Zephyr™ Project
Developer Summit
June 8-10, 2021 ▪ @ZephyrIoT

Software Quality vs Safety in ISO 26262

PETE BRINK – FUNCTIONAL SAFETY ENGINEERING LEADER @ KVA BY UL

Quality

What is quality?

Meeting or exceeding
your customer’s expectations

What is safety?

Establishing or reducing the risk of using a system as acceptably low

Quality vs. Safety

- Quality and Safety can be at odds with each other
  - A car that does not start is very safe …
  - But it clearly does not meet the customer's expectations

- The IEEE-CS SWECOM (Software Engineering Competency Model) defines both as "Crosscutting" skill areas
  - They both derive from following a rigorous development process

Systems Engineering

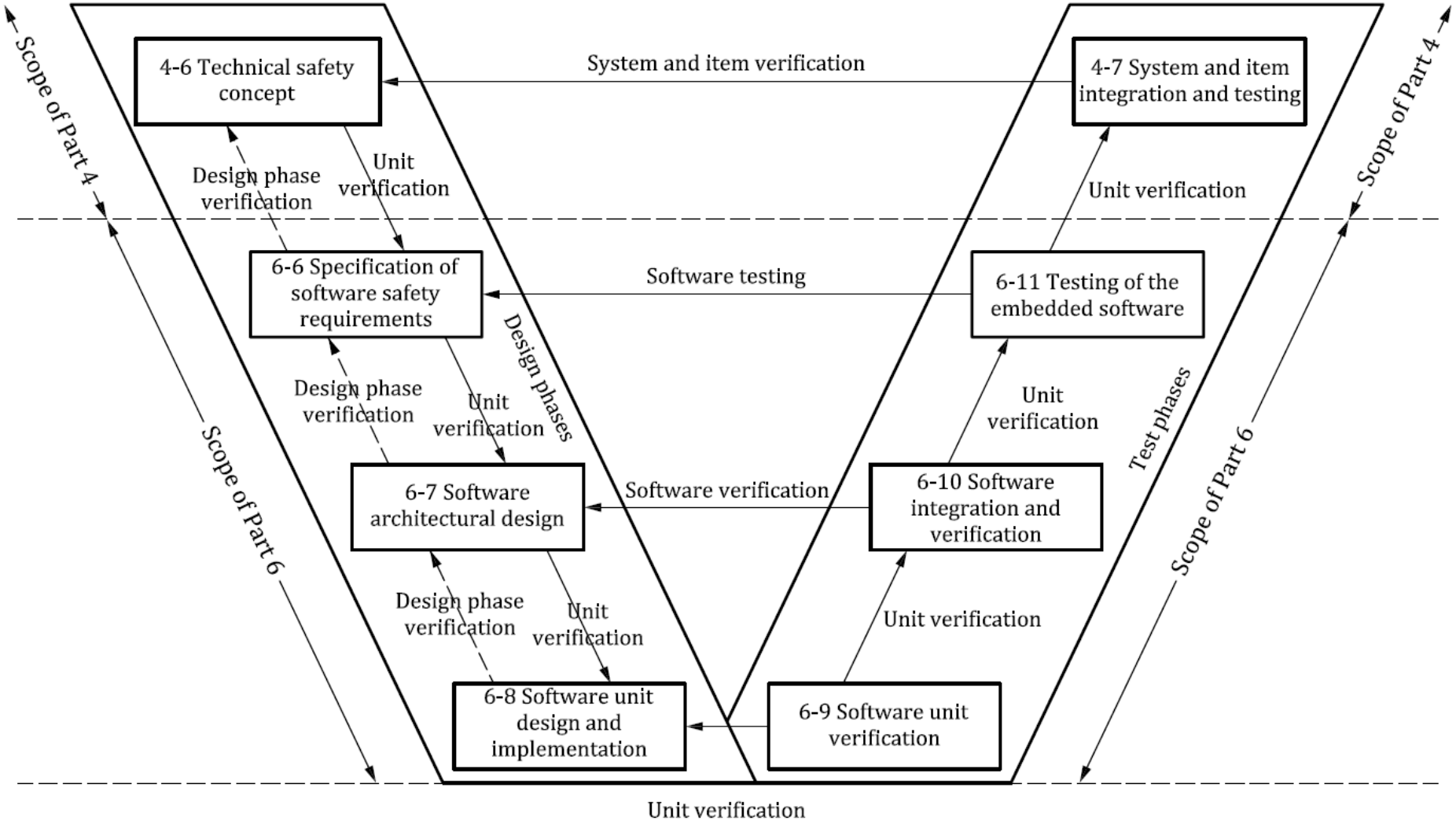
- A key concept in understanding Software Quality and Safety is that software is systemic:
  - Software defines the function of the system
  - Software does not fail – it does exactly what you tell it to do*
  - A software “error” requires a design change to correct
  - Process defines how the property (quality or safety) is produced

* Software is not subject to random failures like hardware

Automotive Safety: ISO 26262

- ISO 26262:2018 is the state-of-the-art automotive functional safety specification
- In Part 6 (Product Development at the Software Level,) ISO 26262 defines a software safety development model:
  - Clause 6: Requirements
  - Clause 7: Architectural Design
  - Clause 8: Unit Design and Implementation
  - Clause 9: Unit Verification
  - Clause 10: Software Integration and Verification
  - Clause 11: Testing of the Embedded Software
- Also included:
  - Clause 5: “General Topics for Product Development at the Software Level”

ISO 26262 Software Development



Prerequisites

Part 2 Clause 5

Part 6 Clause 5

Part 6: Clause 5

- Process Expectations listed in Part 2 Clause 5:
  - “- existing evidence of compliance with standards that support quality management.”
  - QMS = Quality Management System
  - Examples:
    - IATF 16949 in conjunction with ISO 9001
    - ISO 33000 standards, CMMI or ASPICE models
- Quality-related Documentation for Clause 5:
  - Documentation of the QMS compliant software development life cycle (SDLC)
  - Tailoring of the SDLC to comply with ISO 26262

Part 6: Clause 5

- Safety-related Documentation for Clause 5:
  - Documentation of the Software Development Environment
    - Suitability of the SW Development Environment for doing safety-critical development
  - Coding Guidelines:
    - Enforcement of low complexity
    - Language subset used
    - Strong typing (avoidance of compiler performing implicit casts)
    - Defensive programming techniques
    - Well-trusted design principles (SOLID, OOD, Design Patterns, etc.)

    MISRA
  - Software Configuration Management (SCM)
    - Branching and Merging Techniques used
    - General development process flow

Design Phases

Part 6 Clauses 6-8

Part 6: Clause 6 - Requirements

- Software Safety Requirements
  - Requirements must be reviewed according to the **quality** criteria
- Requirement Attributes:
  - Natural language
  - Informal (ASIL A or B) or Semi-formal (ASIL C or D) notation
  - Unique identification
  - Status
- Additional Safety Attributes:
  - Identifiable as a safety requirement
  - ASIL (Automotive Safety Integrity Level)
  - Allocated to an item or element

Requirements Quality Characteristics

- Individual requirements:
  - **Unambiguous**
  - **Comprehensible**
  - **Atomic**
  - Internally consistent
  - Feasible
  - Achievable
  - **Verifiable**
  - Necessary
  - **Implementation free**
  - Complete
  - Conforming
- Group requirements:
  - Hierarchical
  - Grouped appropriately
  - **Complete**
  - Externally consistent
  - Not duplicated
  - Maintainable
  - **Traceable**

Part 6: Clause 7 – Architectural Design

- Architecture
  - Must satisfy the software requirements
  - Verify that the software architectural design is suitable to satisfy the requirements
  - Both the safety-related and non-safety-related requirements are handled in one development process
  - Review the architecture according to the quality characteristics

- The architecture should include
  - The software components and their interactions.
  - Process Sequences
  - Timing behavior

Part 6: Clause 7 – Architectural Design

Characteristic	Description
Comprehensibility	The architecture should be understandable by people with the appropriate expertise who are not the author.
Consistency	The components in the architecture should expose functionality and interfaces in a consistent manner. Allocation of functionality to a component and component interfaces should follow a consistent pattern.
Simplicity	The architecture should minimize the complexity of software components.
Verifiability	The architecture should be verifiable by analysis and test at both the software system level and at the individual component level.
Modularity	The components within the architecture should have clear and defined interfaces by which they communicate with one another. They should be easily separable.
Encapsulation	(from OOD) Software components in the architecture should not expose internal data or functionality. All interactions between components should occur via defined interfaces.
Maintainability	Updates to components should not affect other components in the architecture.

Part 6: Clause 7 – Architectural Design

- Once the architectural design has been documented, the following are expected to be evaluated:
  - Documentation of additional safety mechanisms.
  - Upper estimate of system resources (CPU%, RAM, ROM, comms%)

- The main verification measures that distinguish safety from quality from an architectural design perspective:
  - Safety Analysis report (SW FMEA)
  - Dependent Failure Analysis report (DFA, FFI)

Part 6: Clause 8 – Unit Design

- The Unit Design is expected to detail:
  - The functional behavior of each software component
  - The internal design to the level of detail necessary for implementation.
  - Documentation of additional safety mechanisms.
- Unit Implementation is based on and traceable to the design
- Unit Design Quality Criteria to follow

Part 6: Clause 8 – Unit Design Quality

Characteristic	Description
Comprehensibility	The unit design should be understandable by people other than the author. This is an argument that the design should be simple enough for reviewers and implementers to understand
Consistency	The unit design should comply with the functionality and interfaces defined in the software architecture.
Verifiability	The unit design should be verifiable by analysis and test at the component level.
Maintainability	The component should be easy to modify. Furthermore, the component should be separable enough that a change to a it does not affect the operation of another component.

Part 6: Clause 8 – Unit Implementation Quality

Characteristic	Description
Correct order of execution	The source code should comply with order of execution between functions and components defined in the architecture and design.
Consistency	The source code should exactly implement the functions and interfaces defined in the architecture and design.
Correct control and data flow	The source code should correctly implement any algorithms and interfaces as defined in the architecture and design.
Simplicity	The source code should follow implementation that minimize code complexity as defined in the clause 5 documentation quality criteria. Complex code is difficult or impossible to unit test.
Readability / comprehensibility	The source code should be understandable by people other than the author.
Robustness	Robustness methods: prevent implausible values, mitigate execution errors, prevent division by zero, prevent data and control flow errors
Suitability for modification	This is a specific argument that functions not share variables and prevent unintended interaction by using strict functional interfaces. Functions can be modified without affecting other items within a given component.
Verifiability	The source code should be verifiable both by static analysis and code review and by testing at the function level.

Part 6: Clause 8 – Unit Implementation Safety

Characteristic	Description
Single entry and exit for a function	It is easy to miss or lose a specific return statement buried in an if-else tree causing a program to exit early.
No dynamic memory allocation	Allocate memory during initialization so that it can be verified before run-time starts and then manage the buffers internally.
Variable initialization	A decent percentage of bugs are related to usage of variables before they are initialized. If variables are initialized to a known value when they are created this is not a problem.
Avoid global variables	Global variables can become an unintended communication path between different high-level applications.
Implicit type conversion	Be aware that the compiler (for C and C++) will convert variables of internal types automatically (e.g., the compiler will convert a short to an int without telling you.) Make sure all type conversions are explicit.

Test Phases

Part 6 Clauses 9-11

Unit Testing

- Unit Testing
  - Testing the individual functions to demonstrate that they comply with the requirements
  - Treat the function as a black box and exercise via its exposed prototype

- Quality Metrics for Unit Testing
  - Static Analysis results report – how compliant the code is to the coding standards
  - Documentation of the test environment – how the code is being tested
  - Requirements coverage report - how many of the requirements are implemented
  - Code coverage report - how much of the code has been exercised

Unit Testing

- Safety Metrics for Unit Testing – Code Coverage
  - Varies according to the ASIL you are targeting
  - Statement Coverage – All statements in a function have been executed
  - Branch Coverage – All branches in a function have been taken
  - MC/DC – Modified Condition / Decision Coverage (ASIL D only)
    - Each entry/exit point is invoked
    - Each decision takes every possible outcome
    - Each condition in a decision takes every possible outcome
    - Each condition in a decision is shown to independently affect the outcome of the decision

Integration Testing

- Quality metrics for Integration Testing
  - Provide evidence that the software components fulfill their requirements
  - Provide evidence that the software contains no undesired functionality
  - Verify there are sufficient resources to support the functionality
  - Effectiveness of the safety measures from the safety analysis
- Safety metrics for Integration Testing
  - Function coverage at the architecture level
    - The number of functions in a component that have been executed
  - Call coverage
    - The number of calls between architecture-level components that have been exercised

Embedded Software Testing

- Software Quality
  - Provide evidence that the embedded software fulfills the requirements when executed in the target environment (Requirements Traceability Matrix)
  - Provide evidence that the embedded software contains no undesired functions or properties
  - The release report should contain:
    - Requirements coverage
    - Software status
    - All defects found and dispositioned

Embedded Software Testing

- Software Safety:
- Requirements-based testing
  - This is the final demonstration that the software portion of the safety requirements have been fulfilled.
  - All safety mechanisms should be demonstrated via testing without any instrumentation on a platform that is representative of the target production environment.
- Fault injection testing
  - The safety mechanisms should be demonstrated via the injection of faults into the system. In this instance, the faults should be external to the software, if possible, to cause the software safety mechanism to engage appropriately.

Questions?


Zephyr™ Project
Developer Summit
June 8-10, 2021 ▪ @ZephyrIoT