From Traditional Coding to the Use of Artificial Intelligence in Autonomous Vehicles

How to Ensure Software Safety Compliance

Divya Garikapati

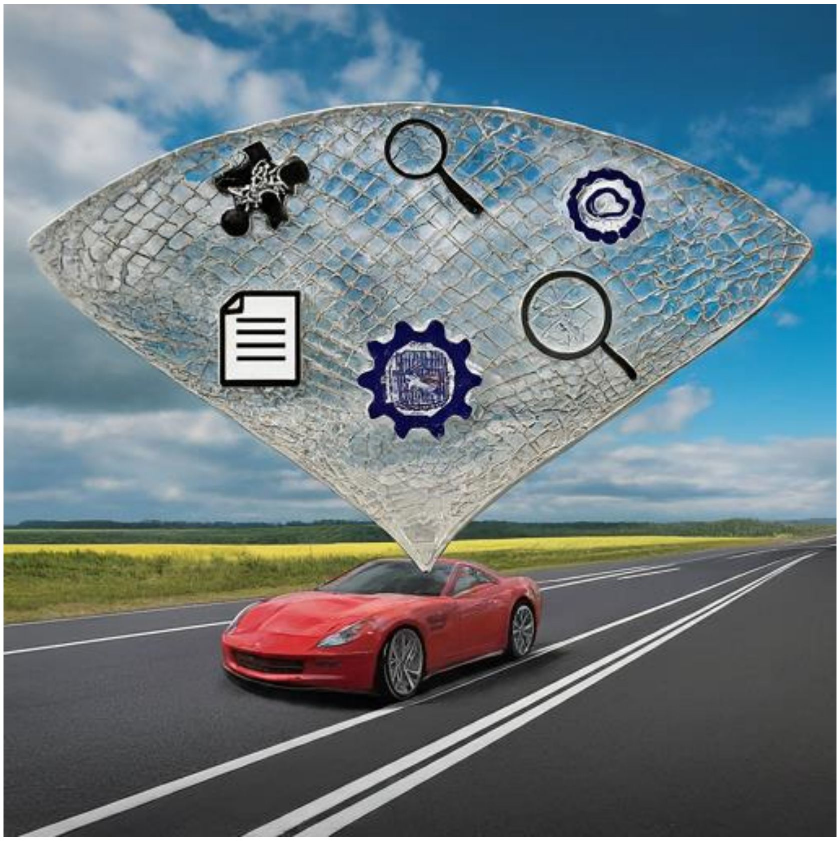
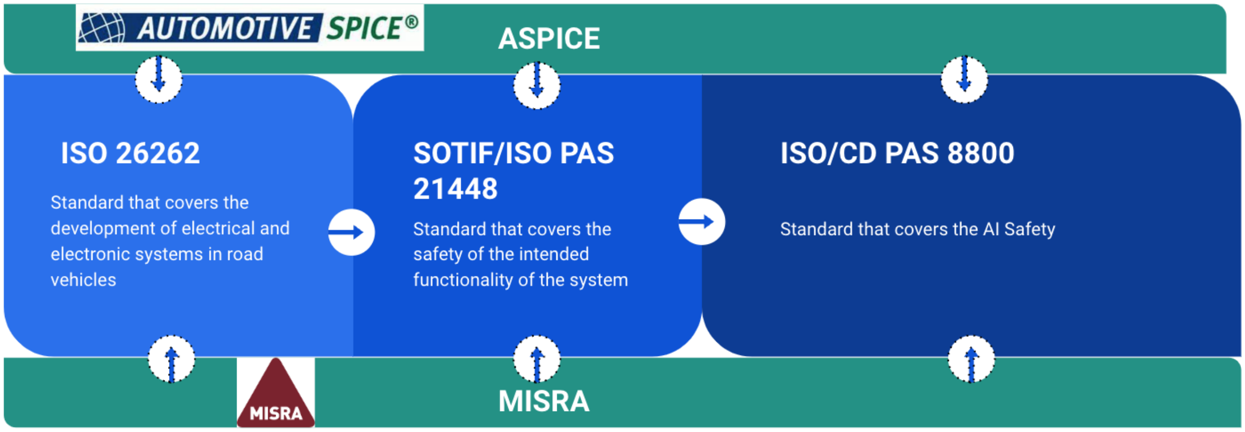


Image Source: Generated by Gemini AI with the Prompt “Create images for Software Safety Compliance”

Introduction

Ensuring software safety compliance in automotive applications involves rigorous testing, validation, and adherence to industry standards such as ISO 26262, ISO 21448, and ISO 8800. This includes a multi-layered approach like establishing coding guidelines like MISRA, thorough risk assessment, robust software development processes, and continuous monitoring for potential hazards or failures throughout the software lifecycle using ASPICE-like frameworks. Additionally, incorporating safety mechanisms like redundancy, fail-safes, detecting misuse, unexpected events, and error detection algorithms is essential to mitigate risks and ensure the safe operation of automotive software systems. Following coding guidelines and processes listed as per MISRA and ASPICE are needed for traditional coding. While these guidelines do not govern AI models or systems directly, they apply to code that interacts with these models. This report covers topics related to ensuring software safety compliance starting from the traditional coding to Artificial Intelligence (AI) based systems in automotive applications.

Figure 1: Multi-layered Approach


AUTOMOTIVE SPICE®
ASPICE
ISO 26262
Standard that covers the development of electrical and electronic systems in road vehicles
SOTIF/ISO PAS 21448
Standard that covers the safety of the intended functionality of the system
ISO/CD PAS 8800
Standard that covers the AI Safety
MISRA
MISRA

Coding Guidelines for Traditional Systems

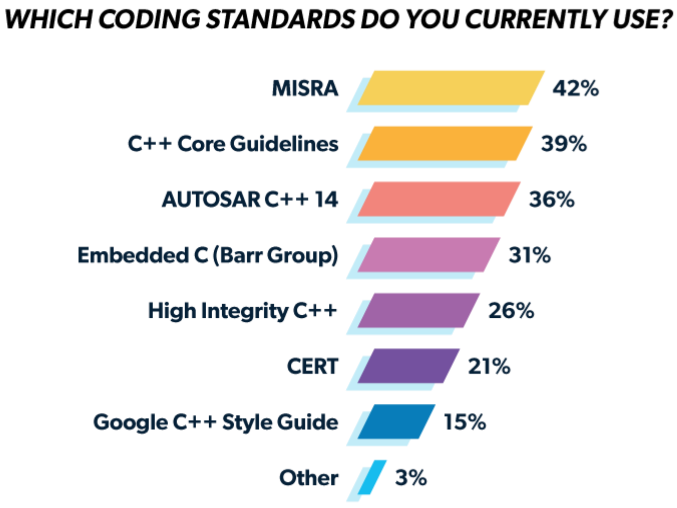
MISRA C and MISRA C++ are sets of coding guidelines, while ISO standards are more comprehensive documents that define requirements and specifications. Both provide the best practices and guidelines for ensuring software safety compliance and here are some of the key differences as listed in the table below:

Feature	MISRA C/C++	ISO Standards (e.g., ISO 26262)
Type	Coding Guidelines	Standards
Focus	Coding style, best practices, safety	Functional safety, quality, security (broader scope)
Goal	Reliable, portable, maintainable code	Overall safety, quality, and reliability of a system
Enforcement	Recommendations	Mandatory (depending on standards and regulations)
Level of Detail	Specific coding rules and recommendations	Framework and high-level requirements

MISRA guidelines and Automotive SPICE (ASPICE) address software safety in automotive applications in fundamentally different ways as shown in the table below. Achieving software safety compliance in automotive applications requires a two-pronged approach. MISRA C/C++ provides specific coding practices, like variable naming conventions and data type restrictions, to write safe and reliable code. It acts like a detailed recipe for building secure code. On the other hand, Automotive SPICE (ASPICE) focuses on establishing a robust software development process - think of it as setting up clear procedures for activities like risk management and quality assurance. By following the ASPICE framework, which might include using MISRA for coding, organizations can ensure they have a capable development process in place that promotes safety throughout the entire software lifecycle. In essence, MISRA provides the "how" to write safe code, while ASPICE focuses on the "what" - establishing a well-defined development process that fosters safety at every step.

Feature	**MISRA C/C++**	**Automotive SPICE (ASPICE)**
Focus	Coding style, best practices, safety	Software development process
Goal	Reliable, portable, maintainable code	Robust development process
Enforcement	Recommendations	Framework and high-level requirements
Level of Detail	Specific coding rules	Framework for activities and best practices

Figure 2: Different Coding Standards being Used.
Image Source: https://www.perforce.com/blog/qac/misra-cpp-2023-intro



MISRA C and C++ coding guidelines:

Coding Practices:

- Avoid unnecessary constructs that may lead to errors.
- Minimize run-time failures using static/dynamic analysis tools or explicit coding checks.
- Ensure that error information is tested when generated by functions.

Storage Practices:

- Do not assign an object to an overlapping object.

Arithmetic Practices:

- Document the use of scaled-integer, fixed-point, or floating-point arithmetic.

Language Practices:

- Adhere to the C++ Standard Incorporating Technical Corrigendum 1.
- Use a single, common compiler when using multiple compilers.
- Compilers and static checking tools should be validated and compliant with ISO 9001/ISO 90003.

Identifier Practices:

- Ensure different identifiers are typographically unambiguous.
- Do not hide outer scope identifiers with inner scope declarations.
- Make typedef names unique identifiers.
- Ensure class, union, or Enum names are unique identifiers.

Style Practices:

- Group #include directives together near the beginning of a file.
- Define and undefine macros only in the global namespace.
- Use consistent code layout and indentation.
- Use {} for block structures and avoid commenting out code.

In addition, compliance with MISRA guidelines can be checked by static analysis tools and the compiler. Compliance can also be manually reviewed, but this method is time-consuming and error-prone. Static analysis tools analyze source code for potential violations of the MISRA guidelines, while the compiler checks for compliance with the language standard.

ASPICE Framework

ASPICE ensures software safety compliance in the automotive industry by requiring a thorough requirement analysis to identify potential defects early in the development process. This analysis ensures that engineers are on the same page and that the software meets the needs of the OEM. Leveraging the V-model software development process ensures that every development step is mirrored by a testing step. This helps to catch defects early and avoid costly fixes later in the development process. ASPICE focuses on achieving the intended outcomes by requiring a thorough requirement analysis and leveraging the V-model software development process.

It emphasizes the importance of incorporating safety practices alongside it to ensure software safety compliance. Specifically, it is recommended that ASPICE be used in conjunction with ISO 26262, which is the functional safety standard for vehicles. This is because ASPICE focuses on how design is conducted, while ISO 26262 addresses random errors. By combining these two standards, suppliers can address both systematic and random errors, improving overall software safety. ASPICE is a framework that defines, implements, and evaluates processes for developing software-intensive systems in vehicles. It is becoming increasingly important as cars become more reliant on software. ASPICE helps ensure that software is developed correctly. OEMs use ASPICE to assess their suppliers. There are benefits for both OEMs and suppliers to using ASPICE.

ASPICE	ISO 26262
This applies to the development of all systems focusing on software and system parts	Applicable to vehicle systems categorized as safety-critical
Focuses on “Continuous Improvement” of the implemented process for improved capability level	Does not require process improvements unless there is a gap in compliance with the standard

Requirements Analysis also includes the consideration of cost and schedule impacts on the product development	There is no consideration of schedule or cost factors: safety is the primary concern of the standard
Focuses on the organization and project level process implementation: the assessment is performed on the organization/project level	Assessments are performed on the system level to ensure the functional safety objectives are satisfied for the defined safety-critical level of that particular system

Standards

ISO 26262-6 Best Practices related to SW Safety Compliance:

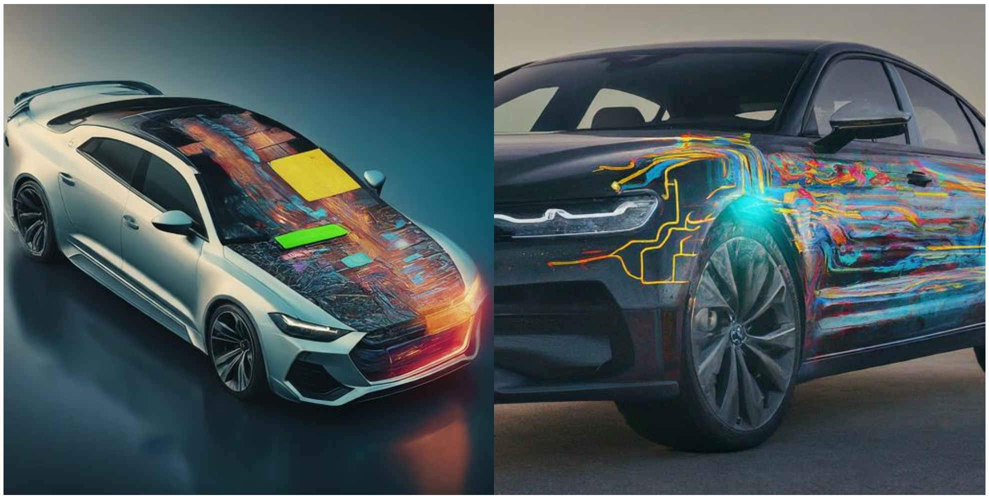


Image Source: Generated by Gemini AI with Prompt “Imagen of A 3D model of a car revealing its software architecture. Highlighted sections within the architecture represent key safety measures: color-coded sections for coding standards applied to traditional and AI components, a glowing circuit for safe AI development processes, and designated areas for rigorous testing and validation of both traditional software and AI algorithms.”

- **Document the software development environment.** This includes the software development process, tools, and techniques used.

- **Specify software safety requirements.** These requirements should be derived from the system safety requirements and should be specific, measurable, achievable, relevant, and time-bound.
- **Design the software architecture to meet the safety requirements.** The software architecture should be designed to prevent or mitigate hazards and to provide sufficient redundancy and fault tolerance.
- **Implement the software according to the design.** The software should be implemented according to the software design and should be tested to ensure that it meets the safety requirements.
- **Verify and validate the software.** The software should be verified and validated to ensure that it meets the safety requirements.
- **Maintain the software throughout its lifecycle.** The software should be maintained throughout its lifecycle to ensure that it continues to meet the safety requirements.

How should the software architectural elements be analyzed to identify possible design weaknesses and consequences?

To identify possible design weaknesses and their consequences, software architectural elements should be analyzed by examining their static and dynamic aspects, as well as their interfaces and interactions.

- Static aspects include the software structure, data types, external interfaces, and global variables.
- Dynamic aspects include the functional chain of events, logical sequence of data processing, control flow and concurrency of processes, and data flow through interfaces and global variables.

The analysis should also consider the following characteristics:

- Clarity (Comprehensibility): Is the design easy to understand, reducing potential misinterpretations?
- Consistency: Does the design follow established patterns to avoid inconsistencies that could lead to errors?

- Simplicity: Is the design free of unnecessary complexity, making it easier to identify and fix weaknesses?
- Verifiability: Can the design be rigorously tested to ensure it meets intended functionality and safety requirements?
- Modularity: Is the design broken down into independent components, simplifying maintenance and minimizing the impact of errors?
- Encapsulation: Do components hide internal details, promoting modularity and preventing unintended interactions?
- Maintainability: Can the design be easily modified and updated as needed to address future issues or evolving requirements?

Specific methods for analyzing software architectural elements include:

- **Walk-throughs and inspections:** Manual reviews of the design to identify errors and inconsistencies.
- **Simulation of dynamic behavior:** Execution of the design to observe its behavior and identify potential problems.
- **Prototype generation:** Creation of a simplified version of the design to test its feasibility and identify areas for improvement.
- **Formal verification:** Use of mathematical techniques to prove that the design meets its requirements.
- **Control flow analysis:** Examination of the design to identify potential control flow errors.
- **Data flow analysis:** Examination of the design to identify potential data flow errors.
- **Scheduling analysis:** Analysis of the design to ensure that it can meet its timing requirements.
- **Resource usage evaluation:** Estimation of the resources required by the design to ensure that it can operate within the available constraints.
- **Fault injection testing:** Introduction of faults into the design to test its robustness and fault tolerance.
- **Back-to-back comparison test between model and code:** Comparison of the design to the implemented code to identify any discrepancies.

The results of the analysis should be used to identify and mitigate any design weaknesses that could lead to errors or failures. This may involve modifying the design, implementing additional safety mechanisms, or conducting additional testing.

ISO 21448 - Safety of the Intended Functionality

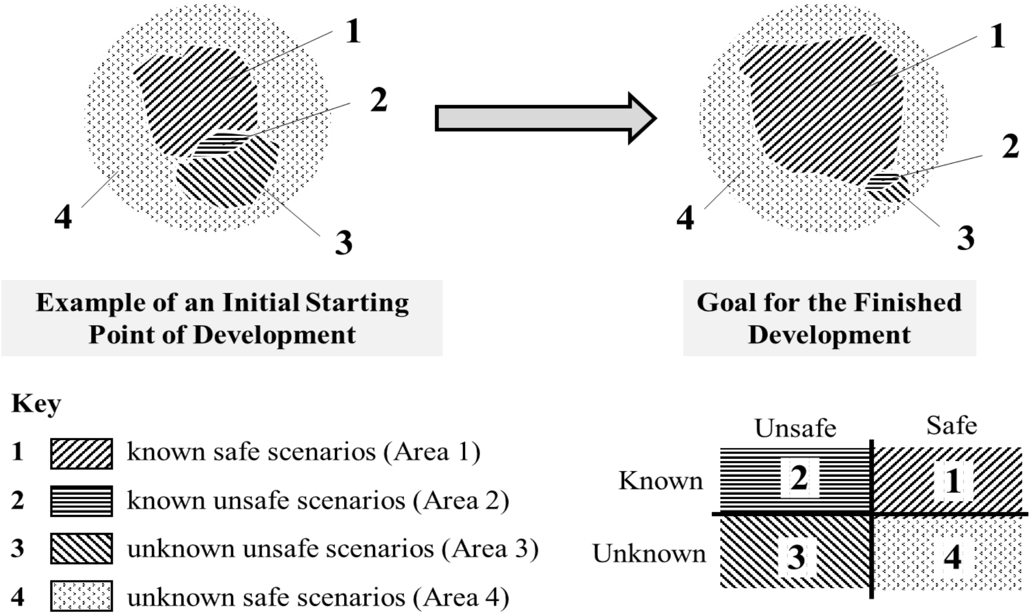
Image Source: Generated by Gemini AI with Prompt "Imagen of A 3D model of a car with a transparent hood revealing its software architecture. Highlighted sections within the architecture represent key safety measures: color-coded sections for coding standards applied to traditional and AI components, a glowing circuit for secure AI development processes, and designated areas for rigorous testing and validation of both traditional software and AI algorithms."

Image Source: Generated by Gemini AI with Prompt "operator misuse generate images for autonomous vehicles"

ISO/PAS 21448 contributes to ensuring software safety compliance for autonomous vehicles:

Focus on Intended Functionality: Unlike ISO 26262 which deals with safety even in case of malfunctions, ISO/PAS 21448 specifically targets the safety of the intended functionalities of the software. This ensures the software operates safely within its designed capabilities, reducing the risk of accidents due to limitations or errors in those functionalities.

Figure 3: SOTIF with Overall Goal to make Unknown Unsafe Scenarios Set to a minimum.
Image Source: ISO/PAS 21448 [https://www.iso.org/standard/70939.html]


1
2
3
4
Example of an Initial Starting Point of Development
1
2
3
4
Goal for the Finished Development
Key
1 ▨ known safe scenarios (Area 1)
2 ▤ known unsafe scenarios (Area 2)
3 ▧ unknown unsafe scenarios (Area 3)
4 ░ unknown safe scenarios (Area 4)
Unsafe
Safe
Known
Unknown
2
1
3
4

Mitigates Risks from Misuse and Unexpected Events: The standard addresses the potential for accidents arising from:

Misuse by humans: It considers how the software might be misused by people and implements safeguards to minimize risks in such scenarios.

Unexpected Situations: The standard encourages developers to consider how the software might react to situations outside its expected operating range. This helps identify and address potential safety concerns in unforeseen circumstances.

Provides a Framework for Development and Testing: ISO/PAS 21448 outlines a framework for developing, testing, and validating the software. This ensures the software undergoes rigorous checks to verify its intended functionalities operate safely. These checks might involve:

> ***Formal verification methods:*** Using mathematical proofs to guarantee the correctness of critical functionalities under specific conditions.

Simulations and Test Cases: Creating scenarios that push the software's boundaries and test its behavior in unexpected situations.

Complements ISO 26262: While ISO/PAS 21448 focuses on the safety of intended functionalities, ISO 26262 remains crucial for ensuring overall functional safety. Together, these standards provide a comprehensive approach to ensuring safe autonomous vehicle software. ISO/PAS 21448 promotes software safety compliance by focusing on the safe operation of intended functionalities and providing a framework for rigorous development, testing, and validation processes.

ISO 8800 - AI Safety

Image Source: Generated by Gemini AI with the Prompt “Create an image with AI software testing for cars”

The objectives of the safety analysis of AI systems are to:

- Ensure that the risk of a safety requirement violation (at the AI system level) due to safety-related errors is sufficiently low.
- Identify safety-related faults that can lead to the violation of a safety requirement.
- Identify potential causes of safety-related faults.
- Support the definition of safety measures for the prevention or control of safety-related issues.
- Support the verification of safety requirements, through the identification of requirements on data specification and collection, design requirements, and test requirements.

To ensure software safety compliance when AI is involved in developing software components, the following measures should be taken:

Derivation and specification of safety requirements: AI safety requirements should be derived from system-level safety goals and ensure the absence of unreasonable risk due to AI errors.

Development measures: AI system development activities should be tailored to ensure the suitability of the AI system to fulfill its safety requirements and to mitigate residual insufficiencies.

Architectural measures: Architectural measures should be implemented to ensure that residual insufficiencies of the AI system are sufficiently mitigated considering the target execution environment.

Safety analysis: AI safety analysis techniques should be applied to identify safety-related faults and their underlying issues, and to define prevention and control measures.

Verification and validation: AI systems should be verified and validated to provide evidence for compliance with safety requirements and the absence of unintended functionality.

Safety assurance during operation: Measures should be defined to assure the safety of the AI system during operation, including monitoring, detection, and mitigation mechanisms.

Continuous assurance: The safety of the AI system should be continuously assured throughout its lifecycle, including updates to safety requirements and design specifications, and collection of data from the field.

Confidence in AI development frameworks and software tools: Confidence should be demonstrated in the AI development frameworks and software tools used to create AI models, and their potential sources of errors and biases should be identified and mitigated.

Some of the specific guidelines provided include Specifying:

- Inference targets.
- Labeling procedure.
- Label evaluation policy (e.g., evaluation timing, evaluation procedure, evaluation measures, and acceptance criteria).

- Metrics and acceptance criteria for model performance.
- Model selection policy and ensure an appropriate model is selected according to the policy.
- Hyperparameter search policy and conditions, e.g., bounds, and document the determined hyperparameter values.
- Perform a safety analysis of the AI model.
- Provide requirements on the verification and validation processes to be employed to ensure that the data within the dataset is correct and appropriate for usage.
- The technical aspects, addressing the following items at a minimum:
    - Size of the dataset.
    - Format of the data within the dataset, including what parameters (both syntactic and semantic) describe the data.
    - Boundaries of the data within the dataset.
- Implement appropriate coding measures to avoid systematic faults like:
    - Enforcement of low complexity.
    - Use of language subsets.
    - Enforcement of strong typing.
    - Use of defensive implementation techniques.
    - Use of well-trusted design principles.
    - Use of unambiguous graphical representation.
    - Use of style guides.
    - Use of naming conventions.

Four types of measures to guarantee the functional adequacy of an AI system in its intended execution environment are:

1. Development Measure: Process steps or activities in the development of the AI system or component that ensure the fulfillment of safety requirements and/or enhance the AI trustworthiness characteristics.

2. Architectural Measure: Technical solutions implemented by the AI system or component to fulfill safety requirements and/or enhance trustworthiness characteristics.

3. Monitoring and AI System Modification: Measures to detect and mitigate distribution shifts related to the AI system during operation which can lead to hazardous malfunctioning behavior.

4. Optimization of Parameters and Optimization of Architectural Entities of AI Components: Optimization techniques enhance the performance and safety of the AI system by adjusting the parameters and architectural entities of AI components.

Types of testing to ensure AI Software Compliance:

White Box Testing:

- **Structural Coverage of AI Component:** Augmentation or filtering of input data based on structural coverage metrics to ensure comprehensiveness of testing, identify undesired behavior, and obtain evidence towards robustness.

Black Box Testing:

- **Error Guessing based on Knowledge or Experience:** Manually generating test cases by leveraging human knowledge and experience to identify potential errors and vulnerabilities.

Other Types of Testing:

- **Boundary Value Analysis:** Testing at the boundaries of the input domain to identify potential issues with input validation and edge cases.
- **Equivalence Class Partitioning:** Dividing the input domain into equivalence classes and testing representative values from each class to reduce the number of test cases while maintaining coverage.
- **State Transition Testing:** Analyzing the state transitions of the AI system and testing specific sequences of state changes to identify potential errors in state handling.
- **Mutation Testing:** Modifying the AI system's code slightly (e.g., by introducing syntactic errors) and testing whether the modifications affect the outputs to assess the system's robustness against code changes.
- **Formal Verification:** Applying mathematical techniques to prove or disprove the correctness of the AI system's behavior against its specifications.
- **Regression Testing:** Re-running existing test cases after changes to the AI system to ensure that those changes have not introduced new errors or unexpected behaviors.
- **Adversarial Testing:** Generating adversarial inputs designed to trick or bypass the AI system's defenses and identify potential vulnerabilities or weaknesses.

Conclusion & Key Takeaways:

Ensuring safe software in autonomous vehicles requires a multi-layered approach. Industry standards like ISO 26262, ISO 21448, and ISO 8800 provide a roadmap for development, testing, and validation throughout the software lifecycle starting from traditional coding towards using AI. This includes defining clear requirements, rigorous testing, and safety analysis to identify and mitigate risks. Coding guidelines like MISRA C/C++ promote reliable and secure code. Frameworks like ASPICE provide guidelines for analysis at different life-cycle stages of development and testing. For AI-based systems, additional steps are necessary. Developers must consider how the AI might react in unexpected situations and ensure the data used to train the AI is accurate and unbiased. Specific coding practices and testing strategies like structural coverage and boundary value analysis are crucial for validating AI functionality and safety. In essence, a combination of established safety standards, best practices, and thorough testing is essential to guarantee the safe operation of autonomous vehicles, both with traditional coding and cutting-edge AI technologies.

Acronyms List:

ISO - International Standards Organization

MISRA - Motor Industry Software Reliability Association

ASPICE - Automotive SPICE

SPICE - Software Process Improvement Capability Determination

AI - Artificial Intelligence

SOTIF - Safety of The Intended Functionality