification tools such as software model checkers can be used to find bugs and also to ensure the absence of bugs [21]. This often comes with its own set of challenges, for instance, with bounded model checkers, parameters such as the number of loop unwindings and execution timeouts must be carefully configured to balance the computational effort required to detect bugs against the necessary thoroughness and time of the search. These limitations motivate the need for automated, adaptive unit test generation systems that can efficiently improve test coverage and reliability in complex legacy systems.

[1] https://github.com/esbmc/esbmc

4. Methodology

UnitTenX operates through a series of 5 steps, which are described in this section. A visual diagram is provided for reference in Figure 1 and summarized below. The remainder of this section provides a detailed description of each step.

1. *AutoMockUps* generates mockups of each target function.
2. *Symbolic Analyzer* uses ESBMC, a formal verification tool to extract sensitization and crash conditions.
3. *Unit Test Generator* employs an LLM to produce unit tests.
4. *Coverage Analysis* compiles and reports coverage from the generated tests, using `gcov`.
5. *Reflection* uses the LLM to evaluate the results and recommends improvements to test quality and coverage.

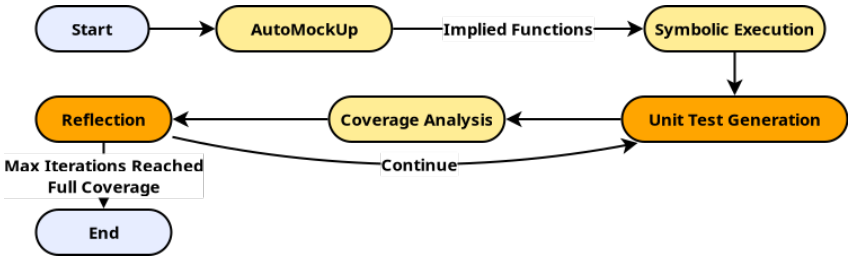

Figure 1: Shows each component of *UnitTenX* along with the flow of data. The orange nodes denote steps that use LLM to process data.

UnitTenX processes code from a single source file when creating regression tests as the whole program is included in the context of the LLM. The *AutoMockups* step automates this by constructing a single file for each function to be tested that contains the transitive closure of the symbol-dependency graph, called the *Implied Functions*. The *Implied Functions* are generated by iterating over all symbols in the source file, analyzing control-flow and dependency relationships between functions. This consolidation enables inclusion of the complete source context in LLM prompts for each target function, which is otherwise difficult because functions often depend on symbols across multiple files (Figure 2). Additionally, because the automated process to generate the mockups is reversible, tests generated using formal methods and LLMs can be correlated and annotated in the original code. By making functions and variables non-static, we increase the transparency and control over the test program, enabling more effective testing and analysis. Lastly, AutoMockups decreases long compilations when testing for coverage as it only compiles a single source file instead of multiple.

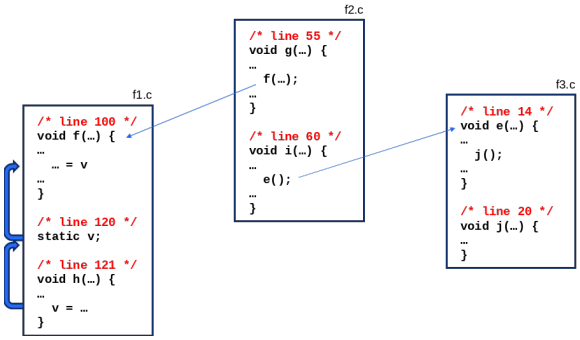

Figure 2: C Package Problems

The *Symbolic Analyzer* scans the single source file generated using AutoMockUps and extracts sensitization conditions that target coverage gaps and identifies potentially unsafe execution states.

Using formal verification, it checks for vulnerabilities such as integer overflows/underflows, buffer overflows, and other crash-inducing behaviors that ESBMC automatically encodes [36, 21]. These extracted conditions then serve as input for the *Unit-Test Generation* step.

The Unit-Test Generation step creates unit tests using the LLM. The LLM receives in its input the entire *Implied Function* C source code, previously generated tests, coverage analysis, and ESBMC outputs. The LLM then generates candidate unit tests. Any unit test that crashes on execution is commented out and annotated with `// CRASH`, creating a record of issues for developers and marking areas requiring further analysis. The unit-tests generated create a regression infrastructure, as illustrated in Figure 3. This infrastructure allows developers to track changes in the code and assess their impact on the software's behavior. Executing all generated tests can detect changes in package accesses or cross-dependencies, as illustrated in Figure 4.

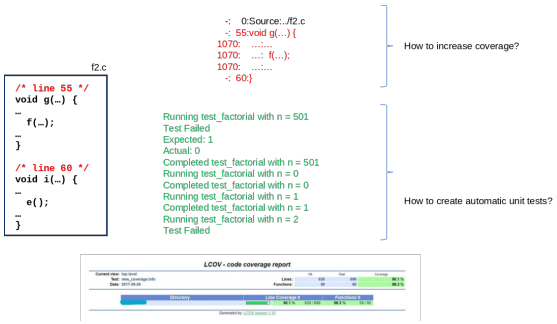

Figure 3: Tests are Regression Units

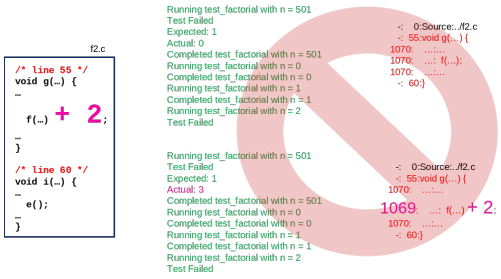

Figure 4: Regression Units Can Detect Changes

The *Coverage Analysis* step compiles and extracts coverage information from the unit tests generated. This is later used in the reflection stage to evaluate the quality of the generated test. While ESBMC counterexamples are not used directly in this step, they influence coverage indirectly by guiding the LLM during unit test generation. If ESBMC completes with a decisive result, the unit-tests generated will usually have high coverage (as seen in Section 5).

The Reflection step analyzes test and coverage results to recommend actions for improving tests. It is at this step that the test generation loop can exit or continue to keep improving the system. *UnitTenX* will exit before the Reflection step if the max number of test generation and evaluation iterations has passed over a predefined value, and there are no errors reported from previous steps. If the loop is continued, the Reflection step tasks the LLM with rating the generated test by using the coverage results and recommending a plan of action for the *Unit-Test Generation* step.

5. Experimental Evaluation

This section presents a comprehensive experimental evaluation of *UnitTenX* [2]. A record of the experimental data and results can be found at [38]. The experiments were conducted using the legacy DNS server `djbdns` [10] over 202 functions. For the experimental execution, the tests involved the execution of *UnitTenX* for each function, with automated logging to capture quantitative data for measuring code coverage, error handling, test generation, edge case analysis, and robustness. For the experiments, *UnitTenX* utilizes the `pycparser` package for parsing C source code, ESBMC `v7.7.0 64-bit x86_64` Linux for symbolic execution with Z3 `v4.13.3 64-bit` as the SMT backend. ESBMC is executed with a 10-second timeout, and GCC/clang with Gcov for instrumentation and coverage analysis. The LLM used to generate the unit tests and for review is `gpt-4o` [39]. The max iterations of the *Unit-Test Generation*, *Coverage Analysis* and *Reflection* steps was set to 4.

[2] *UnitTenX* Source Code: www.github.com/cnunescoelho/UnitTenX

5.1. RQ1: How effectively does *UnitTenX* generate unit tests that increase code coverage for legacy C codebases?

Figure 5 illustrates a scatter plot comparing the initial test quality ratings, from the initial test generation (x-axis) against the final test quality ratings (y-axis) assigned by *UnitTenX*'s reflection step. Each point represents a function, with its position indicating the initial and final subjective test quality ratings on a 0–8 scale, where points above the red dashed "No Improvement Line" signify improvement. Additionally, objective coverage metrics are derived from the dataset to assess coverage effectiveness from a baseline of 0%, as no comprehensive unit tests existed before *UnitTenX*'s intervention:

- **Coverage Effectiveness:** Of the 199 functions that executed (98.5% of 202 total), coverage was successfully measured for 186 functions (93.5%).
- **Test Quality Improvement:** Figure 5 shows that 66/199 functions (33.2%) improved their test quality ratings, moving above the diagonal. The median test quality gain was +3 points (e.g., from 0 to 5), with the largest single improvement being from 0 to 8 (+8 points), as evidenced by the point at (0,8).

These findings highlight *UnitTenX*'s dual capability: generating tests that cover previously untested code (186/199 functions) and enhancing test quality for a significant subset (33.2%). While direct coverage percentages are not plotted, the high coverage success rate and test quality improvements show that *UnitTenX* successfully generates comprehensive test suites for legacy C codebases.

5.2. RQ2: How does *UnitTenX* handle compilation errors, runtime exceptions (segmentation faults), and timeouts during automated test generation?

- **Compilation Errors:** The majority of failures occur due to code generated from the LLM that does not compile. There were 982 compilation errors. However, these were all resolved due to the iterative nature of *UnitTenX*.
- **Segmentation Faults:** There were a total of 118 unit tests generated using the symbolic analyzer that crashed and were commented out. These expose crash conditions and are useful for documentation purposes.
- **Timeouts:** 61 total timeouts occurred when ESBMC exceeded its 10 second time limit.

Despite 1161 total errors across 1167 iterations, *UnitTenX* produced compiling tests for all 199 executed functions, underscoring its robustness in recovering from intermediate setbacks.

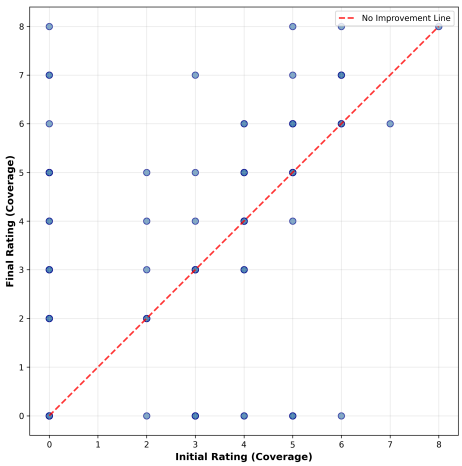

Figure 5: RQ1: *UnitTenX* Initial test coverage rating against final.

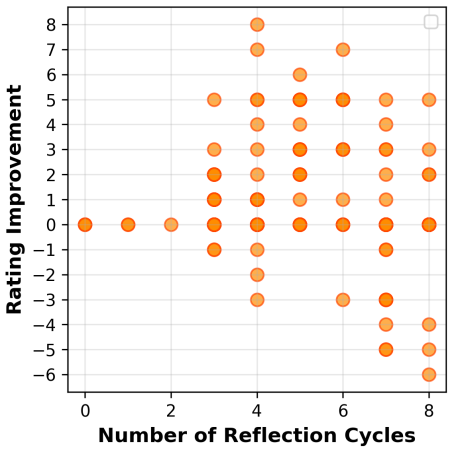

Figure 6: RQ3: Reflection & Feedback Loop Impact

5.3. RQ3: How does the reflection and feedback loop in *UnitTenX* contribute to iterative improvement of generated test suites?

Figure 6 shows a scatter plot of rating improvement (y-axis, -6 to 8) versus reflection cycles (x-axis, 0 to 8) for 199 executed functions. Each orange dot represents a function, with its position reflecting the change in test quality rating post-reflection. 66/199 functions (33.2%) showed improved ratings. Pearson correlation yields $r = 0.0859$, $p = 0.224155$ ($p \geq 0.05$), indicating no significant relationship between reflection cycles and rating improvement. Most rating improvements occurred after the first three cycles, as can be seen by the shape of the diagram.

5.4. Discussion

The findings highlight *UnitTenX*'s potential to significantly reduce the manual effort required for testing legacy codebases. In our evaluation, the codebase under test went from 0% line coverage to 100% proving that with the correct tools. A study showed that up to 50% of the development time was spent on fault localization and bug fixing [40]. By automating the generation of high-coverage test suites and effectively handling errors, *UnitTenX* addresses key challenges in maintaining and modernizing legacy software systems. The ability to generate tests that expose crash conditions also enhances the documentation and understanding of legacy interfaces, which is critical for long-term maintenance. Moreover, the system's robustness in error handling makes it a reliable tool for production environments.

6. Conclusion

Threats to Validity Several limitations should be considered when interpreting the results. First, the evaluation was conducted on a single legacy DNS server code-base. This limits the generalizability of the results. Second, the test quality ratings used in the reflection step are subjective (to the LLM's generation) and could introduce bias in assessing test quality improvement. Objective metrics, such as code coverage percentages or fault detection rates, would provide a more rigorous evaluation.

This paper introduced *UnitTenX*, a tool to generate regression tests to document legacy codebases automatically by using LLMs equipped with formal verification to extract sensitization conditions. The experimental evaluation showed that *UnitTenX* was able to bring 100% line coverage to a real-life code-base without any tests initially, proving that it is capable of being used in production environments. This makes it a viable tool for software maintenance and modernization. However, further research is needed to validate its effectiveness across diverse legacy software systems.

To address these limitations, future research should evaluate *UnitTenX* on a broader range of legacy codebases, including those from different domains and with varying levels of complexity. Integrating more objective metrics, such as coverage into the rating process in the *Reflection* step could improve its overall accuracy in assessing the generated unit-tests.

Declaration on Generative AI During the preparation of this work, the author(s) used Perplexity AI to review documents and sentences. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] S. S. R. Ahamed, Studying the feasibility and importance of software testing: An analysis, 2010. URL: https://arxiv.org/abs/1001.4193. arXiv:1001.4193.

[2] M. Felderer, G. H. Travassos, The evolution of empirical methods in software engineering, in: Contemporary Empirical Methods in Software Engineering, Springer, 2020, pp. 1–24.

[3] C. S. Rina Diane Caballar, What is legacy code?, https://www.ibm.com/think/topics/legacy-code, 2025. URL: https://www.ibm.com/, accessed: 2025-09-07.

[4] M. Feathers, Working effectively with legacy code, Prentice Hall Professional, 2004.
[5] R. Marques, G. Costa, M. M. da Silva, P. Gonçalves, A survey of failures in the software development process., in: ECIS, 2017, p. 155.
[6] D. Esther, Managing Legacy Codebases while Adopting Test- Driven Development in Continuous Integration, AAA (2024).
[7] S. Smyth, Penetration testing and legacy systems, 2023. URL: https://arxiv.org/abs/2402.10217. arXiv:2402.10217.
[8] J. Banks, The heartbleed bug: Insecurity repackaged, rebranded and resold, Crime, media, culture 11 (2015) 259–279.
[9] L. Y. Por, Z. Dai, S. J. Leem, Y. Chen, J. Yang, F. Binbeshr, K. Y. Phan, C. S. Ku, A systematic literature review on the methods and challenges in detecting zero-day attacks: Insights from the recent crowdstrike incident, IEEE Access (2024).
[10] D. J. Bernstein, abh/djbdns, 2008. URL: https://github.com/abh/djbdns, online; Accessed: 2025-07-14.
[11] A. Tornhill, M. Borg, Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases, in: Proceedings of the International Conference on Technical Debt, 2022, pp. 11–20.
[12] Z. Durante, Q. Huang, N. Wake, R. Gong, J. S. Park, B. Sarkar, R. Taori, Y. Noda, D. Terzopoulos, Y. Choi, K. Ikeuchi, H. Vo, L. Fei-Fei, J. Gao, Agent AI: Surveying the Horizons of Multimodal Interaction, 2024. URL: http://arxiv.org/abs/2401.03568. doi:10.48550/arXiv.2401.03568, arXiv:2401.03568 [cs].
[13] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, H. Cui, Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, arXiv preprint arXiv:2312.13010 (2023).
[14] B. Korel, A. M. Al-Yami, Automated regression test generation, ACM SIGSOFT Software Engineering Notes 23 (1998) 143–152.
[15] A. Ramírez, R. Feldt, J. R. Romero, A taxonomy of information attributes for test case prioritisation: Applicability, machine learning, ACM Transactions on Software Engineering and Methodology 32 (2023) 1–42.
[16] W. Wong, J. Horgan, S. London, H. Agrawal, A study of effective regression testing in practice, in: Proceedings The Eighth International Symposium on Software Reliability Engineering, 1997, pp. 264–274. doi:10.1109/ISSRE.1997.630875.
[17] A. S. Verma, A. Choudhary, S. Tiwari, Software test case generation tools and techniques: A review, International Journal of Mathematical, Engineering and Management Sciences 8 (2023) 293.
[18] H. Guan, D. Li, H. Li, M. Zhao, W. E. Wong, A review of the applications of heuristic algorithms in test case generation problem, in: 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C), IEEE, 2024, pp. 856–865.
[19] P. K. Arora, R. Bhatia, A systematic review of agent-based test case generation for regression testing, Arabian Journal for Science and Engineering 43 (2018) 447–470.
[20] F. R. Monteiro, M. R. Gadelha, L. C. Cordeiro, Model checking c++ programs, Software Testing, Verification and Reliability 32 (2022) e1793.
[21] A. Biere, M. Heule, H. van Maaren, Handbook of satisfiability, volume 185, IOS press, 2009.
[22] R. Barreto, L. Cordeiro, B. Fischer, Verifying embedded c software with timing constraints using an untimed bounded model checker, in: 2011 Brazilian Symposium on Computing System Engineering, IEEE, 2011, pp. 46–52.
[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).
[24] J. Jaffar, R. Maghareh, S. Godboley, X.-L. Ha, Tracerx: Dynamic symbolic execution with interpolation, arXiv preprint arXiv:2012.00556 (2020).
[25] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, eXpress: guided path exploration for efficient regression test generation, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 1–11. URL: https://dl.acm.org/doi/10.1145/2001420.2001422. doi:10.1145/2001420.2001422.

[26] R. Su, Z. Zhang, Y. Zhou, Y. Yao, Test generation for mutation testing by symbolic execution, in: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), IEEE, 2023, pp. 55–61.
[27] R. Menezes, H. Rocha, L. Cordeiro, R. Barreto, Map2check using llvm and klee: (competition contribution), in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2018, pp. 437–441.
[28] J. Altmayer Pizzorno, E. D. Berger, Coverup: Effective high coverage test generation for python, Proceedings of the ACM on Software Engineering 2 (2025) 2897–2919.
[29] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, B. Ray, Code-aware prompting: A study of coverage guided test generation in regression setting using llm, 2024. URL: https://arxiv.org/abs/2402.00097. arXiv:2402.00097.
[30] J. Liu, S. Lee, E. Losiouk, M. Böhme, Can llm generate regression tests for software commits?, 2025. URL: https://arxiv.org/abs/2501.11086. arXiv:2501.11086.
[31] E. David, Ai can fix bugs—but can't find them: Openai's study highlights limits of llms in software engineering, 2025. URL: https://venturebeat.com/ai/ai-can-fix-bugs-but-cant-find-them-openais-study-highlights-limits-of-llms-in-software-engineering, online; Accessed: 2025-09-27.
[32] W. Zhang, G. Wang, J. Chen, Y. Xiong, Y. Liu, L. Zhang, Ordinalfix: Fixing compilation errors via shortest-path cfl reachability, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023, pp. 1200–1211. doi:10.1109/ASE56229.2023.00072.
[33] S. Miserendino, M. Wang, T. Patwardhan, J. Heidecke, Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering?, arXiv preprint arXiv:2502.12115 (2025).
[34] O. Hasan, S. Tahar, Formal verification methods, in: Encyclopedia of Information Science and Technology, Third Edition, IGI Global Scientific Publishing, 2015, pp. 7162–7170.
[35] A. Sanghavi, What is formal verification?, EE Times_Asia (2010).
[36] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, D. A. Nicole, Esbmc 5.0: an industrial-strength c model checker, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 888–891.
[37] L. Cordeiro, B. Fischer, J. Marques-Silva, Smt-based bounded model checking for embedded ansi-c software, IEEE Transactions on Software Engineering 38 (2011) 957–974.
[38] Y. Charalambous, C. Coelho Jr, L. Lamb, L. Carvalho Cordeiro, Gecoin 2025: Unittenx: Generating tests for legacy packages with ai agents powered by formal verification, 2025. URL: https://doi.org/10.5281/zenodo.16642158. doi:10.5281/zenodo.16642158.
[39] OpenAI, Hello gpt-4o, 2025. URL: https://openai.com/index/hello-gpt-4o/, online; Accessed: 2025-09-24.
[40] M. N. Rafi, A. R. Chen, T.-H. Chen, S. Wang, Back to the future! studying data cleanness in defects4j and its impact on fault localization, arXiv preprint arXiv:2310.19139 (2023).