ISSN: 2582-7219

International Journal of Multidisciplinary Research in Science, Engineering and Technology

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Impact Factor: 8.206

Volume 8, Issue 11, November 2025

© 2025 IJMRSET | Volume 8, Issue 11, November 2025| DOI:10.15680/IJMRSET.2025.0811022

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Intelligent Approaches to Testing Object-Oriented Applications

Neethu Mariam Mathew[1], Akshara Ashokan[2], Dr. Smita C Thomas[3]

PG Student [CSE], Dept. of CSE, Mount Zion College of Engineering, Kerala, India[1]

Assistant Professor, Dept. of CSE, Mount Zion College of Engineering, Kerala, India[2]

Professor, Dept. of CSE, Mount Zion College of Engineering, Kerala, India[3]

ABSTRACT: This review paper surveys modern techniques for the testing of object-oriented software. We discuss model-based testing, class and cluster-level testing, test-case generation approaches, coverage criteria adapted for OO features, and recent applications of machine learning for test selection and defect prediction. A critical study of a recent article in IEEE Computer Society, Saxena, 2025, is included to identify limitations and open problems. We identify gaps based on the survey and propose some new features along with an integrated framework which integrates OO-specific coverage criteria with ML-based prioritization and automated oracle augmentation. Target: IEEE Transactions on Software Engineering (UGC approved).

I. INTRODUCTION

In recent times, object-oriented programming has become the dominant methodology in software development because of the focus on modularity, reusability, and maintainability. However, while object-oriented design definitely improves the structure of software, it also introduces specific challenges for testing. Unlike traditional procedural systems, object-oriented software is a set of interacting objects that integrate data and behavior. The integration of data and behavior, combined with inheritance and polymorphism. Testing in the object-oriented context should, therefore, shift to accommodate these new design features. Traditional testing methods, such as functional or structural testing, cannot apply directly since they frequently neglect the dynamic nature of object interaction and reuse. For instance, the fact that a base class has been tested does not ensure that derived subclasses will work correctly because of the possibility for method overriding or extended behaviors.

The main objective of object-oriented software testing is to verify that both the individual classes at the unit level and their interactions at integration and system levels function correctly according to the design specifications. To this end, class-based testing, state-based testing, and scenario-based testing are specific strategies that have developed. In each of these methods, different aspects of OOP are considered, like ensuring data integrity, handling state transitions, and managing message passing between objects. This review aims to highlight recent research and techniques in object-oriented software testing, assessing their strengths, weaknesses, and points of improvement. It also discusses state-of-the-art technologies that could bring a drastic change in the quality and efficiency of testing object-oriented systems: machine learning, automation test frameworks, and model-based approaches.

II. BACKGROUND: OBJECT-ORIENTED TESTING CHALLENGES

Key object-oriented system features influence testing approaches are: encapsulation and information hiding obscure the internal state, making it more difficult to observe by white-box testing, inheritance and polymorphism widen the possibilities of execution paths resulting from overridden methods and dynamic dispatch, interactions among objects, sequences of method calls, give rise to state-dependent situations that are difficult to fully account for, patterns of construction and destruction, like constructors, factories, and destructors, have the possibility to introduce faults in the initialization and teardown, it requires special coverage criteria, such as class-level coverage and object coverage, and special test generation techniques.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

III. METHODOLOGY (SEARCH STRATEGY)

The search strategy for this review has been designed to comprehensively and systematically identify relevant research studies in OOST. The goal was to find quality, recent research papers, frameworks, and tools that would address a variety of techniques for testing object-oriented systems.

The following are well-known digital libraries and databases that were used for retrieving accurate and peer-reviewed information: IEEE Xplore Digital Library, ACM Digital Library, ScienceDirect (Elsevier), SpringerLink, Google Scholar. These databases were selected because they contain a wide range of high-quality journals and conference proceedings related to software engineering and testing research.

IV. MODEL-BASED TESTING APPROACHES

MBT is a systematic approach in which abstract models of the system's behavior drive the automatic generation, selection, and execution of test cases. From an OO software perspective, MBT relies on models like UML diagrams, state charts, sequence diagrams, and class diagrams to abstract the structure and behaviors of the software under test. Such models allow testers to deduce relevant test cases that reflect both functional and non-functional aspects of the system. The essential rationale for MBT is the assumption that, if the model faithfully reflects the intended functionality, then test cases derived from the model will verify whether the software indeed performs as specified. As OO systems introduce complicated interactions via classes, objects, inheritance, and polymorphism, model-based testing provides a high level of abstraction and automation required to manage testing complexity. For example, state-based models are capable of capturing dynamic object behavior, transitions, and event-driven interactions. Sequence diagrams can be adopted to ensure the correctness of object collaborations, message passing, and method invocations; class diagrams are often employed for deriving test cases that verify inheritance hierarchies, encapsulation, and associations among classes.

The MBT process typically encompasses four main phases: model creation, where system behavior is modeled in formal or semi-formal notations; test case generation, where algorithms generate test cases automatically from the model; test execution, where the generated test cases are applied to the actual implementation; and result analysis, where discrepancies between expected and actual outcomes are identified to detect defects. Tools like Spec Explorer, Conformiq Designer, and UML Testing Profile-based frameworks are widely used to support MBT in the context of OO software. Early defect detection is one significant benefit that comes with model-based testing: Models can be analyzed and validated even before the implementation of code is started, reducing the cost of finding errors later in development. Additionally, MBT supports test automation and traceability since models can be linked to requirements and code artefacts directly. However, the methodology also entails a number of challenges, such as the need for precise and maintainable models, issues of compatibility between tools, and the difficulty in dealing with non-deterministic behaviors that are common in OO systems. Recent research has also focused on combining MBT with machine learning and search-based testing techniques in order to optimize test case selection and coverage. Hybrid models that integrate MBT with traditional testing methods provide improved fault detection efficiency and scalability for large OO systems. Overall, model-based testing represents a powerful and evolving paradigm for improving the reliability, efficiency, and maintainability of object-oriented software testing.

V. CLASS-LEVEL, CLUSTER-LEVEL AND INTEGRATION TESTING

Testing is a very important phase in software development, where a system is checked for correctness, reliability, and performance before its deployment. Testing in object-oriented software engineering is done at various abstraction levels, starting from class-level testing, cluster-level testing, and ending with integration testing. These stages are hierarchical, with each successive level verifying the correctness of broader system interactions. At the class level, each class is tested as an independent unit to verify internal logic, attributes, and methods. A class represents a self-contained unit encapsulating both data and behaviour; hence, testing at this level shall focus on its functionalities in isolation. The aim will be to ensure that each method behaves as specified, constructors initialize data correctly, and class invariants hold across operations. Mock objects and stubs are used to simulate dependencies, thus allowing the class to be tested independently. Class-level testing also covers object interactions like inheritance and polymorphism, making sure that overridden and inherited methods behave correctly. Examples of tools that may be used during the automation of class-level testing include JUnit, NUnit, and PyUnit. This stage forms the basis for higher testing levels through the early identification of defects in the smallest components of the system. Once the individual classes have been verified, testing

moves on to the cluster level, where a group of interrelated classes is tested together. A cluster represents, usually, a logical subsystem or a set of classes that cooperatively perform some function. The main objective here is to confirm that these related classes interact among themselves properly and that data exchange among them, along with message passing, takes place correctly. In an online shopping application, for example, classes like Customer, Cart, and Payment can form a cluster. Testing this cluster helps to ensure that customer details are being correctly linked with the contents of the cart and payment processing. This stage of testing uncovers interface mismatches, invalid data sharing, and logical inconsistencies not readily observable at the individual class level. Cluster-level testing may employ incremental strategies such as top-down or bottom-up integration to gradually combine related classes and validate their cooperative behaviour.

Integration testing verifies the correctness of the interactions between larger software modules or subsystems. Once clusters or components are developed and tested, they must be integrated into a complete system. Integration testing ensures that these combined components communicate correctly through defined interfaces and achieve the overall functionality of the system. Errors at this stage often arise due to incompatible data formats, incorrect API usage, timing issues, or logical inconsistencies across module boundaries. There are several approaches to integration testing, including top-down, bottom-up, sandwich (hybrid), and big-bang. Top-down integration testing starts with the testing of high-level modules and then progressively adds lower-level modules using stubs. Bottom-up integration testing starts with lower-level modules, integrating upwards by using drivers. The sandwich approach is a hybrid of the previous two, while the big-bang method integrates all the modules simultaneously but makes debugging harder.

In summary, class-level, cluster-level, and integration testing represent a progressive validation process in software development. Class-level tests the individual building blocks, cluster-level testing ensures the correct collaboration of related classes, and integration testing validates the interaction of complete modules within the system. Together, these levels provide a comprehensive framework for identifying defects, improving software quality, and ensuring that the system performs reliably as a unified whole. Effective implementation of these testing levels not only enhances robustness and maintainability but also contributes to reducing the long-term cost of development and maintenance.

VI. TEST CASE GENERATION TECHNIQUES

Test case generation techniques are important for comprehensive and systematic testing of object-oriented software systems. The focus of these techniques is to generate efficient test cases automatically or semi-automatically to find defects and verify the functional correctness, reliability, and performance of the software. Since OO software exhibits encapsulation, inheritance, polymorphism, and dynamic binding, the generation of relevant test cases requires particular strategies that take into account object interactions, method dependencies, and class hierarchies. In general, test case generation techniques for OO systems can be categorized into specification-based, structure-based, and model-based approaches. Specification-based techniques derive test cases from the formal or informal software specifications, such as use case diagrams, class contracts, or pre/post-conditions of methods. These tests ensure that the software meets its functional requirements and user expectations. Structure-based techniques, also known as white-box techniques, focus on the internal structure of the code, including control flow graphs, data flow, and method coverage criteria. They are useful for detecting logical and structural errors in the implementation. Model-based techniques generate test cases from high-level design models, thus allowing early validation and automation in the testing process.

Among the common test case generation methods, path testing guarantees coverage of all the possible execution paths in the code at least once, while data flow testing looks at the usage and definition of variables to find anomalies. State-based testing is quite applicable to OO systems because it deals with transitions in object states and events that lead to those transitions. Other important approaches include mutation testing, which makes small changes to the code on purpose to assess the power and capability of existing test cases. If the test cases cannot find these mutations, then they are considered weak and need enhancement. Recent progress has also introduced search-based and evolutionary algorithms like genetic algorithms, simulated annealing, and particle swarm optimization to optimize the generation of test cases. These find the most efficient set of test cases that maximizes code coverage or fault detection probability with least redundancy. Furthermore, AI-driven and machine learning-based methods are gaining attention for predicting fault-prone areas and the generation of context-aware test cases dynamically. Automation tools such as JUnit, EvoSuite, and TestNG support the automation of test case generation and execution in OO environments. They improve repeatability, reduce manual effort, and ensure consistency during regression testing. However, how to handle test case explosion, handle dynamic polymorphic behavior, and maintain test cases for an evolving system are still active research areas. In summary, test

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

case generation techniques are central in achieving effective and efficient testing of object-oriented software. Employing specification-based, structure-based, and model-based methods supported by automation and optimization approaches will definitely lead to much improved quality, reliability, and maintainability of OO software systems.

VII. COVERAGE CRITERIA FOR OO SYSTEMS

Coverage criteria in OO systems define the extent at which a test suite exercises a program's various structural and behavioral elements. OO systems introduce some testing issues that are not found in procedural systems due to concepts like encapsulation, inheritance, polymorphism, and dynamic binding. Consequently, applying only traditional code coverage techniques-statements, branches, and paths-is not enough. Specialized coverage criteria are needed to determine whether adequate testing has been done concerning OO features and interaction among objects. In OO systems, the coverage criteria are usually distinguished into method-level, class-level, and inter-class (integration) coverage. The method level typically employs criteria like statement coverage, decision coverage, and condition coverage to make sure that every executable statement and logical branch within individual methods is covered. This helps in verifying the correctness of the smallest functional units of the system. However, OO software goes beyond simple method execution. It involves interactions between multiple cooperating objects/classes, and such interactions do require higher-level coverage metrics.

The coverage criteria at the class level focus on the internal structures of classes and the relationships among class members. This includes attribute coverage, which ensures that all data members are accessed or modified; method coverage, which ensures that all public and private methods are invoked at least once; and constructor/destructor coverage, which validates the proper creation and destruction of objects. Since classes may inherit attributes and methods from parent classes, inheritance coverage ensures all inherited features are tested in both their original and overridden contexts. Polymorphic coverage is also extremely important-it checks whether methods invoked through dynamic dispatch are executed in all of their possible forms. Consider, for example, that a base class reference points to derived class objects; in such a case, each possible implementation of the overridden method should be tried. At the integration or inter-class level of coverage, test coverage focuses on exercising interactions between objects with their collaborations in the system. One of the main criteria in this area is message coverage ensuring that all possible messages between objects, in the form of method calls, are exercised. Association coverage checks all links between classes participating in associations, aggregations, or compositions. State-based coverage ensures that all object states and state transitions are tested, which is particularly important for systems modeled using state diagrams. This helps verify that objects behave correctly under all valid sequences of operations.

Another important dimension of testing in OO is interaction coverage, whereby combinations of collaborating objects and their method calls are tested adequately. Given the possibility of multiple sequences and contexts in which objects can interact, complete interaction coverage is usually difficult to obtain. Model-based testing, UML sequence diagram analysis, data flow testing between objects, and other approaches serve as tools and techniques to measure this aspect of coverage. Modern software engineering practices use code coverage tools and automated testing frameworks like JaCoCo, EMMA, and Clover, which support OO-specific coverage analysis. The tools provide information on what part of the system is left untested and help the developer to design additional test cases targeting uncovered elements. Keeping a high degree of coverage does not guarantee there will be no defects, but it greatly increases confidence that the system is correct. In a nutshell, the coverage criteria for OO systems are essential in assessing the adequacy of testing to ensure that both individual and interactive behaviors of objects are verified. By applying the coverage metrics at multiple levels-method, class, and inter-class-developers can achieve comprehensive testing of OO software. The structured approach enhances reliability and reduces maintenance costs by ensuring the behavior of the system conforms to design specifications and real-world requirements.

VIII. CASE STUDY: ANALYSIS OF A RECENT IEEE ARTICLE (SAXENA, 2025)

Selected article: Saxena, A. (2025). 'Rethinking Software Testing for Modern Development' (IEEE Computer Society article). This recent article reviews the shift to automated testing and the integration of AI/ML. The main contributions are the landscape of AI tools for testing and recommendations for industrial adoption.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Key observations by Saxena, 2025, are: the adoption of AI-assisted testing by industry is accelerating, with academic-industrial transfer occurring inconsistently, most ml models are trained on limited datasets with limited generalizability, test oracles and specification mining remain open problems, especially for OO systems with complex states.

IX. LIMITATIONS IN EXISTING MODELS

From the reviewed literature and the Saxena 2025 article, identify these recurring limitations here.

1. Insufficient representation of OO-specific features for ML models; features usually reduce classes to flat metrics like LOC or CBO.
2. Oracle problem: several methods assume the existence of oracles, or depend on weak heuristics.
3. Scalability: Symbolic and model-based techniques do not scale well on large codebases.
4. Dynamic behaviour: handling of runtime polymorphism and reflective calls is still weakly addressed.
5. Availability of dataset and benchmarking: Lack of standardized OO testing benchmark with ground-truth faults.

X. PROPOSED ENHANCEMENTS AND NEW FEATURES

1. OO-aware feature sets for ML: graph-based features using class-dependency graphs, call-site graphs and object-state transition encodings - e.g., graph embedding with GNNs.
2. Automated oracle augmentation: apply specification mining, assertion mining, and differential testing to generate stronger oracles for OO behaviors.
3. Hybrid test generation pipeline that combines model-based test path generation from UML/state diagrams with search-based optimization to prioritize high-value paths.
4. Scalable symbolic-console hybrid exploration with selective concretization for object inputs and lazy initialization strategies.
5. Public benchmark suite: curated OO systems (small-to-large) with seeded faults and rich metadata for reproducible evaluation.

XI. PROPOSED INTEGRATED FRAMEWORK

Stage 1: Model extraction (UML diagrams, class-call graphs) and object-state automata derivation.
Stage 2: The generation of graph embedding using GNNs to obtain ML features representing OO structure.
Stage 3: Test-path generation using MBT augmented with genetic search, prioritizing using the ML-predicted fault scores.
Stage 4: Oracle augmentation via mined assertions and differential oracles (compare behavior across versions or implementations).
Stage 5: CI integration that provides a continuous evaluation-feedback loop where test outcomes refine ML models.
This framework aims to improve the fault-detection rates while reducing test suites and addressing OO-specific challenges.

XII. CONCLUSION

This review synthesized recent advances in OO testing, highlighted key limitations, and proposed OO-TestML — an integrated framework that uses graph representations, ML-based prioritization, and oracle augmentation to address existing gaps. The next steps are implementation, evaluation on benchmarks, and submission to a UGC-approved venue such as IEEE Transactions on Software Engineering. In addition to these directions, future research should also focus on enhancing test automation and scalability for large, evolving object-oriented systems. As software complexity grows, there is an increasing need for adaptive testing approaches capable of handling continuous integration and deployment environments. The integration of artificial intelligence and deep learning with traditional testing techniques can significantly improve defect prediction accuracy, dynamic test case generation, and automated regression testing. Another critical area for future work is test oracle improvement, where automated oracles can intelligently evaluate output correctness using semantic analysis and natural language processing of specifications. Furthermore, the adoption of hybrid testing frameworks that combine model-based, search-based, and machine learning approaches will ensure higher test coverage, reduced human intervention, and faster validation cycles. The collaboration between academia and industry can also accelerate the practical adoption of such frameworks, leading to more robust and maintainable OO systems.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Overall, the advancement of intelligent, model-driven, and learning-enabled testing techniques represents a major step toward the automation of quality assurance in object-oriented software engineering. By bridging theoretical innovation with practical evaluation, OO-TestML and related frameworks have the potential to redefine how testing is performed in modern, adaptive software ecosystems.

REFERENCES

1. Saxena, A. (2025). Rethinking Software Testing for Modern Development. IEEE Computer Society, online article. [source: turn0search21]
2. Labiche, Y. (2000). Testing levels for object-oriented software. ACM. [source: turn2search2]
3. Orso, A., & Pezze, M. (2002). Integration testing of object-oriented software. IEEE Proceedings. [source: turn2search3]
4. Pusarla, S., Peruri, G.S., Yalavarthi, M. (2024). An empirically based object-oriented testing using Machine learning. EAI Endorsed Trans IoT. (2024). [source: turn3search5]
5. Santelices, R., Jones, J.A., Yu, Y., & Harrold, M.J. (2009). Lightweight fault-localization using multiple coverage types. ICSE/IEEE. [source: turn3search17]
6. Wong, W.E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8), 707–740. [source: turn3search11]

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com