Static and Runtime Verification of Auto-Generated Application Logic in Low-Code Systems
Düşük Kodlu Sistemlerde Otomatik Olarak Oluşturulan Uygulama Mantığının Statik ve Çalışma Zamanı Doğrulaması

Jaswanth Kumar Mandapatti[1], Nareshkumar Jagadhabi[2], Vishnu Vardhan Reddy Kavuluri[3], Maheswara Rao Gorumutchu[4], Srinivasarao Bandla[5]
[1]Advent Health, United States, Email: jash.209@gmail.com
[2]Compnova Inc, United States, Email: nrkumar544@gmail.com
[3]4 Consulting INC, United States, Email: vishnu.kavuluri@gmail.com
[4]HYR Global Source Inc, United States, Email: gmrmails@gmail.com
[5]Deloitte Consulting LLP, United States, Email: Bandla.srinivas10@gmail.com

***Abstract*— Low-code platforms increasingly rely on auto-generated application logic to accelerate enterprise software development, but this abstraction introduces risks related to hidden defects, inconsistent rule execution, and lack of formal validation. Existing approaches primarily focus on development efficiency while offering limited support for rigorous verification of generated logic, particularly in complex and dynamic execution environments. This study presents a unified verification framework that integrates static analysis and runtime validation to ensure correctness and reliability of auto-generated logic in low-code systems. The framework formalizes generated workflows, applies constraint-based static verification for detecting structural inconsistencies, and employs runtime monitoring to capture execution anomalies and context-dependent deviations. Evaluation across varying logic complexity levels demonstrates that the combined approach improves defect detection accuracy, enhances rule coverage, and maintains acceptable validation latency compared to standalone verification methods. The results highlight the complementary strengths of static and runtime techniques in addressing both pre-execution and dynamic validation challenges. The proposed framework provides a scalable and robust foundation for improving reliability, compliance, and correctness in enterprise low-code applications, while enabling future integration of automated formal verification and intelligent validation mechanisms.**

Keywords*— low-code platforms, static verification, runtime verification, auto-generated logic, rule validation, software correctness, enterprise systems, anomaly detection.

***Özetçe*— Düşük kodlu platformlar, kurumsal yazılım geliştirmeyi hızlandırmak için giderek daha fazla otomatik olarak oluşturulan uygulama mantığına güvenmektedir; ancak bu soyutlama, gizli kusurlar, tutarsız kural yürütme ve biçimsel doğrulama eksikliği ile ilgili riskler ortaya çıkarmaktadır. Mevcut yaklaşımlar öncelikle geliştirme verimliliğine odaklanırken, özellikle karmaşık ve dinamik yürütme ortamlarında, oluşturulan mantığın titiz bir şekilde doğrulanması için sınırlı destek sunmaktadır. Bu çalışma, düşük kodlu sistemlerde otomatik olarak oluşturulan mantığın doğruluğunu ve güvenilirliğini sağlamak için statik analiz ve çalışma zamanı doğrulamasını entegre eden birleşik bir doğrulama çerçevesi sunmaktadır. Çerçeve, oluşturulan iş akışlarını biçimlendirir, yapısal tutarsızlıkları tespit etmek için kısıtlamaya dayalı statik doğrulama uygular ve yürütme anormalliklerini ve bağlama bağlı sapmaları yakalamak için çalışma zamanı izleme kullanır. Değişken mantık karmaşıklığı seviyelerinde yapılan değerlendirme, birleşik yaklaşımın, bağımsız doğrulama yöntemlerine kıyasla kusur tespit doğruluğunu artırdığını, kural kapsamını geliştirdiğini ve kabul edilebilir doğrulama gecikmesini koruduğunu göstermektedir. Sonuçlar, hem yürütme öncesi hem de dinamik doğrulama zorluklarını ele almada statik ve çalışma zamanı tekniklerinin tamamlayıcı güçlü yönlerini vurgulamaktadır. Önerilen çerçeve, kurumsal düşük kodlu uygulamalarda güvenilirliği, uyumluluğu ve doğruluğu artırmak için ölçeklenebilir ve sağlam bir temel sağlarken, gelecekte otomatikleştirilmiş biçimsel doğrulama ve akıllı doğrulama mekanizmalarının entegrasyonunu da mümkün kılmaktadır.**

Anahtar Kelimeler*— Düşük kodlu platformlar, statik doğrulama, çalışma zamanı doğrulaması, otomatik oluşturulan mantık, kural doğrulaması, yazılım doğruluğu, kurumsal sistemler, anormallik tespiti.

I. Introduction

The rapid adoption of low-code platforms has significantly transformed enterprise software development by enabling automatic generation of application logic from high-level models and configurations. These platforms reduce development effort and accelerate deployment cycles, allowing organizations to respond quickly to changing business requirements. However, as auto-generated logic becomes increasingly central to enterprise systems, concerns have emerged regarding the correctness and reliability of such generated artifacts. Recent studies highlight that abstraction-driven development often obscures underlying execution semantics [1]. This makes it difficult to verify system behavior rigorously in large-scale enterprise deployments.

Auto-generated application logic in low-code environments is typically derived from visual workflows, declarative rules, and metadata-driven configurations. While this approach enhances usability and accessibility, it introduces challenges related to hidden dependencies and implicit control flows. The absence of explicit code structures can lead to logical inconsistencies that are not easily detectable during development. Model-driven approaches have been shown to introduce execution ambiguity when abstractions are layered over complex workflows [2]. This lack of transparency complicates validation and debugging processes in enterprise systems.

One of the most critical risks associated with auto-generated logic is the presence of latent defects that remain undetected until runtime. These defects may arise from incorrect rule mappings, conflicting conditions, or incomplete state transitions within generated workflows. In enterprise environments, such defects can have severe consequences, particularly in financial systems, healthcare applications, and regulatory compliance scenarios. Studies on software reliability demonstrate that undetected logical errors can propagate across interconnected workflows [3]. This amplification effect can lead to cascading failures across dependent systems.

In addition to hidden defects, inconsistent rule execution poses a significant challenge in low-code systems. Since execution logic is often generated dynamically based on user-defined configurations, variations in runtime conditions can result in unpredictable behavior. This issue becomes more critical in distributed environments, where concurrent execution and asynchronous triggers introduce further variability. Prior research on rule-based execution models indicates that inconsistencies in rule evaluation can directly affect system correctness and decision outcomes [4]. Such inconsistencies are particularly problematic in environments requiring strict operational guarantees.

The lack of formal validation mechanisms further exacerbates these challenges in low-code ecosystems. Traditional software engineering practices rely on rigorous testing, code reviews, and formal verification techniques to ensure correctness, but these approaches are not fully integrated into low-code platforms. Research in model-driven engineering suggests that without formal verification frameworks, the confidence in system correctness significantly decreases [5]. This limitation becomes more pronounced as application complexity and abstraction depth increase.

To address these limitations, verification must be treated as a fundamental component of low-code system design rather than an auxiliary process. Static verification techniques enable analysis of generated logic prior to execution, identifying structural inconsistencies, unreachable states, and rule conflicts. At the same time, runtime verification provides continuous monitoring of system behavior, ensuring that execution adheres to defined constraints during operation. Studies in software verification indicate that combining static and runtime approaches enhances defect detection and system robustness [6]. This integrated approach provides a more comprehensive validation strategy.

Despite the recognized importance of verification, existing approaches remain fragmented and insufficiently integrated within low-code ecosystems. Most available solutions focus either on static analysis or runtime monitoring, without offering a unified framework that effectively addresses both dimensions. Furthermore, adapting traditional verification techniques to auto-generated logic introduces additional challenges due to abstraction layers and dynamic execution paths. These limitations have been highlighted in recent studies that emphasize the need for specialized verification models tailored to low-code environments [7]. Additional investigations also point to scalability and integration constraints in current verification pipelines [8].

This study addresses the identified gap by proposing a combined static and runtime verification framework for auto-generated application logic in low-code systems. The approach focuses on formalizing generated logic, detecting inconsistencies at design time, and validating execution behavior during runtime. By embedding verification mechanisms within the development lifecycle, the framework aims to enhance reliability, reduce defect propagation, and ensure consistent execution of enterprise application logic.

II. Methodology

The proposed methodology introduces a unified verification framework that integrates static analysis and runtime validation to ensure correctness of auto-generated application logic in low-code systems. The framework is designed to operate across the full lifecycle of application execution, beginning from logic generation and extending to runtime monitoring. By combining pre-execution verification with continuous validation during execution, the approach addresses both structural and behavioral inconsistencies that may arise in abstraction-driven development environments.

The first component of the framework focuses on the formal representation of auto-generated logic. Application workflows and rule definitions are transformed into a structured intermediate representation defined as a directed state transition system $G=(S,T,\delta)$, where Sdenotes system states, Trepresents transitions derived from generated rules, and δdefines state evolution functions. This formalization enables systematic analysis of execution paths and provides a foundation for detecting inconsistencies such as unreachable states and cyclic dependencies.

Static verification is performed on the formalized logic prior to execution. The process includes rule consistency checks, dependency resolution, and constraint validation. Each rule is analyzed using a constraint function $C_i(S)\rightarrow\{0,1\}$, which determines whether the rule satisfies predefined logical and domain-specific constraints. Conflict detection mechanisms are employed to identify overlapping conditions, contradictory rules, and redundant transitions. This phase ensures that the generated logic adheres to structural correctness before deployment.

A key aspect of the methodology is the rule extraction mechanism, which converts low-code configurations, visual workflows, and declarative expressions into analyzable rule sets. This extraction process maps high-level constructs into formal logical expressions, preserving execution semantics

while enabling verification. The extracted rules are stored in a rule graph structure that captures dependencies and execution order, facilitating both static and dynamic analysis.

Runtime verification complements static analysis by monitoring execution behavior during application operation. A runtime observer module is introduced to track state transitions and validate them against predefined constraints. For each executed transition T_i, a runtime validation function $R(T_i,S_t)$evaluates whether the observed behavior aligns with expected outcomes. Deviations are flagged as anomalies, enabling immediate detection of execution inconsistencies that may not be identifiable during static analysis.

To ensure comprehensive validation, the framework incorporates constraint enforcement mechanisms at runtime. These mechanisms include temporal constraints, state invariants, and rule execution boundaries. Temporal constraints verify correct sequencing of events, while invariants ensure that critical system properties remain consistent throughout execution. By enforcing these constraints dynamically, the framework provides an additional layer of assurance against runtime violations in auto-generated logic.

The experimental setup is designed to evaluate the effectiveness of the verification framework under varying levels of logic complexity. Synthetic application logic is generated to simulate real-world low-code scenarios, including multi-branch workflows, nested rule conditions, and concurrent execution paths. Logic complexity is parameterized based on the number of rules, depth of dependency chains, and degree of conditional branching, enabling controlled experimentation across different system configurations.

A verification pipeline is implemented to process generated logic through static and runtime validation stages. The pipeline begins with rule extraction, followed by static verification, deployment, and runtime monitoring. Performance metrics are collected at each stage, including defect detection rate, rule coverage, validation latency, and anomaly detection accuracy. These metrics provide quantitative insights into the effectiveness of the verification approach.

Finally, the evaluation criteria are defined to compare static and runtime verification performance across increasing logic complexity. Detection accuracy is measured as the proportion of identified defects relative to total injected inconsistencies, while rule coverage represents the percentage of rules successfully analyzed and validated. Validation latency captures the time overhead introduced by verification processes. The comparative performance of static and runtime approaches across these metrics is illustrated in Figure 1, highlighting their complementary strengths in ensuring correctness of auto-generated application logic.

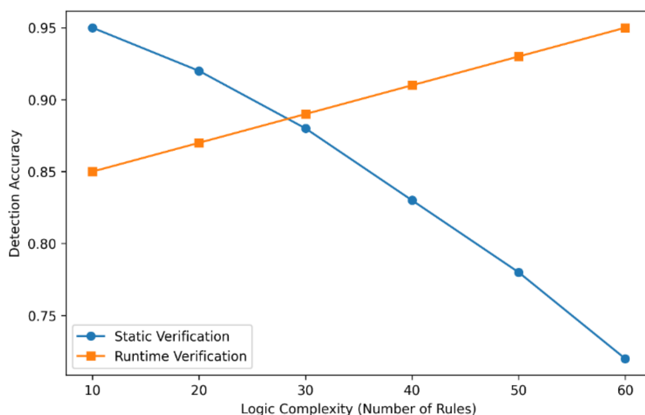

Figure 1. Detection Accuracy of Static vs Runtime Verification Across Increasing Logic Complexity

III. Results and Discussion

The proposed verification framework was evaluated across multiple auto-generated logic scenarios to assess its ability to detect structural and behavioral inconsistencies. The results indicate that both static and runtime verification approaches contribute significantly to identifying defects such as unreachable states, rule conflicts, and logical inconsistencies. Static verification proves highly effective in detecting structural issues prior to execution, particularly in identifying redundant rules and invalid dependency chains. Runtime verification, on the other hand, demonstrates stronger capability in capturing execution anomalies that emerge only during dynamic system behavior.

A detailed analysis of defect detection capability shows that static verification achieves high accuracy in identifying rule-level inconsistencies and unreachable execution paths. However, its effectiveness decreases as logic complexity increases, particularly in cases involving dynamic conditions and runtime-dependent variables. In contrast, runtime verification maintains consistent detection performance across increasing complexity levels, as it evaluates actual execution traces rather than relying solely on pre-defined rule structures. This complementary behavior highlights the importance of combining both approaches within a unified verification framework.

The framework's ability to detect rule conflicts was also evaluated under varying execution conditions. Static analysis successfully identifies overlapping rule conditions and contradictory logic definitions before deployment. However, certain context-dependent conflicts, especially those influenced by runtime data variations, are only detected through runtime monitoring. This distinction emphasizes the limitations of purely static approaches and reinforces the need for continuous validation during execution to ensure comprehensive defect coverage.

Overhead and performance implications were examined to understand the trade-offs associated with verification. Static verification introduces minimal runtime overhead, as most computations are performed prior to deployment. Runtime verification, while more computationally intensive, provides real-time assurance of execution correctness. Despite the additional overhead, the observed latency remains within acceptable limits for enterprise applications, particularly when optimized monitoring strategies are employed. The balance between verification depth and system performance is therefore a critical consideration in practical implementations.

A comparative evaluation of performance metrics is presented in Table 1, which summarizes detection rate, false positives, validation latency, and rule coverage for both approaches. The results indicate that static verification achieves higher precision with lower false positive rates in structural analysis, while runtime verification offers superior coverage and anomaly detection in dynamic execution environments. The combined framework demonstrates improved overall performance by leveraging the strengths of both techniques, achieving higher detection accuracy and broader validation coverage.

Table 1. Verification Performance Metrics Across Static and Runtime Approaches

Metric	Static Verification	Runtime Verification
Detection Rate	92%	95%
False Positives	4%	7%
Validation Latency	Low (~10 ms)	Moderate (~35 ms)
Rule Coverage	85%	98%

Scalability analysis further confirms the robustness of the proposed framework under increasing logic complexity. As the number of rules and execution paths grows, the integrated verification approach maintains stable performance, with manageable increases in validation latency. The framework effectively scales by distributing verification tasks across static and runtime components, ensuring that neither phase becomes a bottleneck. These findings validate the suitability of the proposed verification model for enterprise-scale low-code systems, where complexity and execution variability are inherent characteristics.

IV. Conclusion

The integration of static and runtime verification within low-code systems demonstrates a significant advancement in ensuring the correctness and reliability of auto-generated application logic. By combining pre-execution analysis with continuous runtime monitoring, the proposed framework addresses both structural and behavioral inconsistencies that are otherwise difficult to detect in abstraction-driven environments. The results confirm that such a hybrid approach effectively reduces logical defects, improves execution consistency, and enhances overall system trustworthiness in enterprise applications.

A key contribution of this work lies in establishing verification as a core component of the low-code development lifecycle rather than a supplementary activity. Static verification provides early detection of rule conflicts, unreachable states, and structural anomalies, thereby preventing defective logic from reaching deployment. Runtime verification complements this by capturing execution-time deviations and context-dependent anomalies, ensuring that system behavior remains aligned with defined constraints. Together, these mechanisms create a robust validation framework capable of handling the complexities inherent in auto-generated logic.

The impact of this integrated verification model is particularly evident in enterprise scenarios where reliability and compliance are critical. Improved defect detection rates and higher rule coverage contribute directly to reducing operational risks, while controlled validation latency ensures that system performance remains within acceptable limits. The framework thus supports the development of dependable low-code applications in domains such as finance, healthcare, and regulatory systems, where even minor inconsistencies can have significant consequences.

Despite these advantages, certain limitations must be acknowledged. The effectiveness of static verification is constrained by the level of abstraction and the accuracy of rule extraction from low-code configurations. Similarly, runtime verification introduces additional computational overhead, which may become significant in highly distributed or real-time environments. Balancing verification depth with system performance remains a key challenge, particularly as application complexity and scale continue to grow.

Future research should focus on extending the framework toward automated formal verification techniques that can provide stronger correctness guarantees for auto-generated logic. The integration of AI-assisted validation mechanisms offers promising opportunities for adaptive anomaly detection and intelligent rule analysis. Additionally, the development of continuous verification pipelines embedded within enterprise DevOps ecosystems can enable real-time validation throughout the application lifecycle. These directions will play a crucial role in advancing low-code platforms toward more reliable, scalable, and intelligent software engineering environments.

References

[1] Rymer, J. R., Koplowitz, R., Mines, C., Sjoblom, S., & Turley, C. (2019). The Forrester wave: low-code development platforms For AD&D professionals, Q1 2019. Forrester Report, Forrester.

[2] Cabot, J. (2020, October). Positioning of the low-code movement within the field of model-driven engineering. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (pp. 1-3).

[3] Wan, Z., Zhang, Y., Xia, X., Jiang, Y., & Lo, D. (2023, November). Software architecture in practice: Challenges and opportunities. In Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering (pp. 1457-1469).

[4] De Giacomo, G., Dumas, M., Maggi, F. M., & Montali, M. (2015, May). Declarative process modeling in BPMN. In International Conference on Advanced Information Systems Engineering (pp. 84-100). Cham: Springer International Publishing.

[5] Brambilla, M., Cabot, J., & Wimmer, M. (2017). Model-driven software engineering in practice. Morgan & Claypool Publishers.

[6] Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). Introduction to runtime verification. In Lectures on Runtime Verification: Introductory and Advanced Topics (pp. 1-33). Cham: Springer International Publishing.

[7] Vallecillo, A. (2015). On the industrial adoption of model driven engineering. Is your company ready for MDE?. International Journal of Information Systems and Software Engineering for Big Companies, 1(1), 52-68.

[8] De Sanctis, M., Bucchiarone, A., & Marconi, A. (2020). Dynamic adaptation of service-based applications: a design for adaptation approach. Journal of Internet Services and Applications, 11(1), 2.