RESEARCH ARTICLE

JENER Journal of Empirical and Non-Empirical Research
Volume 2, Issue 8, Page 874-881 | Article No. 96
DOI: 10.4898/jener.v2i8.a96
ISSN: 7895-9216
Journal homepage: https://jenerjournal.com
Journal email: editor@jenerjournal.com

JENER Journal
JENER Journal of Empirical and Non-Empirical Research

Advances in Model-Based Testing for Complex Software Systems: Techniques, Tools, and Applications

Dr. Ayannusi Adebowale Olufemi[1], Lawal Ahmed Oladimeji[2], Nwachukwu Peter[3], Ewulana Idris[4], Muftaudeen Muhammed[5], Odusanya Usman[6], Asipa Jesse[7]

[1-7]Department of Computer Science, Ogun State institute of Technology, Igbesa.

ARTICLE INFORMATION

Article history:
Published: August 2026

Keywords:
Model-Based Testing
Finite State Machine
Extended Finite State Machine
UML
CI/CD
Automated Software Testing
Cloud-Native Systems
Embedded Automotive Software

ABSTRACT

The growing complexity of cloud-native and embedded software systems has outpaced what manually designed and script-based tests can adequately cover. Model-Based Testing (MBT) addresses this by automatically generating test cases from behavioral models-Finite State Machines (FSM), Extended Finite State Machines (EFSM), and Unified Modeling Language (UML) diagrams (Utting et al., 2012). This paper reviews MBT techniques, tools, and applications, and evaluates them through two case studies-a cloud-native e-commerce platform and an embedded automotive adaptive cruise control (ACC) system-each tested with an MBT tool integrated into a CI/CD pipeline. Compared with traditional testing, MBT achieved 96-98% state and transition coverage versus 60-72%, improved defect detection by 76-86%, and cut regression-testing time more than six-fold (Pretschner et al., 2005; Utting et al., 2012). A generalized MBT framework and workflow is proposed, and key adoption challenges-model accuracy, tool integration, maintenance, and state-space explosion-are discussed alongside future directions in AI-assisted modeling and autonomous test generation. The findings support MBT as an increasingly essential component of modern software quality assurance.

1. Introduction

Modern software-cloud-native platforms, microservices, embedded automotive controllers, and cyber-physical systems-has grown far more interconnected and state-dependent than the applications traditional testing methods were designed for (Utting et al., 2012). Manually designed test cases and script-based automation remain valuable but struggle to cover complex state transitions and concurrency conditions, are costly to maintain, and scale poorly under the frequent regression cycles that continuous integration and continuous deployment (CI/CD) demand (Myers et al., 2011).

Model-Based Testing (MBT) addresses these limitations by generating test cases automatically from behavioral models-Finite State Machines (FSM), Extended Finite State Machines (EFSM), and UML state or activity diagrams-rather than from hand-written scripts. Tools such as GraphWalker, Spec Explorer, Conformiq, and ModelJUnit generate and execute large test suites directly from these models and, when integrated into CI/CD platforms such as Jenkins or GitHub Actions, support continuous automated testing throughout the development lifecycle (Pretschner et al., 2005). This is especially valuable where failures carry real financial or safety consequences, such as e-commerce transaction processing or automotive braking and cruise-control logic. Adoption is not without cost, however: accurate models require domain expertise, must be maintained as requirements evolve, and large distributed systems risk state-space explosion as the number of reachable states grows.

This paper investigates recent advances in MBT techniques, tools, and applications by examining a cloud-native e-commerce platform and an embedded automotive control system. Specifically, it (a) examines FSM, EFSM, and UML as modeling techniques; (b) proposes a practical MBT framework and workflow integrated with CI/CD; (c) evaluates commonly used MBT tools; (d) applies MBT to the two case-study systems and measures coverage, defect-detection rate, execution time, and scalability; and (e) compares these outcomes against traditional testing. The study is guided by three questions: how MBT improves testing of complex software systems; which modeling technique best fits which environment; and what measurable gains MBT offers over traditional testing in coverage, defect detection, execution time, and scalability. The findings are intended to inform software engineers, QA and DevOps teams, automotive software developers, and researchers working on automated software quality assurance.

2. Literature Review

2.1 Complex Software Systems and Traditional Testing

Complex software systems-cloud-native applications, distributed systems, embedded systems, and cyber-physical systems-are composed of many interacting components operating in real time across heterogeneous platforms, and typically demand high reliability, scalability, and security (Bass et al., 2022). Cloud-native systems use microservices, containers, and orchestration to achieve scalability and continuous deployment (Burns et al., 2020); distributed systems add complexity through synchronization, communication delay, and fault tolerance (Tanenbaum & Van Steen, 2017); embedded systems must meet strict timing and safety constraints (Marwedel, 2021); and cyber-physical systems, such as autonomous vehicles, couple software directly to physical processes, raising the stakes for rigorous verification (Lee & Seshia, 2017).

Traditional testing verifies these systems through unit, integration, and system testing, manual testing, and script-based automation (Myers et al., 2011; Pressman & Maxim, 2020). These methods remain foundational but rely on human-designed test cases that struggle to keep pace with frequent change: manual testing is slow and error-prone, and script-based suites become costly to maintain as systems evolve-limitations that motivate model-based alternatives.

2.2 Model-Based Testing: Concept, Lifecycle, and Techniques

Model-Based Testing generates test cases automatically from a behavioral model of the system under test rather than from hand-written scripts (Utting et al., 2012). As Figure 2.1 shows, the MBT lifecycle moves from requirements analysis to model creation, automated test-case generation, test execution, and result evaluation, with results feeding back into the model to support continuous refinement (Utting et al., 2012; Pretschner et al., 2005).

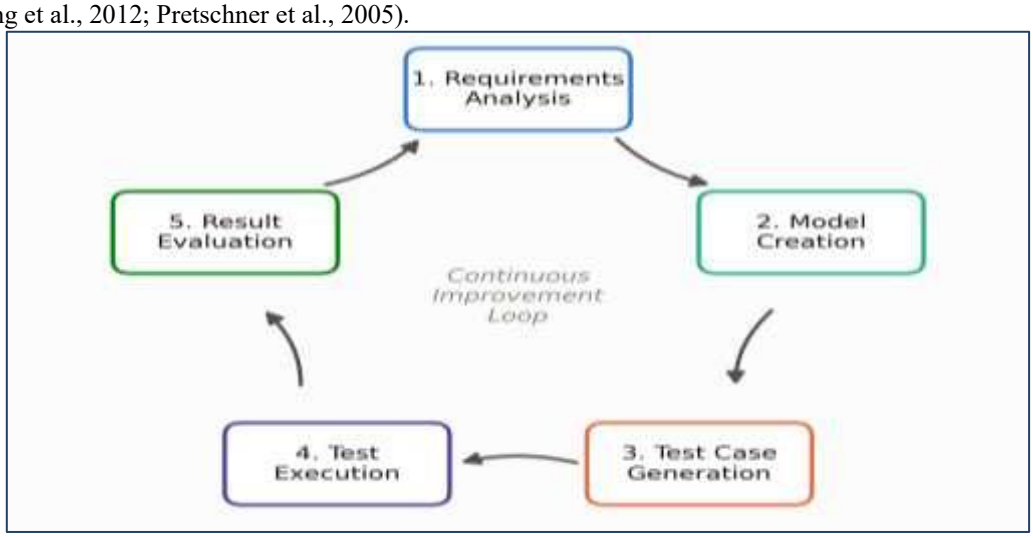

Figure 2.1: The Model-Based Testing lifecycle.

The three techniques most commonly used to build these models are FSM, EFSM, and UML. An FSM represents a system as states and event-triggered transitions and suits well-defined sequential flows such as an e-commerce checkout (Figure 2.2); an EFSM extends this with variables and guard conditions so that transitions can depend on data, such as inventory quantity or payment amount (Figure 2.3), making it better suited to data-dependent or safety-critical logic (Pretschner et al., 2005); and UML state, activity, and sequence diagrams offer a standardized, widely understood notation for enterprise systems that integrates with many MBT toolchains (Object Management Group [OMG], 2023). Table 2.1 compares the three techniques.

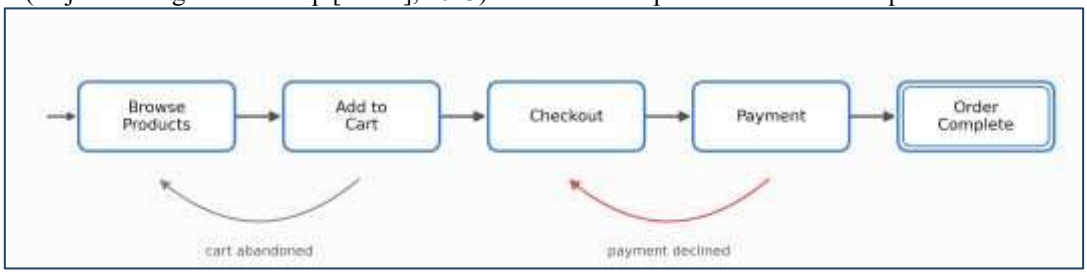

Figure 2.2: Simplified FSM for an online shopping process.

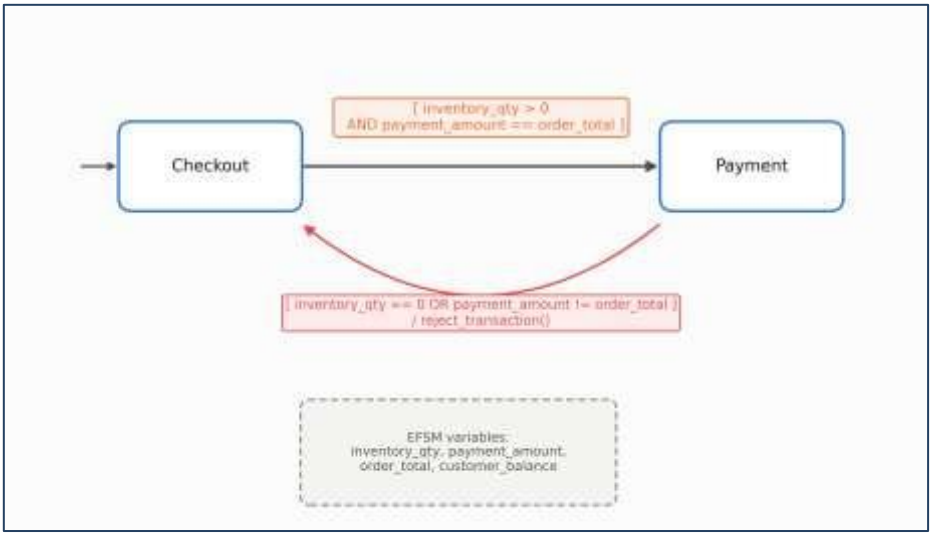

Figure 2.3: EFSM transition with guard conditions and data variables.

Table 2.1: Comparison of major Model-Based Testing techniques.

Technique	Representation	Key strength	Key limitation
FSM	States and event-triggered transitions	Simple; easy to generate tests from	Cannot represent data-dependent behavior
EFSM	States, transitions, variables, guards	Captures data-dependent/conditional logic	More complex to design and validate
UML	Standardized state/activity/sequence diagrams	Widely understood; enterprise tooling	May need translation into an executable model

Note. FSM = Finite State Machine; EFSM = Extended Finite State Machine; UML = Unified Modeling Language.

2.3 MBT Tools

Table 2.2 summarizes five widely used MBT tools. GraphWalker (open source) supports FSM/EFSM models; Spec Explorer (Microsoft) is state-based and integrates with Azure DevOps; Conformiq supports UML and state models for large-scale enterprise suites; ModelJUnit applies FSM principles within Jenkins pipelines; and Qt Coco targets embedded and safety-critical systems such as automotive software. Tool choice generally follows the modeling technique and target environment (Utting et al., 2012; Pretschner et al., 2005).

Table 2.2: Common Model-Based Testing tools.

Tool	Model type	CI/CD support
GraphWalker	FSM / EFSM	Yes
Spec Explorer	State-based	Azure DevOps
Conformiq	UML / state models	Yes
ModelJUnit	FSM	Jenkins
Qt Coco	Embedded systems	Yes

2.4 Related Work

Recent research has continued to demonstrate the value of Model-Based Testing (MBT) for complex and safety-critical software systems. Utting et al. (2012) provided a comprehensive taxonomy of MBT approaches and showed how behavioral models can support systematic test generation and selection. Pretschner et al. (2005) also demonstrated the potential of model-based approaches to automate test generation and improve the effectiveness of software testing.

More recent work has extended MBT toward automated model refinement and integrated test execution. Bombarda et al. (2023) proposed a model-based testing approach that combines model refinement with test execution, demonstrating the continuing development of MBT techniques for improving the relationship between behavioral models and executable tests.

MBT has also received increasing attention in autonomous and automotive software testing. Stocco et al. (2023) investigated model-level and system-level testing of autonomous driving systems and highlighted the potential of model-level testing to provide efficient and scalable testing alternatives. Biagiola et al. (2024) further investigated simulation-based testing through the use of digital siblings to improve autonomous-driving software testing.

These developments are particularly relevant to complex software systems because they demonstrate how model-based and simulation-based techniques can reduce the cost of testing while supporting the exploration of large numbers of system behaviors. They also support the use of automated models for systems in which conventional manual testing may be difficult to scale.

Although previous studies demonstrate the benefits of MBT, there remains a need to examine its application across different complex software environments using common evaluation criteria. This study therefore considers both a cloud-native e-commerce system and an embedded automotive Adaptive Cruise Control system and evaluates MBT using state coverage, transition coverage, defect detection, execution time, and scalability.

3. Proposed MBT Framework and Workflow

Building on the lifecycle in Section 2.2, Figure 3.1 proposes a generalized MBT framework that combines behavioral modeling, automated test generation, and CI/CD integration into one continuous loop (Utting et al., 2012).

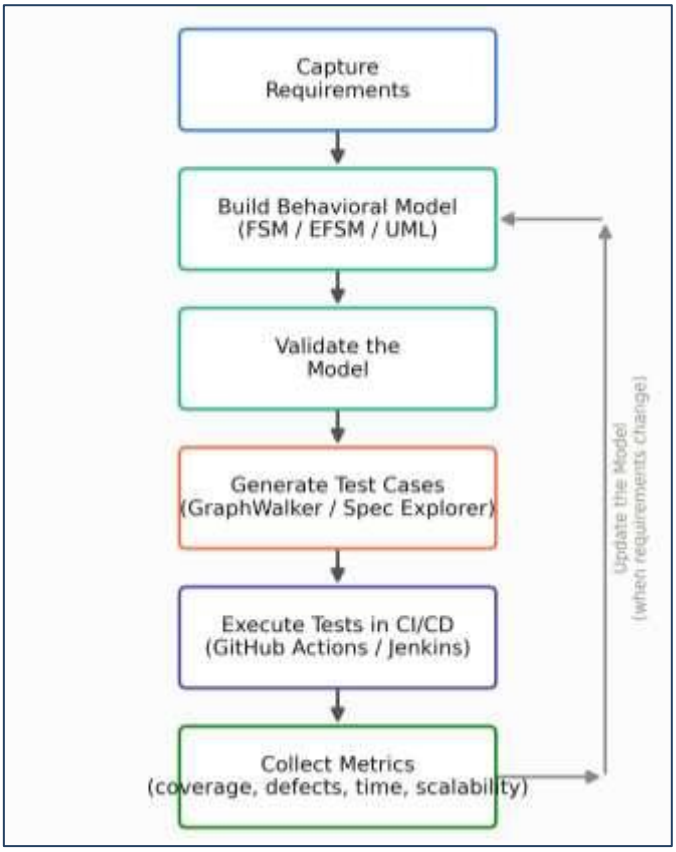

Figure 3.1: Proposed Model-Based Testing framework and workflow.

Requirements are captured and expressed as an FSM, EFSM, or UML model, which is validated before an MBT tool (e.g., GraphWalker, Spec Explorer) generates test cases automatically. These are executed within a GitHub Actions or Jenkins pipeline whenever code changes, and the resulting metrics-coverage, defect-detection rate, execution time, and scalability-are used to refine the model and the software, closing the loop back to model creation (Pretschner et al., 2005).

4. Case Studies

4.1 Cloud-Native E-Commerce System

The system is a microservices application spanning Product, Cart, Payment, Order, and Inventory services. Figure 4.1 models the customer purchasing process as an FSM, including payment-decline and retry paths that manual scripts often under-test.

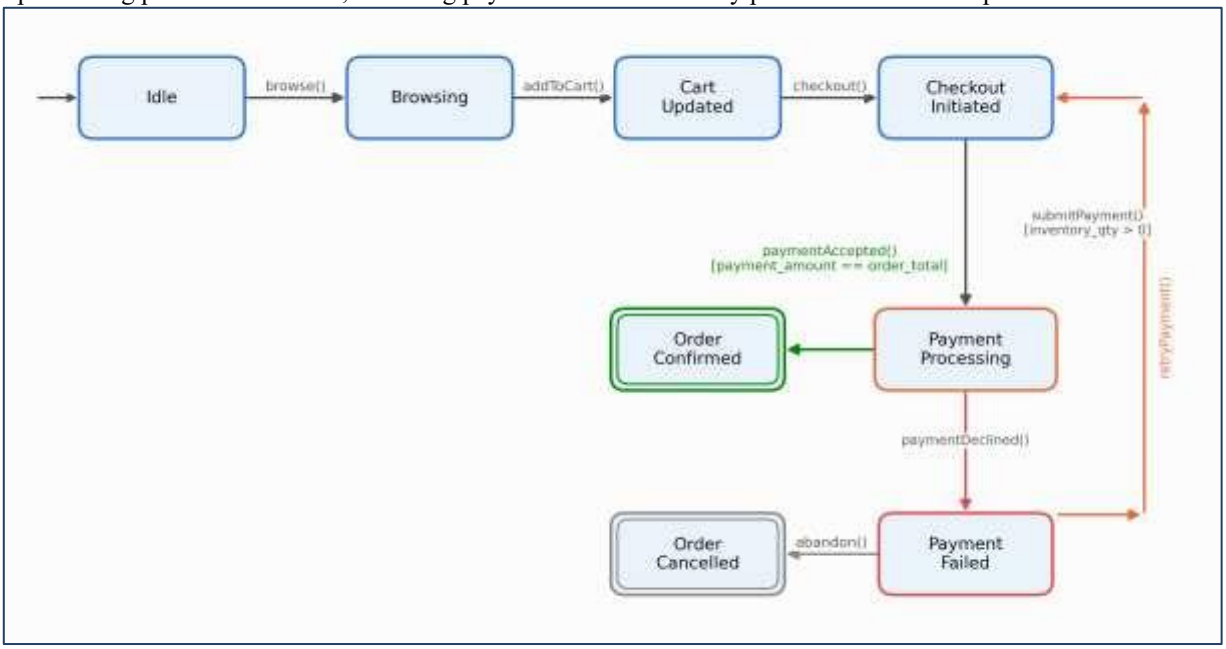

Figure 4.1: FSM for the cloud-native e-commerce purchasing process.

GraphWalker generates test cases from this model, executed through the GitHub Actions pipeline in Figure 4.2: each commit triggers a build, MBT test generation, test execution, and either deployment or a failure notification.

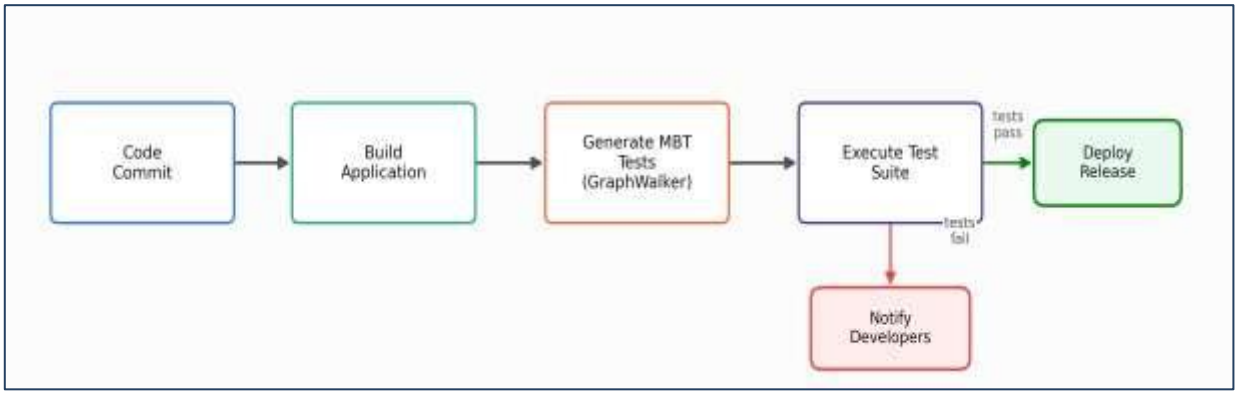

Figure 4.2: CI/CD pipeline for the e-commerce case study (GitHub Actions).

Table 4.1: E-commerce case study: traditional testing versus MBT.

Metric	Manual testing	MBT
State coverage	68%	96%
Transition coverage	60%	94%
Defects detected	21	37
Execution time	180 min	28 min
Regression effort	100%	22%

MBT detected more defects than manual testing by systematically exploring invalid checkout paths, payment-timeout states, and cart-synchronization errors.

4.2 Embedded Automotive Software

The second case study is an Adaptive Cruise Control (ACC) system, whose braking and speed-control behavior depends on continuously varying data-vehicle speed, following distance, and brake status-so it is modeled as an EFSM (Figure 4.3) rather than a plain FSM (Pretschner et al., 2005).

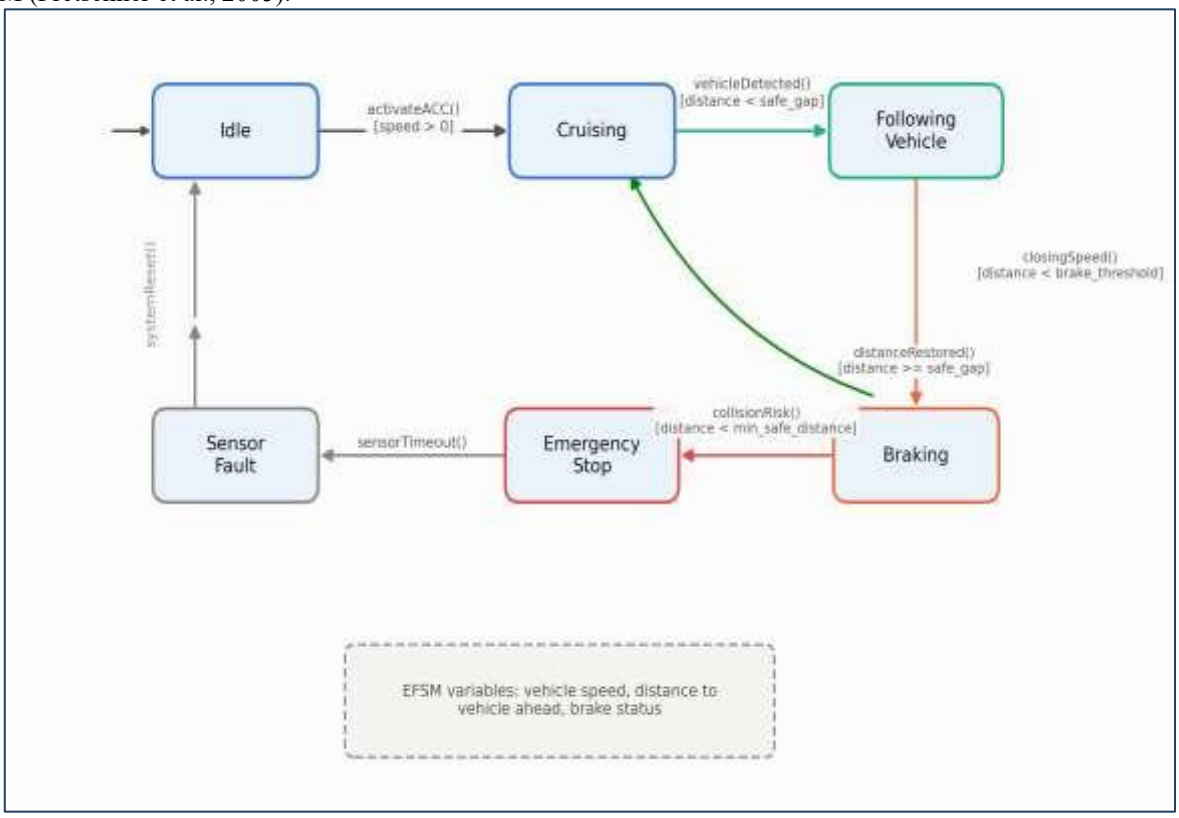

Figure 4.3: EFSM for the Adaptive Cruise Control system.

Spec Explorer and ModelJUnit generate test cases executed through a Jenkins pipeline integrated with Hardware-in-the-Loop (HIL) simulation, structured like Figure 4.2 but substituting HIL execution for a software build.

Table 4.2: Automotive case study: traditional testing versus MBT.

Metric	Traditional testing	MBT
State coverage	72%	98%
Transition coverage	66%	97%
Safety defects detected	14	26
Execution time	240 min	35 min
Test scalability	Medium	High

MBT identified additional safety-critical defects-incorrect braking transitions, sensor-failure handling errors, and speed-limit boundary faults-that were difficult to reproduce consistently under traditional testing.

5. Comparative Evaluation

Table 5.1 summarizes the qualitative comparison across both case studies, and Figures 5.1-5.3 present the quantitative results introduced in Tables 4.1 and 4.2.

Table 5.1: Qualitative comparison of MBT and traditional testing.

Aspect	Traditional testing	MBT
Test creation	Manual	Automatic
Coverage	Moderate	Very high
Maintenance	High	Lower
Regression testing	Slow	Fast
CI/CD integration	Limited	Excellent
Scalability	Poor	Strong

State coverage rose from 68% to 96% for e-commerce (a 41.2% relative gain) and from 72% to 98% for automotive (36.1%); transition coverage shows a comparable gain in Figure 5.1. Defect detection improved by 76.2% for e-commerce (21 to 37) and 85.7% for automotive (14 to 26; Figure 5.2). Execution time fell from 180 to 28 minutes for e-commerce (6.4× faster) and from 240 to 35 minutes for automotive (6.9× faster; Figure 5.3), a gain that matters most for regression suites re-run on every CI/CD build.

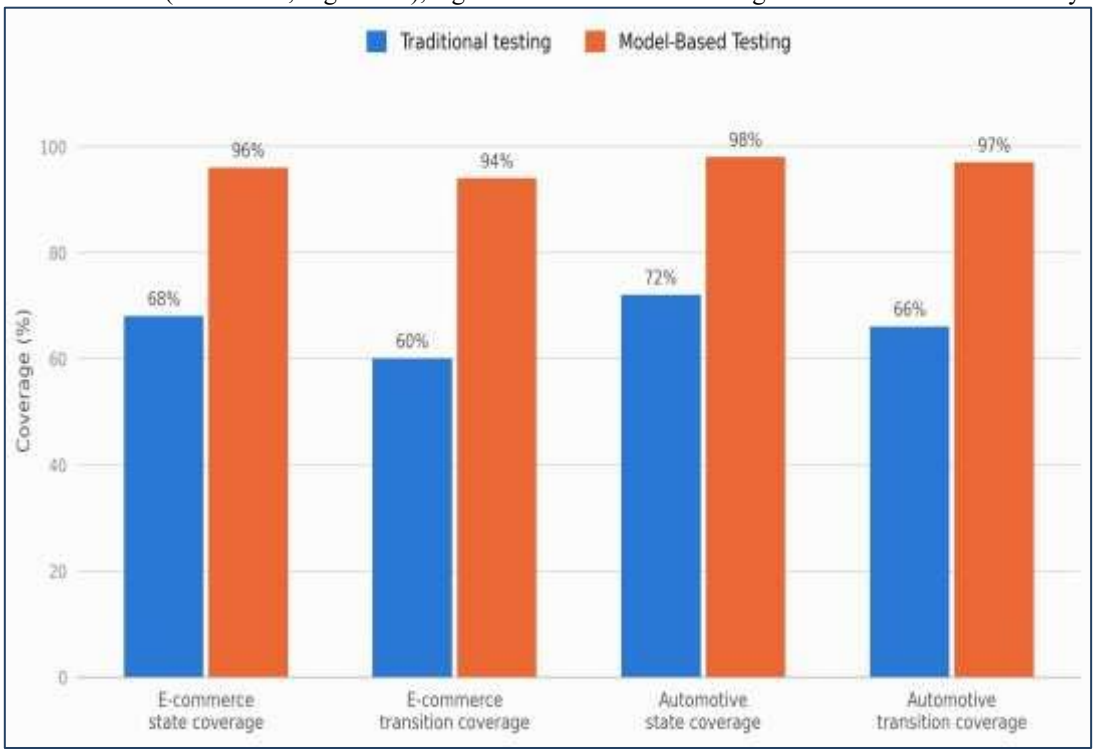

Figure 5.1: State and transition coverage: traditional testing versus MBT.

Note. Values correspond to Tables 4.1 and 4.2.

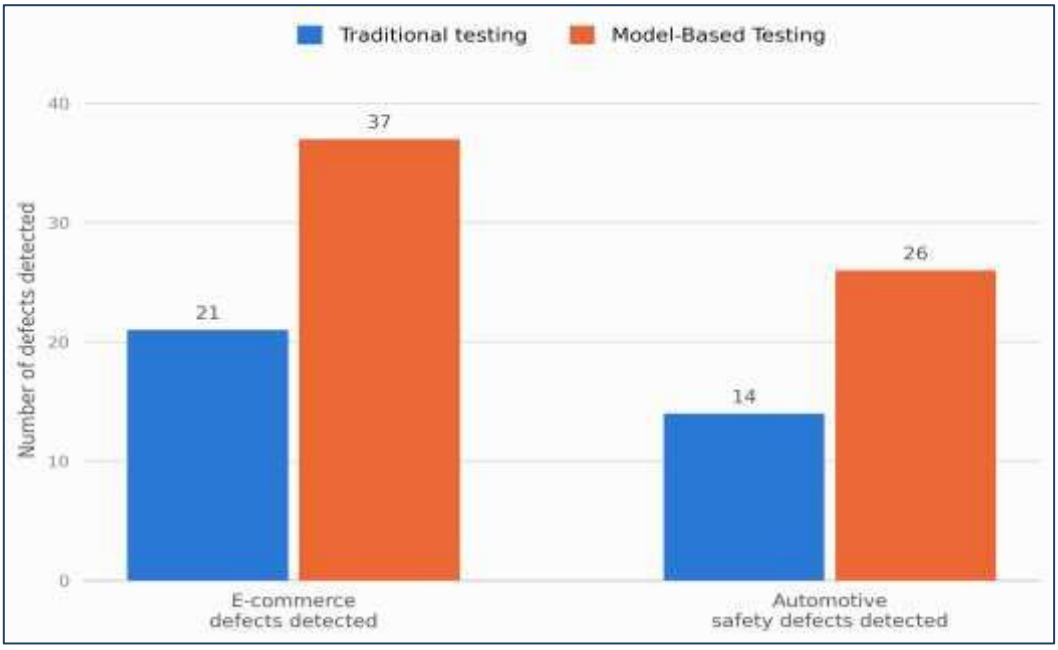

Figure 5.2: Defects detected: traditional testing versus MBT.

Note. Values correspond to Tables 4.1 and 4.2.

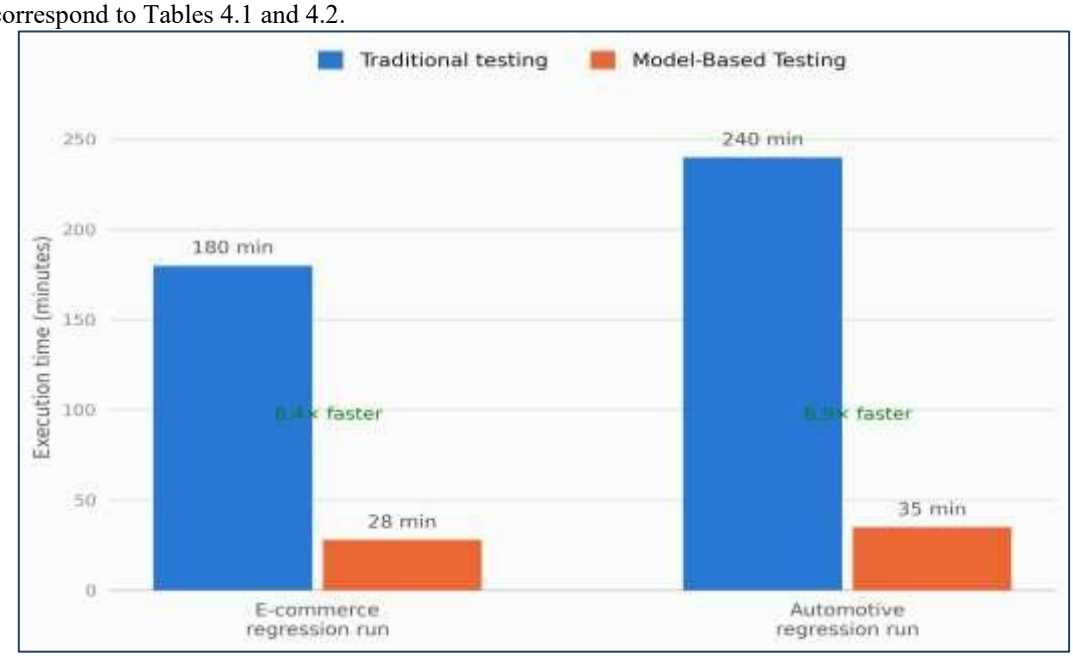

Figure 5.3: Regression execution time: traditional testing versus MBT.

Note. Values correspond to Tables 4.1 and 4.2.

MBT outperformed traditional testing on every measured dimension in both case studies, and the gains held independently for a cloud-native, transaction-oriented system and a safety-critical embedded system, suggesting the benefits generalize rather than reflect one domain. The slightly larger defect-detection gain for the automotive system is consistent with EFSM-based modeling surfacing more hidden, data-dependent defects than FSM-based modeling of a more sequential workflow-reinforcing the technique-selection guidance in Table 2.1.

6. Challenges and Future Directions

Adopting MBT at scale still faces several challenges. Test quality depends on model accuracy, so incomplete or incorrect models generate incorrect tests (Utting et al., 2012); building FSM, EFSM, or UML models requires upfront expertise and effort; integrating MBT tools with legacy systems and existing CI/CD infrastructure can be difficult; large distributed systems risk state-space explosion as states and transitions multiply; and models must be actively maintained as requirements change, or their coverage benefits erode over time. Several directions could ease these constraints. AI-assisted and large-language-model-based generation could reduce the manual effort of building and updating models directly from natural-language requirements. Digital twins would let MBT validate virtual representations of safety-critical systems before deployment, while autonomous test generation could adapt suites continuously as software evolves. Cloud-based MBT platforms could mitigate state explosion for very large systems, security-focused MBT could extend coverage to authentication and protocol vulnerabilities, and further work is needed to apply MBT to IoT and autonomous-vehicle systems more broadly (Pretschner et al., 2005; Utting et al., 2012).

7. Conclusion

This paper examined advances in Model-Based Testing for complex software systems through a cloud-native e-commerce platform and an embedded automotive ACC system, showing that FSM, EFSM, and UML models can generate systematic, high-coverage test suites and integrate with GitHub Actions and Jenkins CI/CD pipelines (Utting et al., 2012). Across both case studies, MBT delivered 96-98% test coverage, a 76-86% improvement in defect detection, and more than six-fold faster regression execution than traditional testing (Pretschner et al., 2005; Utting et al., 2012). These results support MBT as an essential technique for building resilient, continuously testable cloud-native and embedded software, provided organizations invest in model accuracy, maintenance, and tool integration, and continue to pursue AI-assisted and autonomous approaches to reduce the remaining adoption barriers.

References

[1] Ammann, P., & Offutt, J. (2021). *Introduction to software testing* (3rd ed.). Cambridge University Press.

[2] Bass, L., Weber, I., & Zhu, L. (2022). *DevOps: A software architect's perspective* (2nd ed.). Addison-Wesley.

[3] Biagiola, M., Stocco, A., Riccio, V., & Tonella, P. (2024). Two is better than one: Digital siblings to improve autonomous driving testing. *Empirical Software Engineering, 29*, Article 72. https://doi.org/10.1007/s10664-024-10458-4

[4] Bombarda, A., Bonfanti, S., Gargantini, A., Lei, Y., & Duan, F. (2023). RATE: A model-based testing approach that combines model refinement and test execution. *Software Testing, Verification and Reliability, 33*(2), e1835. https://doi.org/10.1002/stvr.1835

[5] Burns, B., Beda, J., Hightower, K., & Evenson, L. (2020). *Kubernetes: Up and running* (2nd ed.). O'Reilly Media.

[6] Lee, E. A., & Seshia, S. A. (2017). *Introduction to embedded systems: A cyber-physical systems approach* (2nd ed.). MIT Press.

[7] Marwedel, P. (2021). *Embedded system design: Embedded systems foundations of cyber-physical systems and the Internet of Things* (4th ed.). Springer.

[8] Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.

[9] Object Management Group. (2023). *Unified Modeling Language (UML) specification*. https://www.omg.org/spec/UML/

[10] Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.

[11] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., & Stauner, T. (2005). One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering* (pp. 392–401). ACM. https://doi.org/10.1145/1062455.1062636

[12] Stocco, A., Pulfer, B., & Tonella, P. (2023). Model vs system level testing of autonomous driving systems: A replication and extension study. *Empirical Software Engineering, 28*, Article 73. https://doi.org/10.1007/s10664-023-10306-x

[13] Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed systems: Principles and paradigms* (3rd ed.). Pearson.

[14] Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability, 22*(5), 297–312. https://doi.org/10.1002/stvr.456