Dealing with Pseudo-Random Software Faults in Complex Critical Driver Assistance Systems

Rüdiger Nortmann, MCG Management Consult GmbH

Summary

Modern automotive systems, particularly advanced driver assistance systems operating at SAE automation levels L3 and L4, present unprecedented challenges for functional safety management. While ISO 26262 establishes the foundational principle that software faults are exclusively systematic in nature, the increasing complexity of automated driving functions introduces a critical issue: software behavior that appears random despite its deterministic origins. This paper examines the phenomenon of pseudo-random software faults in complex automotive systems and presents practical methodologies for managing these safety-critical challenges within the ISO 26262 framework.

The Challenge of Complexity in Modern Automotive Software

The ISO 26262 Foundation and Its Limitations

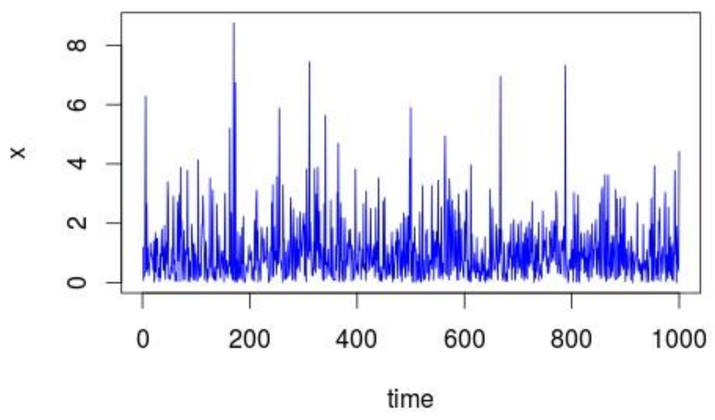
ISO 26262 was developed with a fundamental assumption that remains valid for systems of moderate complexity: software faults are not random but exclusively systematic. This principle holds true for low-complexity systems where the behavior can be fully understood, tested, and verified through traditional V-model development processes. However, this foundation encounters significant challenges when applied to highly complex software systems such as automated driver assistance systems operating at SAE levels L3 and L4.

In these advanced systems, faults can manifest in ways that appear random to observers, even though the underlying software remains deterministic. This apparent randomness emerges not from true stochastic processes but from the interaction of complexity factors that exceed our practical capacity for complete analysis and testing. The critical safety implication is that such faults cannot be detected solely through design-time verification activities. Instead, they require continuously active safety mechanisms that monitor system behavior during operation.

Understanding Different Types of Randomness

To properly address pseudo-random software behavior, it is essential to distinguish between fundamentally different types of randomness. The concept can be divided into two categories: mild randomness and wild randomness, each with distinct characteristics and safety implications.

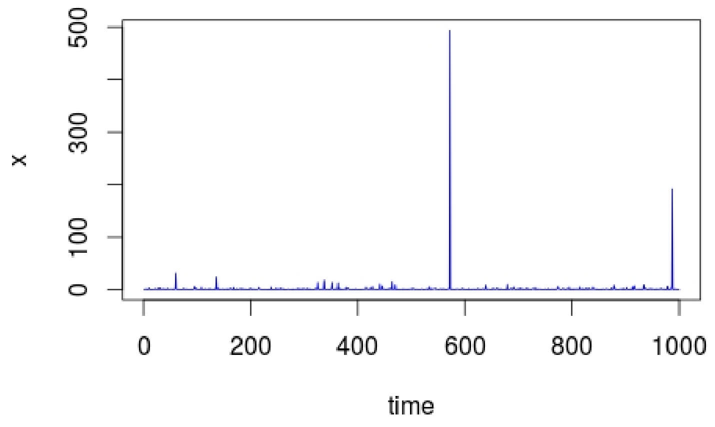
Mild randomness is subject to physical constraints and corresponds to physical quantities that can be measured and observed in the real world. When observing a system exhibiting mild randomness over time, patterns emerge that allow reasonable predictions about future behavior. The total behavior is not determined by any single observation or extreme event, and the randomness can be effectively characterized using probabilistic distributions. For safety engineering, this means that robust designs can be verified through stress testing and worst-case scenario analysis, as the physical boundaries of the system constrain the possible outcomes.



Wikipedia

Fig.1 Mild randomness

Wild randomness, in contrast, operates in the mathematical number space without physical constraints. It corresponds to calculated numerical values rather than directly observable physical phenomena. The critical difference is that wild randomness can be dominated by a small number of extreme events, making it difficult to characterize through observation alone. Understanding the true nature of such systems requires extended observation periods, and past information provides limited predictive value. Wild randomness is better represented by fractal distributions than traditional probability distributions, and verification requires antifragile design approaches combined with extensive simulation rather than simple stress testing.



Wikipedia

Fig. 2 Wild randomness

Sources of Pseudo-Random Software Behavior

Pseudo-random software failures arise from three primary sources in complex automotive systems. First, software operating within a complex system context, such as chaotic systems or stability control functions, inherently exhibits behavior that can appear random. The interaction between software and complex physical dynamics creates emergent behaviors that are difficult to predict despite the deterministic nature of both the software and the physical system.

Second, software that exceeds certain thresholds of complexity measured by established metrics such as cyclomatic complexity becomes practically impossible to fully test. When the number of possible execution paths through the software exceeds what can be verified through testing, untested combinations of conditions can produce unexpected results that appear random when they occur in operation.

Third, numeric computations introduce their own sources of apparent randomness. Binary floating-point arithmetic, for example, involves rounding errors and precision limitations that accumulate through iterative calculations. Algorithm robustness issues can cause small variations in input values to produce disproportionately large variations in output, creating behavior that appears random despite the mathematical determinism of the algorithms.

Complexity Metrics and Management

Categorizing Complexity in Software Systems

Software complexity can be categorized into three distinct types, each requiring different management strategies. Accidental complexity represents non-essential complexity introduced into the design by mistake. This form of complexity provides no value and should be eliminated through careful design review and refactoring. Incidental complexity is non-essential complexity introduced either through ignorance of better approaches or deliberately for reasons such as short-term development efficiency. While this complexity may seem justified during development, it should be minimized as it contributes no essential value to system functionality.

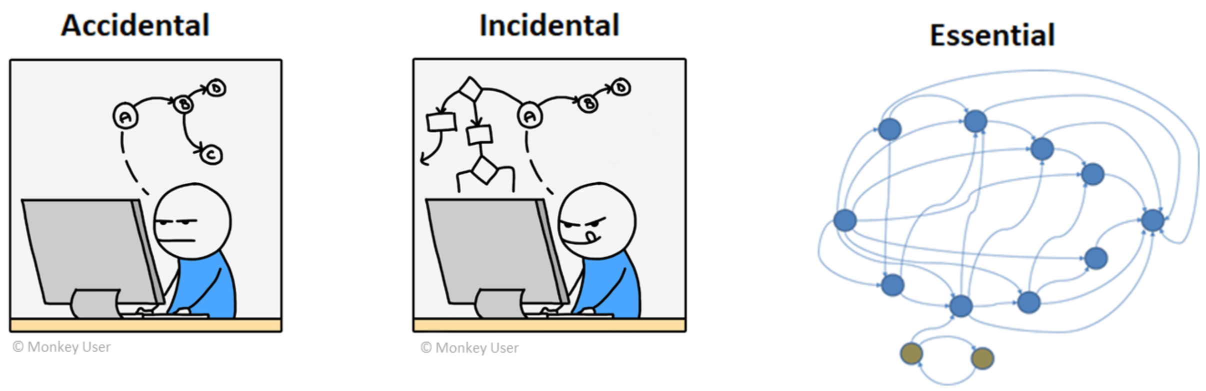


Fig. 3 Complexity categories

Essential complexity, however, is unavoidable. Automated driving systems represent a prime example of essential complexity. The problem domain itself is inherently complex, involving real-time sensor fusion, environmental interpretation, prediction of other traffic participants' behavior, and generation of appropriate vehicle control responses. No amount of clever design can eliminate this fundamental complexity; it can only be managed through appropriate architectural and verification strategies.

Cyclomatic Complexity as a Quantitative Measure

Cyclomatic complexity provides a quantitative measure of software complexity based on the control flow graph of the code. The metric is calculated using the formula $M = E - N + 2P$, where E represents the number of edges in the control flow graph, N represents the number of nodes, and P represents the number of connected components. This mathematical formulation captures the number of linearly independent paths through the code.

Tom McCabe, who developed this metric, provided interpretation guidelines that have direct implications for functional safety. A cyclomatic complexity between 1 and 10 indicates a simple procedure with little risk and low complexity. Values between 11 and 20 indicate more complex code with moderate risk, which becomes unacceptable for ASIL D development. Complexity values between 21 and 50 represent complex, high-risk code that is not acceptable for any ASIL level. Values exceeding 50 indicate untestable code with very high risk.

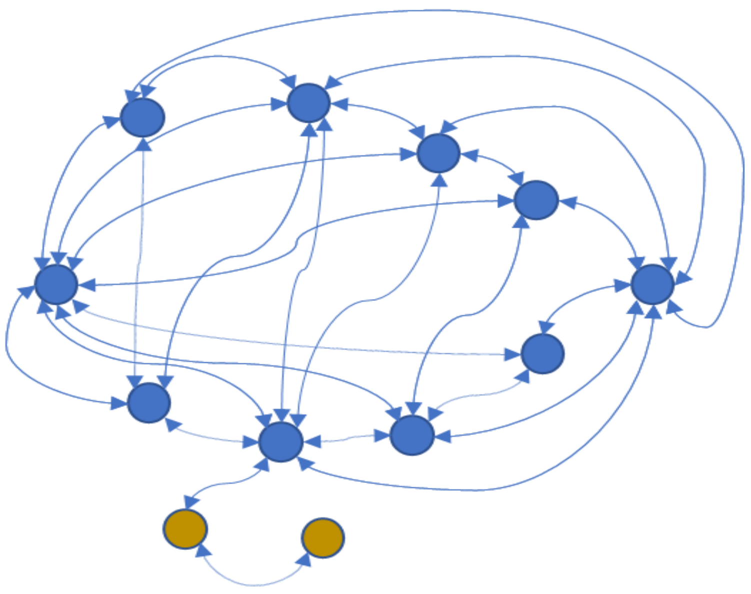


Fig. 4 Complex system with M = 48

These thresholds have profound implications for highly complex systems. Consider a system with a cyclomatic complexity of 48. Even with binary decision points only, this requires more than 576 system test cases to achieve complete path coverage. As automation levels increase from SAE L2 through L4, the complexity grows exponentially, quickly exceeding any practical testing threshold.

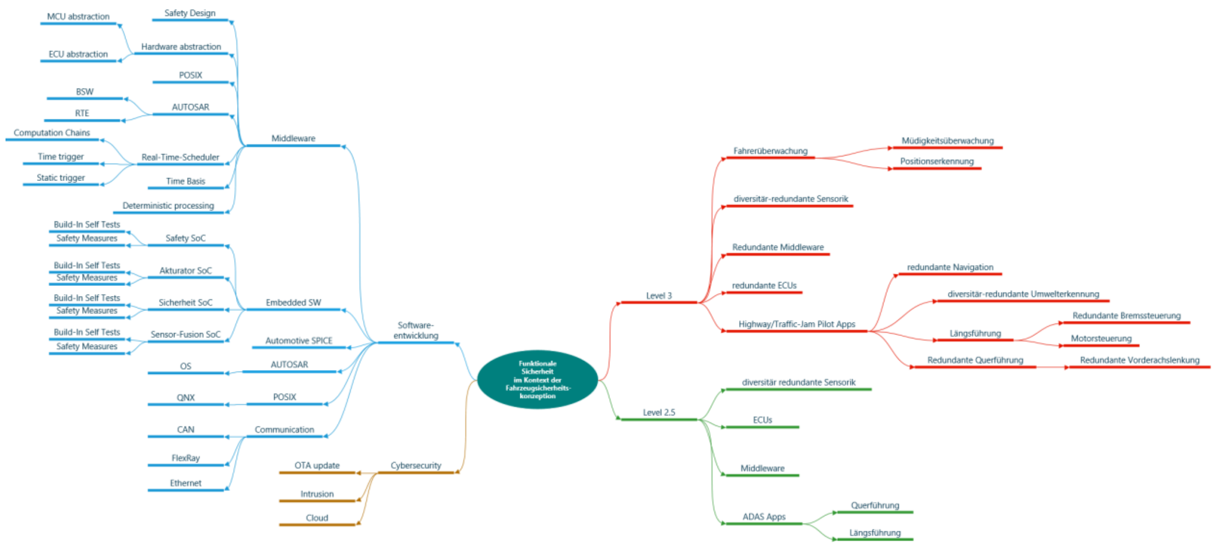


Fig. 5 System with SAE Level 2 and 3

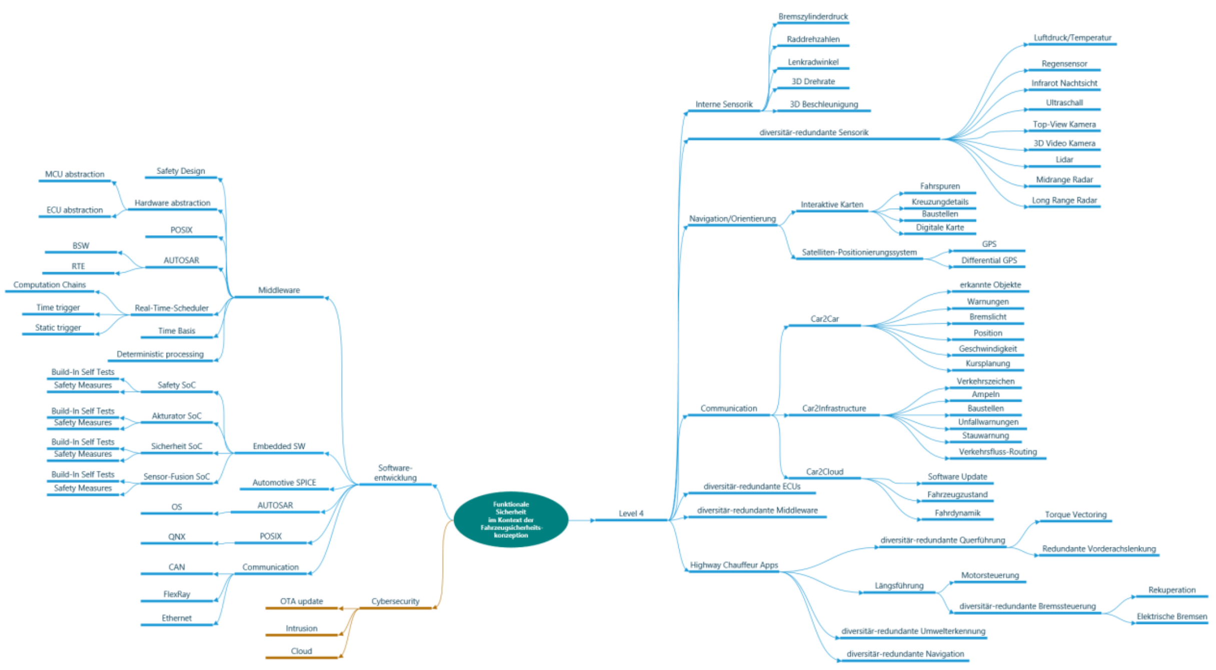


Fig. 6 System with SAE Level 4

The Insufficiency of Testing for Complex Systems

Traditional testing approaches face fundamental limitations when applied to complex systems. Several critical challenges emerge that make testing alone insufficient for safety assurance. First, design verification through testing cannot be complete for highly complex systems. The combinatorial explosion of possible states and transitions makes exhaustive testing impossible.

Second, equivalence class partitioning, a standard testing technique, provides no protection against "black swan" events—rare but significant failures that lie outside the tested equivalence classes. Third, safety can only be truly proven through simulation-based approaches where critical situations are first identified through simulation and then selectively tested. Fourth, even extensive endurance testing protects only against systematic errors, not against the pseudo-random failures that emerge from complexity. This limitation applies equally to hardware and software components.

Non-Linear and Chaotic System Behavior

by Stephen Cobeldick

Deterministic Systems Exhibiting Chaotic Behavior

A profound insight from complexity theory is that simple systems can be completely deterministic yet demonstrate chaotic behavior. The Lorenz system provides a canonical example that directly applies to automotive control systems. Originally derived to model fluid convection, the Lorenz system consists of three coupled ordinary differential equations. Despite the simplicity and complete determinism of these equations, the system exhibits chaotic behavior for certain parameter values.

Ordinary differential equation (ODE)

Chaotic behavior for values around:

- $\sigma = 10$
- $\beta = 8/3$
- $\rho = 28$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x),$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y,$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

The three differential equations describe how three state variables evolve over time, with each equation depending on the current values of multiple state variables. For parameter values around σ = 10, β = 8/3, and ρ = 28, the system produces the famous "butterfly attractor" pattern. Small perturbations in initial conditions lead to trajectories that diverge exponentially over time, creating behavior that appears random despite the underlying deterministic equations.

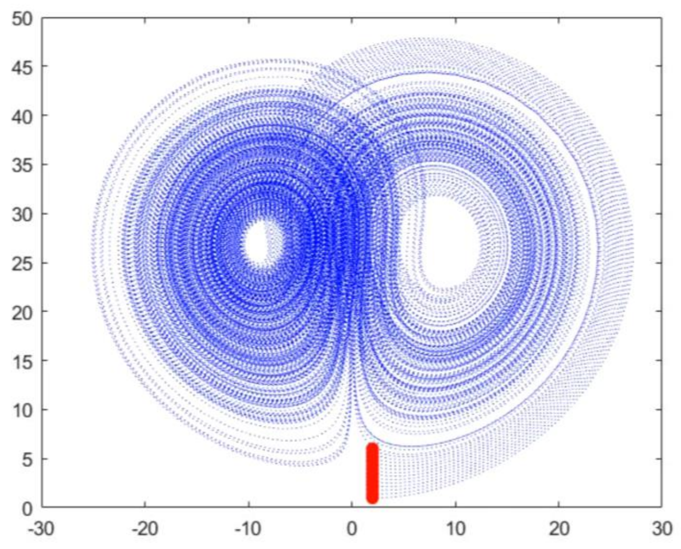


Fig. 7 Deterministic, yet chaotic

Relevance to Automotive Control Systems

The relevance of chaotic behavior to automotive systems cannot be overstated. Differential equations naturally describe many systems that utilize feedback control, including actuators, sensors with automatic gain control such as cameras and LIDAR that compensate for lighting levels, and decision-making algorithms that depend on prior system states. Each of these components can exhibit sensitivity to initial conditions that leads to divergent behavior over time.

The practical implication is that it becomes impossible to simulate or test all permutations of any non-trivial system. Later system states can only be sampled by distribution, and chaotic behavior may occur in regions of the state space that were not anticipated during design. The deterministic nature of the underlying equations provides no guarantee against apparently random behavior emerging during operation.

Probabilistic Characterization of Chaotic Behavior

Despite the deterministic nature of chaotic systems, their long-term behavior can be characterized probabilistically. By running repeated simulations with slightly different initial conditions and observing the system over extended time periods, the density of trajectories in different regions of the state space reveals an underlying probability distribution. This distribution represents where the system is likely to spend time during operation, even though individual trajectories remain unpredictable.

This insight connects directly to the definition of random hardware failure in ISO 26262-1:2018, clause 3.118. The standard defines random hardware failure as failure that can occur unpredictably during the lifetime of a hardware element and that follows a probability distribution. While this definition was written for hardware, the parallel to chaotic software behavior is striking. Both follow probability distributions, both can occur unpredictably, and both require similar approaches for safety management.

Methodologies for Managing Pseudo-Random Software Faults

Architectural Approaches: The Three-Component, Two-Channel Architecture

Managing pseudo-random software faults requires architectural strategies that go beyond traditional functional decomposition. A proven approach for highly automated driving functions employs a three-component architecture with two independent monitoring channels. The primary automated driving component implements the complex functionality required for SAE L3 or L4 automation. This component necessarily operates with high complexity and is therefore susceptible to pseudo-random faults.

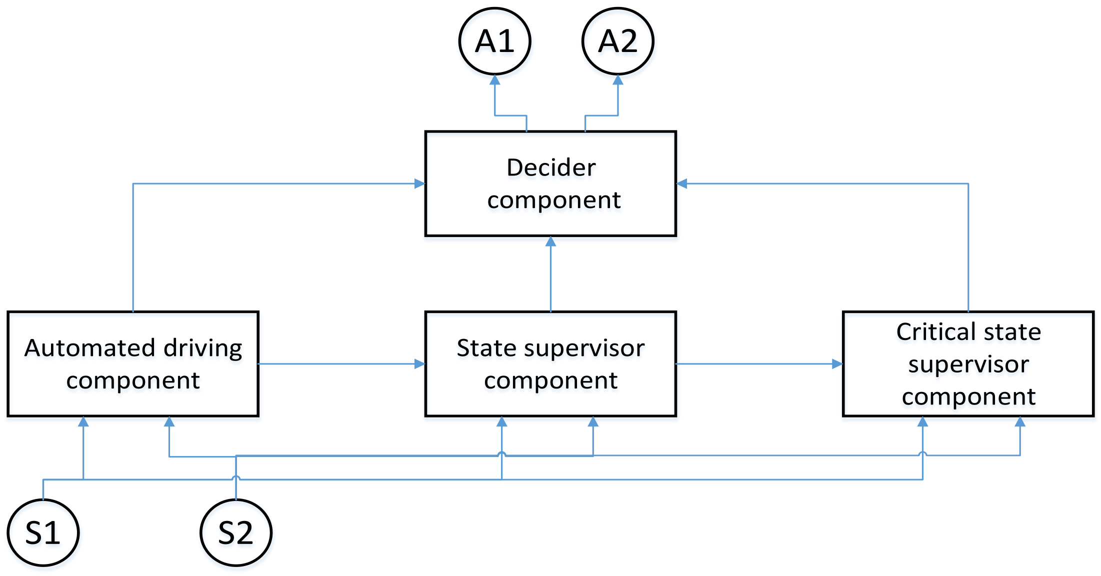

A1
A2
Decider component
Automated driving component
State supervisor component
Critical state supervisor component
S1
S2

Fig. 8 Automated Driving – 3-Components and 2-Channels

The state supervisor component implements the first level of safety monitoring. This component observes the behavior of the automated driving component and verifies that state transitions occur correctly and that the system remains within safe operational boundaries. To fulfill this safety function, the state supervisor must be designed with significantly lower complexity than the component it monitors. A target cyclomatic complexity below 20 makes the state supervisor itself verifiable through conventional means while still providing effective monitoring.

The critical state supervisor component provides an independent second channel of monitoring focused specifically on safety-critical states. This component implements the minimum necessary logic to verify that the system correctly enters and maintains safe states when required. By targeting cyclomatic complexity below 10, the critical state supervisor can be developed to ASIL D requirements with high confidence in its correctness. The decider component arbitrates between the outputs of the two channels, implementing appropriate response strategies when discrepancies are detected.

State Transition Modeling and Analysis

Formal modeling of system state transitions provides the foundation for both design verification and runtime monitoring. A complete state transition model identifies all valid system states, all permitted transitions between states, and the conditions that trigger each transition. For the three-component architecture, this means modeling the states of the automated driving function, the states of the monitoring components, and the relationships between them.

State transition modeling serves multiple purposes in managing pseudo-random faults. First, it enables Markov chain analysis to quantify the probability of reaching unsafe states under various fault assumptions. Second, it provides the specification for the state supervisor component, which implements runtime checking of state transitions. Third, it identifies critical transitions that require focused verification attention, either through targeted testing or formal methods. Fourth, it enables simulation-based verification where the state machine is exercised under a wide range of conditions to search for unintended behaviors.

Petri Net Analysis for Critical State Supervision

Petri nets provide a graphical and mathematical framework for modeling and analyzing concurrent systems with discrete states. For automotive safety applications, Petri nets excel at modeling the critical state supervision function because they naturally represent the relationship between states, transitions, and the conditions that enable transitions. A Petri net model consists of places representing states, transitions representing events or actions, and tokens that move through the net according to firing rules.

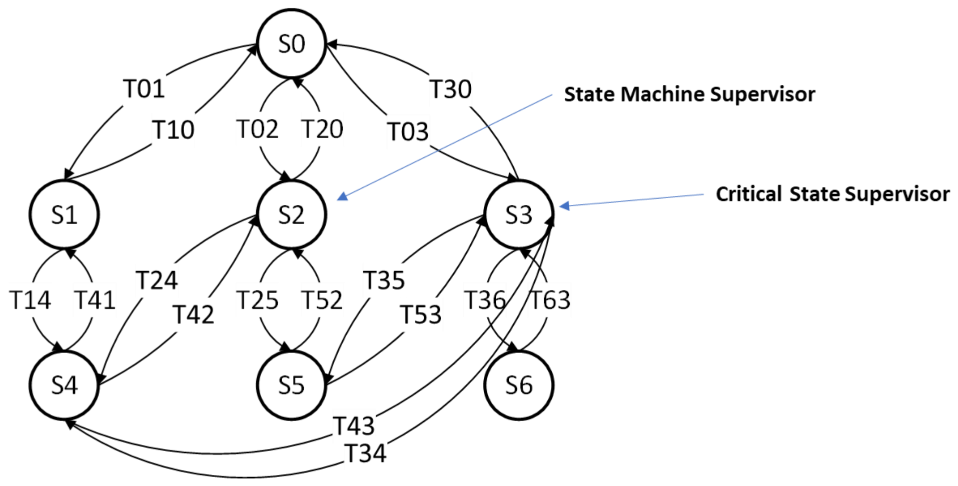


Fig. 9 State Transition Model for 3-Component System

The power of Petri net analysis lies in its ability to verify safety properties mathematically. Critical properties such as mutual exclusion—ensuring the system never simultaneously occupies conflicting states—can be proven through reachability analysis. Deadlock freedom can be verified by confirming that the net always has at least one enabled transition. Liveness properties ensure that the system can always eventually reach safe states when required. By constructing separate Petri nets for the main control logic and the critical state supervisor, and then verifying that the supervisor correctly monitors all possible states of the main system, designers can build confidence that no unsafe state can persist undetected.

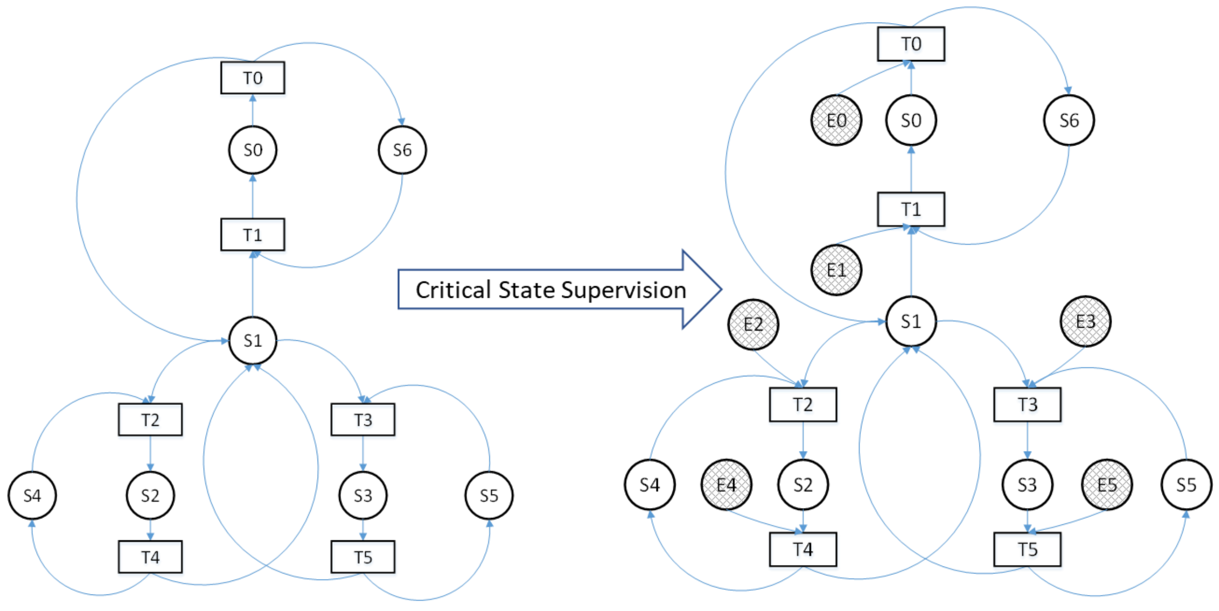


Fig. 10 Petri Net for Control of 3-Component System

Markov Chain Analysis for Probabilistic Safety Verification

Markov chain analysis provides the mathematical framework for quantifying the probability of safety-critical events in systems subject to pseudo-random faults. A Markov chain models the system as a set of discrete states with probabilistic transitions between states. The transition probabilities can be estimated from simulation data, field data from similar systems, or expert judgment informed by complexity analysis.

For automotive safety applications, Markov chain analysis addresses a critical question: what is the probability that a pseudo-random fault will lead to a safety violation before being detected by safety mechanisms? This probability depends on multiple factors including the fault occurrence rate as a function of complexity, the detection coverage of safety mechanisms, the reaction time of safety mechanisms, and the system architecture including redundancy and diversity.

The analysis proceeds by constructing a Markov chain that includes normal operational states, faulted states where pseudo-random errors have occurred but not yet been detected, detected fault states where safety mechanisms have identified the fault, and safe states where the system has responded to a detected fault. Transition probabilities between these states are assigned based on the factors mentioned above. Solving the Markov chain yields the steady-state probability of occupying each state and the expected time until unsafe states are reached, providing quantitative metrics for safety evaluation.

Simulation-Based Verification

Simulation plays an indispensable role in verifying complex systems subject to pseudo-random faults because it bridges the gap between the impossibility of complete testing and the need for evidence of safe behavior. Modern simulation environments such as Simulink, Rhapsody, and dSPACE enable comprehensive testing of system behavior across a vastly larger set of scenarios than physical testing could ever achieve.

The simulation-based verification process begins with identifying critical scenarios through systematic analysis methods such as STPA (System-Theoretic Process Analysis). These scenarios include boundary conditions where the system transitions between operational modes, edge cases where sensor inputs are ambiguous or conflicting, and stress conditions where the system operates near performance limits. Simulation then exercises the system model through these scenarios, with systematic variation of parameters to explore the behavior space.

Critically, simulation is not merely used to verify that the system behaves correctly in known scenarios. It is also used to discover previously unknown scenarios where the system behaves unexpectedly. By running extended simulations with pseudo-random variation in initial conditions and environmental parameters, engineers can identify regions of the state space where pseudo-random faults are likely to emerge. These discovered scenarios then become the focus of intensive targeted testing and potential design refinement.

Integration with ISO 26262 Safety Lifecycle

Reconciling Pseudo-Random Faults with Systematic Fault Treatment

The existence of pseudo-random software faults creates a conceptual tension with ISO 26262's foundation that software faults are systematic. However, this tension can be resolved by recognizing that the underlying cause remains systematic even when the manifestation appears random. The systematic cause is the excessive complexity that makes complete verification impossible. The pseudo-random manifestation is the observable consequence of this systematic cause.

This understanding leads to a dual strategy for safety management. At the design level, systematic fault avoidance continues through requirements management, architectural design principles, coding standards, and all the traditional methods for preventing software errors. Additionally, complexity is managed as a systematic cause by decomposing functions, limiting cyclomatic complexity, and applying design patterns that contain complexity within manageable modules.

At the operational level, safety mechanisms treat the manifestation of pseudo-random faults analogously to random hardware faults. Runtime monitoring detects when the system deviates from expected behavior, regardless of whether the root cause is a traditional software bug or a pseudo-random fault emerging from complexity. The safety goal is achieved through a combination of reducing the probability of faults through systematic methods and detecting and mitigating faults that do occur through runtime mechanisms.

Safety Mechanism Design and Validation

The design of safety mechanisms for pseudo-random faults must address several specific requirements. First, mechanisms must operate continuously during system operation rather than only during startup or periodic diagnostic intervals, because pseudo-random faults can emerge at any time depending on the specific sequence of conditions encountered. Second, mechanisms must have demonstrably lower complexity than the functions they monitor, establishing a credible independence between the monitor and the monitored function.

Third, mechanisms must achieve adequate detection coverage for the range of pseudo-random faults that could occur. This coverage is validated through fault injection testing where pseudo-random faults are artificially introduced into the system through methods such as state variable corruption, timing perturbations, and input signal modification. The safety mechanisms must detect these injected faults with high probability and within acceptable time limits. Fourth, the mechanisms must ensure that detected faults lead to safe states reliably. This requires careful design of the transition to safe states, including considerations of actuator control during transitions and coordination between multiple safety mechanisms.

Evidence and Argumentation for Assessment

ISO 26262 conformance requires appropriate evidence and safety argumentation. For systems with pseudo-random faults, the evidence portfolio must address several specific aspects. Complexity metrics for all software components must demonstrate that safety-critical components remain below acceptable thresholds and that monitoring components have substantially lower complexity than monitored components. State transition models and analysis results must document all possible state transitions and provide mathematical verification that unsafe states cannot persist undetected.

Simulation results must demonstrate system behavior across a representative range of scenarios, including edge cases and stress conditions identified through systematic hazard analysis. The simulation evidence must include coverage metrics showing that the simulation has exercised the critical regions of the state space. Fault injection test results must demonstrate detection coverage and response time for safety mechanisms when subjected to pseudo-random faults. Markov chain or similar probabilistic analysis must provide quantitative estimates of residual risk considering the probability of pseudo-random faults, detection coverage, and response time of safety mechanisms.

The safety argument ties this evidence together by explaining how the combination of complexity management, architectural safeguards, runtime monitoring, and verification activities achieves the required safety goals. The argument explicitly addresses pseudo-random faults as a recognized hazard and demonstrates that the safety measures are sufficient considering the specific characteristics of these faults.

Practical Implementation Considerations

Defining Signal Supervision Requirements

Effective runtime monitoring begins with comprehensive signal supervision that validates the plausibility of key internal signals and system states. For complex automated driving systems, this includes supervision of sensor fusion outputs to verify consistency between multiple sensor modalities, monitoring of internal state variables to detect values that exceed physically plausible bounds, checking of timing constraints to identify calculation delays that could indicate computational anomalies, and validation of actuator commands to ensure they remain within safe operating envelopes.

Each supervised signal requires carefully defined plausibility bounds that are tight enough to detect genuine pseudo-random faults but loose enough to avoid false positives that would cause unnecessary safe state transitions. These bounds are typically derived from simulation data showing normal operating ranges, augmented by analysis of worst-case scenarios, and validated through testing including fault injection.

State Machine Supervisor Implementation

The state machine supervisor implements runtime verification that the system follows its defined state transition model. This supervisor maintains an independent representation of the expected system state based on observed inputs and outputs. It compares this expected state against the actual reported state of the automated driving function, generating a fault signal when discrepancies exceed defined thresholds.

Implementation of the state machine supervisor requires careful attention to several design considerations. The supervisor must maintain synchronization with the supervised function, accounting for known timing delays in state transitions. It must handle transient discrepancies that occur during legitimate state transitions without generating false alarms. It must prioritize simplicity in its own design to achieve the target cyclomatic complexity below 20, which may require accepting less sophisticated state prediction algorithms than would otherwise be possible. The supervisor's output must interface appropriately with the critical state supervisor and decider components to enable coordinated system response to detected faults.

Critical State Supervisor Implementation

The critical state supervisor focuses exclusively on verifying that the system correctly enters and maintains safe states when required. This narrower scope enables even greater simplicity, targeting cyclomatic complexity below 10. The critical state supervisor typically monitors entry conditions for safe states, verifying that all prerequisites for safe state entry are satisfied, checks continuous safe state maintenance conditions to ensure the system remains in the safe state once entered, validates safe state exit conditions to prevent premature return to active operation, and coordinates with actuator control to verify that physical system responses match commanded safe state behaviors.

The implementation principle for the critical state supervisor is minimalism. Every function included must be directly necessary for validating safe state behavior. Any capability that is not strictly required for this purpose should be excluded, even if it would provide useful diagnostic information, because each additional capability increases complexity and thereby reduces confidence in the supervisor's own correctness.

Conclusion and Future Outlook

Summary of Key Principles

Managing pseudo-random software faults in complex critical driver assistance systems requires a paradigm that extends beyond traditional ISO 26262 approaches while remaining consistent with the standard's fundamental safety philosophy. Several key principles emerge from this analysis.

First, pseudo-random software behavior is a real phenomenon in sufficiently complex systems, arising from the interaction of complexity factors that exceed practical verification capabilities. Second, while the root cause remains systematic—namely, excessive complexity—the manifestation can appear random and must be managed with techniques analogous to those used for random hardware failures. Third, testing alone is insufficient for verifying complex systems; simulation, formal analysis methods, and runtime monitoring must complement testing. Fourth, architectural strategies that explicitly manage complexity through decomposition and monitoring are essential for achieving safety in complex systems.

Fifth, quantitative complexity metrics provide objective criteria for evaluating when software components remain within verifiable bounds versus when they require additional safeguards. Sixth, state-based modeling and analysis techniques including state transition models, Markov chains, and Petri nets provide the mathematical foundation for reasoning about system safety in the presence of pseudo-random faults. Seventh, the combination of multiple independent monitoring channels with demonstrably lower complexity than monitored functions provides effective detection and mitigation of pseudo-random faults during operation.

Implications for Current Practice

These principles have immediate implications for organizations developing highly automated driving systems. Complexity management must be treated as a first-class safety activity, with complexity metrics tracked throughout development and complexity budgets allocated to system components. Architectural decisions must explicitly address the complexity-safety tradeoff, recognizing that functional decomposition serves safety purposes beyond just software engineering best practices.

Verification strategies must embrace simulation as a peer to testing, with equivalent rigor in simulation scenario development, coverage metrics, and results analysis. Safety mechanism design must recognize that runtime monitoring is not merely a defense against random hardware failures but a fundamental strategy for managing pseudo-random software faults. Evidence portfolios for ISO 26262 conformance must include complexity analysis, formal model analysis results, and simulation evidence in addition to traditional test results.

Evolution of Standards and Practice

The automotive functional safety community continues to grapple with the challenges posed by increasingly complex systems. Future editions of ISO 26262 and related standards will need to provide more explicit guidance on managing complexity-related safety issues. The potential introduction of software error concepts or software fault models in ISO 26262 3rd edition represents one possible evolution, though such changes must be carefully evaluated to ensure they provide genuine safety value rather than just additional analytical overhead.

Consistency across related standards including SOTIF for performance limitations and cybersecurity standards for protection against malicious threats will be essential as these safety domains increasingly overlap in complex systems. The emergence of AI and machine learning components in safety-critical functions will further challenge traditional assumptions, requiring new approaches that build on the foundation established for managing pseudo-random faults.

Final Perspective

The recognition that software can exhibit pseudo-random behavior in sufficiently complex systems represents not a failure of ISO 26262's principles but rather an evolution in our understanding of how those principles must be applied. The fundamental safety engineering approach remains valid: identify hazards, assess risks, implement appropriate risk reduction measures, and verify effectiveness. What changes is the specific nature of the hazards we must address and the appropriate risk reduction measures for those hazards.

By explicitly recognizing pseudo-random software faults as a distinct hazard category, applying appropriate architectural and verification strategies, and providing clear evidence and argumentation for safety, organizations can develop highly automated driving systems that achieve required safety levels while advancing the state of the art in automotive functionality. The path forward requires neither abandoning established safety principles nor clinging rigidly to assumptions that no longer hold for highly complex systems, but rather thoughtfully evolving our practices to address the reality of modern automotive software development.