Jaswanth Kumar Mandapatti[1], Srinivasarao Bandla[2], Vishnu Vardhan Reddy Kavuluri[3], Maheswara Rao Gorumutchu[4], Nareshkumar Jagadhabi[5]
[1]Advent Health, United States
[2]Deloitte Consulting LLP, United States
[3]Deloitte Consulting LLP, United States
[4]HYR Global Source Inc, United States
[5]Compnova Inc, United States

Received: 13-07-2024; Revised: 21-08-2024; Accepted: 07-10-2024

State Explosion Risks in Rule-Driven Execution Engines

Abstract
State explosion in rule-driven low-code execution engines presents a major challenge as increasing rule complexity leads to rapid growth in execution states and transitions. Existing studies have explored rule evaluation and declarative execution, but a unified understanding of how rule interactions, branching, and dependencies contribute to state space expansion remains limited. This work addresses this gap by modeling state growth as a multi-dimensional phenomenon influenced by rule chain length, branching factors, condition density, and dependency depth. The analysis demonstrates that state expansion follows non-linear patterns, with clear threshold points beyond which system performance degrades rapidly due to increased memory usage, transition overhead, and execution latency. The results provide insights into controlling state explosion through structured rule design, reduced branching, and optimized execution strategies, enabling more scalable and efficient low-code platforms.

Keywords: State Explosion, Low-Code Execution, Rule Engines, System Scalability.

1. Introduction

State explosion has emerged as a fundamental challenge in rule-driven low-code execution engines, where application behavior is defined through declarative rules rather than imperative logic. These platforms rely on runtime interpretation of rules, conditions, and workflows, which significantly increases the number of possible execution states as system complexity grows. The control logic is no longer fixed but dynamically evaluated, leading to rapid expansion of state transitions under varying inputs. This behavior has been recognized in recent studies on rule-based systems and declarative execution environments [1]. As a result, managing execution state becomes a critical concern for performance and scalability. The problem intensifies when multiple rules interact simultaneously. Understanding this phenomenon is essential for ensuring system stability.

Low-code execution engines operate by evaluating rule chains that define application behavior across multiple layers, where each rule may introduce branching conditions and alternative execution paths. This leads to combinatorial growth in the number of possible states, especially when rules are nested or interdependent. Research in rule-based reasoning systems highlights that such branching significantly increases computational complexity [2]. Additionally, the presence of conditional dependencies further amplifies state growth, as different combinations of rule outcomes must be evaluated [3]. The system must track multiple execution paths simultaneously. This increases memory usage and processing overhead. As rule density increases, the state space expands rapidly.

The complexity of rule evaluation is further influenced by the dynamic nature of inputs and real-time execution contexts. Each incoming event or user action can trigger multiple rule evaluations, leading to different state transitions. This dynamic behavior has been studied in event-driven rule engines, where state transitions are highly sensitive to input variability [4]. The system must continuously adapt to changing conditions while maintaining consistency across execution paths. This creates a situation where even small changes in input can lead to large variations in state outcomes. As a result, predicting system behavior becomes increasingly difficult. This unpredictability is a key characteristic of state explosion.

The interaction between rules introduces another layer of complexity, where dependencies between rules create cascading effects across the execution engine. When one rule triggers another, the number of possible execution paths increases significantly, leading to exponential growth in state space. Studies on dependency-driven execution models show that such interactions are a major source of state explosion [5]. Furthermore, rule conflicts and overlapping conditions require additional evaluation steps, increasing processing time [6]. The engine must resolve these conflicts while maintaining logical consistency. This adds further overhead to state management. These cascading interactions make the system highly sensitive to rule design.

Branching factors within rule-driven systems play a critical role in determining the rate of state

expansion. Each conditional branch introduces multiple possible execution paths, which multiply as rules are chained together. Research in formal verification and state space analysis has shown that branching is one of the primary contributors to exponential state growth [7]. In addition, recursive rule structures can lead to repeated evaluation cycles, further increasing the number of states [8]. These factors collectively create a rapidly expanding execution space. The system must explore or approximate this space during runtime. This significantly impacts performance and scalability.

Resource constraints become increasingly important as state space grows, particularly in terms of memory consumption and processing time. Execution engines must allocate resources to track active states, evaluate rule conditions, and manage transitions. Studies on execution engine optimization indicate that resource utilization increases sharply with state complexity [9]. Moreover, parallel execution of rules introduces additional overhead, as synchronization between parallel paths must be maintained [10]. This creates contention for shared resources. As a result, system efficiency declines under high state complexity. Managing resource allocation becomes a key challenge.

Despite these challenges, existing approaches often focus on optimizing individual components such as rule evaluation or memory management, without addressing the overall structure of state growth. Recent work in low-code platform analysis suggests that a holistic understanding of state expansion is necessary to mitigate explosion risks [11]. Additionally, research on execution modeling emphasizes the need for systematic analysis of rule interactions and state transitions [12]. These studies highlight the limitations of current approaches. There is a clear need for frameworks that capture the full complexity of rule-driven execution. Addressing this gap is essential for improving system robustness.

This research focuses on understanding how rule structures, branching conditions, and execution paths contribute to state explosion in low-code engines. It examines how state space grows under different rule configurations and identifies key factors that influence this growth. The study aims to provide insights into managing complexity and improving execution efficiency. By analyzing state behavior, it becomes possible to design more scalable rule-driven systems. The findings can support better rule design and system optimization. This contributes to more reliable and efficient low-code platforms.

2. Methodology

The methodology is designed to systematically analyze state explosion behavior in rule-driven low-code execution engines by modeling how rule evaluation leads to expansion of execution states. The approach treats the execution engine as a dynamic state transition system, where each rule evaluation generates new possible states based on conditions and dependencies. A formal representation is established to capture how rules interact, branch, and propagate execution paths. The focus is on quantifying how state space grows under varying rule configurations. This requires identifying measurable parameters that influence state expansion. The methodology integrates rule modeling with

execution tracing to observe system behavior. This structured approach enables clear analysis of state growth patterns.

The first step involves defining core rule evaluation parameters that contribute to state expansion. These include rule chain length, branching factor, condition density, and dependency depth. Rule chain length represents the number of sequential rules executed in a workflow. Branching factor captures the number of alternative paths introduced by conditional logic. Condition density reflects how many logical checks are embedded within each rule. Dependency depth measures how many rules depend on the outcome of previous evaluations. These parameters collectively determine the complexity of execution. Higher values in these parameters directly increase the number of possible states. They form the primary drivers of state explosion.

To capture state growth quantitatively, the methodology introduces state space growth indicators such as state count, transition rate, state reuse ratio, and expansion coefficient. State count represents the total number of unique execution states generated during rule evaluation. Transition rate measures how frequently the system moves between states. State reuse ratio indicates how often previously visited states are revisited, which can reduce or amplify growth depending on system design. Expansion coefficient captures the rate at which new states are generated relative to rule execution steps. These indicators provide a measurable view of state expansion. They help identify whether growth is linear, polynomial, or exponential. Monitoring these indicators is central to understanding system behavior.

A formal execution model is constructed using directed graphs to represent rule interactions and state transitions. In this model, nodes represent execution states, while edges represent transitions triggered by rule evaluations. Each rule introduces edges that connect existing states to new states based on conditions. The graph evolves dynamically as rules are evaluated. This representation allows visualization of how state space expands over time. It also helps in identifying cycles, redundant paths, and high-density regions. The graph-based model provides a clear structure for analyzing execution complexity. It supports both qualitative and quantitative analysis of state explosion.

The methodology incorporates workload scenarios to simulate different execution conditions within the engine. These scenarios include simple linear rule chains, moderate branching workflows, and highly nested rule structures. Each scenario is designed to stress different aspects of the execution engine. Linear chains test sequential growth, while branching workflows examine combinatorial expansion. Nested structures introduce recursive and dependent behaviors that further increase complexity. By comparing these scenarios, it becomes possible to observe how different rule designs impact state growth. This comparative approach highlights the sensitivity of the system to rule configuration. It also helps identify worst-case conditions.

To evaluate execution performance, a set of performance metrics is defined, including evaluation time, memory consumption, and transition overhead. Evaluation time measures how long it takes to process

rule chains and generate corresponding states. Memory consumption reflects the resources required to store active and historical states. Transition overhead captures the cost of moving between states during execution. These metrics provide insight into how state explosion affects system efficiency. As state space grows, these metrics typically increase, indicating performance degradation. Tracking these metrics allows identification of thresholds where the system becomes inefficient. This supports performance-aware system design.

The methodology also accounts for optimization mechanisms that may influence state growth, such as rule pruning, state caching, and condition short-circuiting. Rule pruning reduces unnecessary evaluations by eliminating irrelevant rules. State caching allows reuse of previously computed results, reducing redundant state generation. Condition short-circuiting stops evaluation when outcomes are determined early. These mechanisms are incorporated into the model to observe their impact on state expansion. Their effectiveness is evaluated by comparing state growth with and without optimization. This helps in understanding how practical systems can mitigate explosion risks. It also provides guidance for designing efficient execution engines.

Fault and stress conditions are introduced to evaluate how the system behaves under abnormal or extreme scenarios. These include delayed rule evaluation, inconsistent inputs, and partial execution failures. Such conditions increase uncertainty in execution paths and can lead to additional state generation. The system must handle retries and re-evaluations, which further expand the state space. This aspect of the methodology reflects real-world behavior where ideal conditions are rarely maintained. It helps in understanding how robust the system is under stress. It also reveals how faults contribute to state explosion.

All parameters and indicators used in the analysis are systematically organized in Table 1 to ensure clarity and reproducibility. The table provides a compact representation of rule evaluation parameters and state space growth indicators. Each parameter is clearly defined to avoid ambiguity. This structured representation allows consistent application of the methodology across different scenarios. It also enables comparison of results across experiments. By standardizing these elements, the methodology ensures analytical rigor. The table serves as a reference for interpreting results.

Table 1. Rule Evaluation Parameters and State Space Growth Indicators in Rule-Driven Low-Code Execution Engines

Type	Parameter	Description
Rule Parameter	Rule Chain Length	Number of sequential rules executed
Rule Parameter	Branching Factor	Number of alternative paths per rule
Rule Parameter	Condition Density	Logical conditions within each rule
Rule Parameter	Dependency Depth	Interdependence between rules
Growth Indicator	State Count	Total number of generated execution states
Growth Indicator	Transition Rate	Frequency of state transitions
Growth Indicator	State Reuse Ratio	Repetition of previously visited states

Growth Indicator	Expansion Coefficient	Rate of new state generation per step

3. Results and Discussion

The results show how state space expands as rule complexity increases across different execution conditions. At low rule density and minimal branching, the system maintains a manageable number of states, and execution remains stable. The state graph grows in a near-linear pattern, with limited transitions and minimal redundancy. However, as rule chain length increases, the number of possible execution paths begins to rise. This leads to a noticeable increase in state count and transition activity. The system starts to exhibit early signs of complexity growth. This phase represents controlled expansion before exponential effects begin.

As branching factors increase, the results reveal a sharp transition from linear to exponential state growth. Each additional conditional branch multiplies the number of possible execution paths, significantly increasing the total state space. The surface plot in Figure 1 clearly shows this behavior, where state count rises steeply along the branching axis. The interaction between branching and rule chain length creates a compounding effect. Even moderate increases in both parameters lead to a large number of states. This demonstrates that branching is a dominant factor in state explosion. The system quickly becomes difficult to manage under these conditions.

The impact of condition density further amplifies state expansion, particularly when multiple logical checks are embedded within each rule. Higher condition density increases the number of evaluation outcomes, which in turn creates additional state transitions. The results indicate that systems with dense rule conditions experience faster growth in state space compared to those with simpler rules. In Figure 1, this is reflected as an increase in surface height along the condition axis. The combination of high condition density and branching leads to highly complex execution paths. This significantly increases computational overhead. It also makes state tracking more challenging.

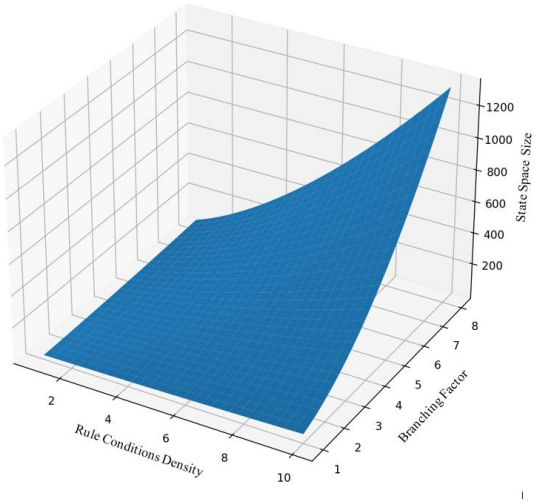

Figure 1. Multi-Dimensional State Space Expansion Across Rule Conditions, Branching Factors, and

Execution Paths

Dependency depth introduces another dimension to state expansion, where rules depend on the outcomes of previous evaluations. The results show that deeper dependencies create cascading effects, where a single state transition triggers multiple downstream transitions. This leads to layered growth in the state graph. In the surface plot, this behavior appears as a steep gradient along the dependency dimension. Systems with high dependency depth show delayed but rapid state expansion. Once triggered, the growth accelerates quickly. This confirms that dependencies contribute to hidden complexity within the execution engine.

The interaction between all four dimensions rule conditions, branching factors, execution paths, and dependency depth reveals that state explosion is not caused by a single parameter but by their combined effect. The surface plot illustrates regions of moderate growth and regions of extreme expansion, highlighting threshold behavior. Beyond certain parameter values, the system transitions into an unstable state where state count increases rapidly. This indicates the presence of critical limits in rule-driven execution. Identifying these limits is essential for system optimization. It allows developers to design rules that avoid extreme growth conditions.

4. Conclusion

The study confirms that state explosion is a fundamental limitation in rule-driven low-code execution engines, arising from the combinatorial growth of execution paths as rule complexity increases. The results show that factors such as branching, condition density, and dependency depth interact in a non-linear manner to expand the state space rapidly. Even moderate increases in these parameters can lead to a significant rise in the number of execution states. This behavior makes the system difficult to manage and directly impacts performance. The findings highlight that state explosion is not an isolated issue but an inherent characteristic of rule-driven execution.

A key insight from this work is the identification of critical thresholds where the system transitions from manageable growth to exponential expansion. These thresholds are influenced by the combined effect of rule parameters rather than any single factor. Once crossed, the system experiences rapid increases in state count, transition overhead, and resource consumption. This indicates that early-stage control of rule complexity is essential. Preventing the system from reaching these thresholds is more effective than attempting to manage explosion after it occurs. This understanding is important for designing scalable low-code platforms.

The analysis also emphasizes the importance of controlling rule interactions and execution structures. Reducing unnecessary branching, limiting deep dependencies, and simplifying rule conditions can significantly reduce state growth. The use of structured rule design and modular execution logic can

help contain complexity. Additionally, optimization techniques such as state reuse and pruning can mitigate the impact of state expansion. However, these techniques must be applied carefully to avoid compromising correctness. A balanced approach is required to maintain both efficiency and accuracy.

Another important outcome is the need for multi-dimensional monitoring of execution behavior. Tracking state count alone is not sufficient to understand system performance. Metrics such as transition rate, memory usage, and execution time must be analyzed together to capture the full impact of state explosion. The interaction between these metrics reveals hidden bottlenecks and feedback loops that accelerate system degradation. This highlights the need for integrated monitoring frameworks within low-code platforms. Such frameworks can provide early warning signals of excessive state growth.

In conclusion, this research provides a comprehensive understanding of state explosion risks in rule-driven low-code execution engines and identifies the key factors that drive this phenomenon. The findings offer practical guidance for designing more efficient and scalable execution systems by focusing on rule structure, interaction control, and performance monitoring. Future work can explore adaptive rule evaluation strategies, intelligent pruning mechanisms, and predictive models for state growth. Addressing these areas will further improve system resilience and enable low-code platforms to handle increasing complexity without compromising performance.

References

1. Nejati, F., Abd Ghani, A. A., Yap, N. K., & Jafaar, A. B. (2021). Handling state space explosion in component-based software verification: A review. *IEEE Access*, *9*, 77526-77544.
2. Favorito, M. (2022). Automata-theoretic techniques for reasoning and learning in linear-time temporal logics on finite traces.
3. Bhuyan, B. P., Ramdane-Cherif, A., Singh, T. P., & Tomar, R. (2024). Rule-based systems and expert systems. In *Neuro-Symbolic Artificial Intelligence: Bridging Logic and Learning* (pp. 63-85). Singapore: Springer Nature Singapore.
4. Shoaip, N., El-Sappagh, S., Abuhmed, T., & Elmogy, M. (2024). A dynamic fuzzy rule-based inference system using fuzzy inference with semantic reasoning. *Scientific reports*, *14*(1), 4275.
5. Mariappan, M., & Vora, K. (2019, March). Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (pp. 1-16).
6. Singh, B. (2023). Unleashing alternative dispute resolution (ADR) in resolving complex legal-technical issues arising in cyberspace lensing e-commerce and intellectual property: Proliferation of e-commerce digital economy. *Revista Brasileira de Alternative Dispute Resolution-Brazilian Journal of Alternative Dispute Resolution-RBADR*, *5*(10), 81-105.
7. Ying, M., & Feng, Y. (2021). *Model checking quantum systems: principles and algorithms*.

Cambridge University Press.

8. Hensel, C., Junges, S., Katoen, J. P., Quatmann, T., & Volk, M. (2022). The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*, *24*(4), 589-610.
9. Courtehoute, B. (2023). *Time and space complexity of rule-based graph programs* (Doctoral dissertation, University of York).
10. Feist, M., Brinkschulte, U., & Pacher, M. (2024, May). The Adaptation Mechanism of Chameleon-A Comprehensive Adaptive Middleware for Mixed-Critical Cyber-Physical Networks. In *2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)* (pp. 1-10). IEEE.
11. Keshireddy, S. R., Kavuluri, H. V. R., Mandapatti, J. K., Jagadabhi, N., & Gorumutchu, M. R. (2023). Enhancing Enterprise Data Pipelines Through Rule Based Low Code Transformation Engines. *The SIJ Transactions on Computer Science Engineering & its Applications*, *11*(1), 60-64.
12. Richards, M., & Ford, N. (2020). *Fundamentals of software architecture: an engineering approach*. O'Reilly Media.