Proceedings of the 8th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/2026.36951

The Development of Black-Box Testing Technology and Its Application in Complex Software Systems

Yikai Zhu

Faculty of Science, Civil Aviation Flight University of China, Chengdu, China
2412211701@qq.com

Abstract. With the continuous increase in software system complexity, black-box testing, which verifies system behavior based on input-output relationships without requiring access to source code, has been widely applied to closed-source software and complex application scenarios. Based on existing studies, this paper investigates the development of black-box testing technology, including its theoretical models, testing methods, optimization strategies, and practical applications in complex software systems. With respect to testing theory, it analyzes the differences between black-box testing and white-box testing, and introduces the input-output model and fundamental testing process. At the methodological level, this paper reviews traditional test case design methods, including equivalence class partitioning, boundary value analysis, cause-effect graphing, decision tables, and scenario-based testing, and examines optimization techniques such as test case minimization and black-box fuzz testing. Regarding practical application, this paper analyzes the applicability of black-box testing methods across different system domains, using examples from industrial software, rail transit, smart home appliances, autonomous driving, artificial intelligence models, and cybersecurity protocols. The results show that artificial intelligence technology is expanding the application scope of black-box testing in requirements analysis, test case generation, and result analysis. However, complex system coverage, defect localization, and the adaptability of intelligent methods remain the main current challenges.

Keywords: Black-Box Testing, Test Case Design, Test Optimization, Fuzz Testing, Intelligent Testing

1. Introduction

The expanding use of software systems in industrial control, transportation, and artificial intelligence (AI) has raised the requirements for software quality, security, and reliability. Accordingly, software testing verifies functionality and detects defects. Among various testing approaches, black-box testing does not require access to source code or internal implementation details. Instead, it evaluates system behavior through input-output relationships defined by requirements specifications, making it suitable for closed-source software, complex industrial systems, and safety-critical applications. As software systems become larger and more complex, traditional black-box testing methods face limitations in efficiency, coverage, and automation. Research has shifted from conventional test case design to test case optimization, automated defect

DOI: 10.54254/2755-2721/2026.36951

detection, and testing strategies for complex systems. Existing methods include classical techniques such as equivalence class partitioning, boundary value analysis, cause-effect graphing, and scenario-based testing [1, 2]. They also include optimization approaches like test case minimization, fuzz testing, and intelligent testing [3-5]. These methods have been applied in industrial software, safety-critical systems, autonomous driving, and artificial intelligence applications [1, 6, 7]. This paper examines the theoretical foundations, core methods, optimization techniques, and applications of black-box testing in complex systems. Furthermore, it traces the evolution of black-box testing from traditional approaches to intelligent methods and discusses the characteristics and challenges of different testing methods across application scenarios.

2. Theoretical foundations of black-box testing

2.1. Software testing and characteristics of black-box testing

Through manual or automated approaches, software testing verifies software functions, detects defects, and evaluates software quality. Based on the use of internal program information, software testing can be classified into white-box testing, black-box testing, and gray-box testing [8]. White-box testing analyzes source code and internal logic, while gray-box testing combines partial internal information with external behavior. In contrast, black-box testing evaluates software behavior through input-output relationships defined by requirements specifications without analyzing internal implementation details [9]. Without access to source code, black-box testing is widely used in system testing, acceptance testing, and closed-source software testing. However, the lack of internal structure information limits its ability to locate defects. In contrast, white-box testing analyzes code logic and internal defects, but requires source code and design documents, thus making it less suitable for verifying whether software functions meet user requirements [8]. And the differences between black-box and white-box testing in terms of testing objectives, bases, and application scenarios are summarized in Table 1.

Table 1. Comparison of core differences between black-box testing and white-box testing

Comparison Dimension	Black-Box Testing	White-Box Testing
Testing perspective	User perspective, focusing on system functional behavior	Development perspective, focusing on internal implementation logic
Source code access	No need to access source code and internal structure	Requires access to source code and design information
Testing basis	Requirements specification document	Source code and program design documents
Primary objective	Verify whether functions meet requirements	Verify the correctness of code logic
Typical applications	System testing, acceptance testing, closed-source software testing	Unit testing, code logic testing
Advantages	Close to user needs, wide application scope	Can cover code paths, facilitate defect localization
Limitations	Weak defect localization capability	Relies on source code, high testing cost

2.2. Black-box testing model and execution mechanism

Black-box testing designs test inputs based on requirements specifications, and judges whether system behavior meets requirements by comparing actual outputs with expected results. Its core idea

DOI: 10.54254/2755-2721/2026.36951

is to regard the system under test as a mapping relationship between input and output. The testing process does not analyze the internal implementation of the program, but focuses on the external response of the system under different input conditions [9]. Black-box testing can be abstracted as an input-output mapping:

$$f : X \rightarrow Y \tag{1}$$

where X denotes the input space, including operations, data, and environment conditions, and Y refers to the output space, including responses, state changes, and exceptions. Testing selects the input x_i, and compares whether the actual output is consistent with the expected result $f(x_i)$, so as to judge whether the system behavior meets the requirements. In the test implementation process, black-box testing revolves around requirements analysis, test case design, test execution and result analysis. Based on requirements specifications, testers define constraints and generate test cases for normal, abnormal, and boundary conditions using equivalence class partitioning, boundary value analysis, and scenario-based testing. Subsequently, defects are identified by comparing actual and expected outputs, and failure conditions are recorded for debugging [10, 11].

3. Black-box testing methods and optimization techniques

3.1. Traditional black-box testing design methods

Black-box test case design determines input conditions according to requirements specifications, and judges whether system behavior meets expectations by analyzing output results. According to the input characteristics and logic complexity of test objects, typical methods like equivalence class partitioning, boundary value analysis, cause-effect graphing and decision tables, and scenario-based testing have been formed.

For functional requirements with clear input rules, equivalence class partitioning designs test cases by dividing input classes and selecting representative values, reducing test data size and applying to scenarios with clear ranges and constraints. Yet, this method mainly targets single-input conditions and provides limited coverage for multi-input scenarios. Liu et al. found that equivalence class partitioning improves testing efficiency but lacks adaptability to complex combinations [1]. For boundary defects, boundary value analysis verifies system behavior by testing maximum, minimum, and adjacent values, and applies to scenarios with clear parameter ranges. Compared with the equivalence class partitioning method, this method pays more attention to critical conditions, but it is difficult to cover multi-factor combination problems. Its effectiveness in boundary testing has been verified in previous studies [6].

For test requirements involving multi-input logic, the cause-effect graph and decision table methods generate test cases by analyzing condition combinations and output relationships, and apply to systems with complex business rules. As input conditions increase, the number of test combinations also grows, so key combinations need to be screened. Wang et al. showed that the decision table method can cover test requirements in complex logic scenarios [4]. For software systems with complex interactions, the scenario-based testing validates system behavior by simulating user activities, but its effectiveness is limited by scenario coverage. Besides, the error guessing method uses testing experience to supplement abnormal situation testing and is usually applied as a complement to other methods.

DOI: 10.54254/2755-2721/2026.36951

3.2. Black-box testing efficiency optimization techniques

With the expansion of software scale, traditional black-box testing faces problems such as increased number of test cases, rising execution costs and difficulties in test resource allocation. Therefore, relevant research mainly focuses on test case scale optimization, automated defect detection and test data configuration.

To address reduced execution efficiency caused by excessive test cases, test case minimization reduces test size while also maintaining defect detection capability by removing redundant cases. Since traditional methods usually rely on code coverage information and are difficult to directly apply to black-box scenarios without source code access, Olsthoorn et al. proposed the ATM (Adaptive Test Minimizer) method, which screens effective test case sets by analyzing test case characteristics without obtaining the source code of the system under test [12]. Experimental results show that this method can reduce the number of test cases by 40% to 60% and reduce test execution time while maintaining defect detection capability.

Furthermore, black-box fuzz testing improves defect detection efficiency in complex systems by automatically generating abnormal inputs to identify vulnerabilities and unexpected behaviors. To overcome these limitations, a BDFuzz framework based on deterministic finite automata (DFA) was proposed to generate structured test inputs through protocol state modeling [3]. Experimental results show that this approach improves protocol vulnerability detection efficiency and identifies unknown logic vulnerabilities. In terms of test resource allocation, test data optimization methods are used to analyze the relationship between test scale and defect discovery effect. The polynomial fitting method based on the least squares method determines a reasonable test case scale that meets test objectives by establishing a relationship model between the number of test cases and the defect detection rate, thereby reducing resource consumption caused by excessive testing.

4. Application and development of black-box testing in complex software systems

4.1. Black-box testing methods and their applicability

The selection of black-box testing methods needs to be adjusted in combination with the functional characteristics, input forms and verification objectives of the system under test. For software functions with clear input ranges and constraints, equivalence class partitioning and boundary value analysis can effectively reduce the scale of test data and verify parameter validity; for systems with multi-condition combinations and logical constraints, cause-effect graphing and decision tables are more suitable for analyzing behavioral relationships under different input combinations; for software with multi-function interactions, scenario-based testing verifies business process integrity. Therefore, in the actual testing process, different methods are usually combined according to system characteristics to improve test coverage.

In engineering applications, different types of software have different demands for black-box testing methods. Industrial software usually contains a large number of calculation parameters and complex input conditions, which requires combining parameter partitioning and boundary analysis to verify the calculation process; safety-critical systems pay more attention to abnormal conditions and logic failure situations, and need to adopt methods that can cover state combinations for verification; embedded software involves user operations and device interaction, which needs to focus on functional processes and scenario integrity. Previous studies have verified the applicability of combined testing methods across different software types, showing that black-box testing methods should be selected according to system characteristics [4, 6, 7].

DOI: 10.54254/2755-2721/2026.36951

4.2. Application of black-box testing in complex systems

Due to large input spaces, dynamic states, and opaque internal mechanisms, complex systems are difficult to fully evaluate through traditional source-code-based testing methods. For such systems, black-box testing verifies system functions by observing the relationship between input and output, and is more suitable for test scenarios where internal structures cannot be obtained or complete models are difficult to establish.

In autonomous driving systems, decision-making models must handle diverse environmental states, and the main testing challenge is the coverage of high-dimensional scenario spaces. In particular, the TISA test adequacy metric system evaluates testing effectiveness through traffic scenario semantics, risk boundaries, and extreme cases, providing a quantitative basis for black-box testing in autonomous driving [13]. Furthermore, boundary exploration strategies have improved high-risk scenario sampling efficiency and enhanced the detection of dangerous behaviors [5].

As test objects expand from complex systems to intelligent models and closed-source protocols, black-box testing is increasingly used to verify model output reliability and identify protocol security vulnerabilities. For intelligent models, the BTM framework derives test samples from input-output information and identifies abnormal model behaviors without requiring internal parameter access [14]. Similarly, for closed-source protocols, the BDFuzz framework generates structured test inputs through protocol state learning and improves vulnerability detection under complex protocol states [7]. These studies demonstrate that black-box testing provides effective verification approaches for systems with invisible internal structures and complex behavioral states.

4.3. Optimization and intelligent development of black-box testing

With the development of artificial intelligence technology, the research focus of black-box testing has gradually shifted from test execution automation to intelligent testing process. Traditional black-box testing mainly relies on manual completion of requirements analysis, test case design and result judgment, which is easily limited by the experience and analysis efficiency of testers in complex software systems. Methods based on machine learning and large language models can use requirement documents, historical test data and execution results to assist test decision-making, making the testing process shift from manual-driven to data-driven.

In terms of requirements understanding, large language models can extract functional constraints and business scenarios from requirements, providing support for test scope determination and test case design; in terms of test analysis, intelligent algorithms can identify abnormal patterns combined with execution results to assist defect judgment. Singh et al. showed that AI technology has been gradually applied to links such as requirements analysis, test case generation and defect analysis [2]. Alshamaa et al. further pointed out that large models can expand the application scope of black-box testing in requirements processing and result analysis, but their understanding ability for complex business logic and the reliability of test results are still insufficient, which need to be verified combined with domain knowledge [15].

5. Conclusion

This paper analyzes the development and evolution of black-box testing technology and its application in key software systems, and sorts out the basic principles, traditional test case design methods, optimization techniques and typical application scenarios of black-box testing. Studies show that black-box testing can adapt to the testing requirements of closed-source software,

DOI: 10.54254/2755-2721/2026.36951

complex industrial systems and intelligent application scenarios through input-output behavior analysis; methods such as equivalence class partitioning, boundary value analysis, cause-effect graphing and decision tables, and scenario-based testing have strong applicability in testing different types of systems, while test case optimization, fuzz testing and artificial intelligence technology further improve testing efficiency and adaptability to complex scenarios. However, black-box testing still faces limited coverage in complex systems, weak defect localization, and poor generalization of intelligent methods. Future black-box testing should integrate AI, formal analysis, and domain knowledge modeling to improve test adequacy, automation, and reliability in complex software systems, particularly for autonomous driving, artificial intelligence, and safety-critical applications.

References

[1] Liu, C. L., & Lei, H. H. (2012). Research on design methods of black-box test cases. *Modern Electronics Technique, 35*(20), 46–48. https://doi.org/10.3969/j.issn.1004-373X.2012.20.014

[2] Singh, A., & Kaur, A. (2025). A comparative analysis of AI-driven black-box testing techniques. *International Journal of Research in Engineering and Science, 13*(10), 78–86.

[3] Zhao, D. L., Gu, C. X., Zheng, Y. H., & Zhang, X. L. (2025). Logic vulnerability detection of security protocols based on black-box fuzz testing. *Journal of Information Engineering University, 26*(1), 105–112. https://doi.org/10.3969/j.issn.1671-0673.2025.01.016

[4] Wang, S., Guo, J., & Zhang, Y. D. (2019). Hazard analysis method for black-box testing of train control system safety software. *Journal of Railway Science and Engineering, 16*(3), 590–595. https://doi.org/10.19713/j.cnki.43-1423/u.2019.03.006

[5] Thompson, J. M., Goss, Q., & Akbaş, M. İ. (2023). A strategy for boundary adherence and exploration in black-box testing of autonomous vehicles. In *Proceedings of the 2023 IEEE International Conference on Artificial Intelligence Testing (AITest)* (pp. 1–8). IEEE. https://doi.org/10.1109/AITest58266.2023.00009

[6] Zhang, F., Liu, W., Guo, Y. Y., Zeng, Z. C., He, Q. W., & Zhao, Z. (2024). Application practice of black-box testing technology in fluid simulation software research and development. *Journal of Beijing University of Aeronautics and Astronautics, 50*(3), 699–708. https://doi.org/10.13700/j.bh.1001-5965.2023.0621

[7] Zhao, N. (2020). Research on black-box software testing method for smart home appliances. *China Appliance Technology*(5), 48–50.

[8] Murnane, T., Hall, R., & Reed, K. (2005). Towards describing black-box testing methods as atomic rules. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)* (pp. 41–46). IEEE. https://doi.org/10.1109/COMPSAC.2005.157

[9] Laycock, G. T. (1993). *The theory and practice of specification based software testing*. Sheffield, U.K.: University of Sheffield.

[10] Kaner, C., Bach, J., & Pettichord, B. (2002). *Lessons learned in software testing: A context-driven approach*. hOBOKEN, NJ, USA: John Wiley & Sons.

[11] I. Forgács, & A. Kovács. (2019). *Practical test design: Selection of traditional and automated test design techniques*. Swindon, U.K.: BCS Learning & Development Limited. ISBN: 9781780174723.

[12] M. Olsthoorn, S. Verwer, & A. Panichella. (2023). ATM: Black-box test case minimization based on test code similarity and evolutionary search. In *Proc. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 700–712). IEEE. doi: 10.1109/ICSE48619.2023.00066

[13] K. Gajananan, S. Soremekun, & M. Papadakis. (2024). Towards reliable AI: Adequacy metrics for ensuring the quality of system-level testing of autonomous vehicles. In *Proc. 2024 ACM International Conference on the Foundations of Software Engineering* (pp. 1789–1801). doi: 10.1145/3597503.3623314

[14] Z. Zhang, Z. Zhang, Z. Wang, *et al.* (2023). BTM: Black-box testing for DNN based on meta-learning. In *Proc. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)* (pp. 561–572). IEEE. doi: 10.1109/QRS60937.2023.00063

[15] Z. M. Alshamaa, & N. N. Saleem. (2025). Black box software testing techniques: A literature review. *Passer Journal of Basic and Applied Sciences, 7*(2), 1001–1011. doi: 10.24271/PSR.2025.508216.1963