IEC 61508 Techniques

SCSD Lecture 25 Mar 2024

Overview

- IEC 61508-7 is an overview of techniques and measures.
- This covers all aspects of safety related systems.
- Appendix C discusses software.
- Unfortunately, the last revision date on the standard is 2002 so many of the techniques are out of date.
- Here we focus on coding standards.

Requirements and Design

- Annex C focusses on maintaining control over requirements and ensuring requirements are validated.
- Most of the design techniques date from the 20th century.
- The list includes a list of “formal methods” all of which have been substantially revised or superseded.

Coding Standards

C.2.6.2 Coding standards

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

Aim: To reduce the likelihood of errors in the safety-related code and to facilitate its verification.

Description: The following principles indicate how safety-related coding rules (for any programming language) can assist in complying with the IEC 61508-3 normative requirements and in achieving the informative “desirable properties” (see Annex F). Account should be taken of available support tools.

Modular Approach

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Modular approach (Table A.2-7, Table A.4-4)	Software module size limit (Table B.9–1) and software complexity control (Table B.9–2). Examples: • Specification of “local” size and complexity metrics and limits (for modules) • Specification of “global” complexity metrics and limits (for overall modules organisation) • Parameter number limit / fixed number of subprogram parameters (Table B.9–4) Information hiding/encapsulation (Table B.9–3): e.g., incentives for using particular language features. Fully defined interface (Table B.9–6). Examples: • Explicit specification of function signatures • Failure assertion programming (Table A.2-3a) and data verification (7.9.2.7), with explicit specification of pre-conditions and post-conditions for functions, of assertions, of data types invariants

Understandability

Code understandability • Promote code understandability (7.4.4.13) • Readable, understandable and testable (7.4.6)	Naming conventions promoting meaningful, unambiguous names. Example: avoidance of names that could be confounded (e.g., IO and I0). Symbolic names for numeric values. Procedures and guidelines for source code documentation (7.4.4.13). For example: • Explain why's and meanings (and not only what), • Caveats • Side effects Where practicable, the following information shall be contained in the source code (7.4.4.13): • Legal entity (for example: company, author(s), etc.) • Description • Inputs and outputs • Configuration management history (See also Modular Approach)

Verifiability

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Verifiability and testability • Facilitate verification and testing (7.4.4.13) • Facilitate the detection of design or programming mistakes (7.4.4.10) • Formal verification (Table A.5 - 9) • Formal proof (Table A.9 - 1)	• Wrappers for “critical” library functions, to check pre- and post-conditions • Incentives for using language features that can express restrictions on the use of particular data elements or functions (e.g., const) • For tool supported verification: rules for complying with the limitations of the selected tools (provided this does not impair more essential goals) • Limited use of recursion (Table B.1 – 6) and other forms of circular dependencies (See also Modular Approach)

Static Verification

Static verification of conformance to the specified design (7.9.2.12)	Coding guidelines for the implementation of specific design concepts or constraints. For example: • Coding guidelines for cyclic behaviour, with guaranteed maximum cycle time (Table A.2-13a) • Coding guidelines for time-triggered architecture (Table A.2-13b) • Coding guidelines for event-driven architecture, with guaranteed maximum response time (Table A.2-13c) • Loops with a statically determined maximum number of iterations (except for the infinite loop of the cyclic design) • Coding guidelines for static resource allocation (Table A.2-14) and avoidance of dynamic objects (Table B.1–2) • Coding guidelines for static synchronisation of access to shared resources (Table A.2-15) • Coding guidelines to comply with limited use of interrupts (Table B.1–4) • Coding guidelines to avoid dynamic variables (Table B.1–3a) • Online checking of the installation of dynamic variables (Table B.1–3b) • Coding guidelines to ensure compatibility with other programming languages used (7.4.4.10) Guidelines to facilitate traceability with design

Language Subsets

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Language subset (Table A.3 - 3) • Proscribe unsafe language features (7.4.4.13) • Use only defined language features (7.4.4.10) • Structured programming (Table A.4 - 6) • Strongly typed programming language (Table A.3 - 2) • No automatic type conversion (Table B.1 - 8)	Exclusion of language features leading to unstructured designs. E.g., • Limited use of pointers (Table B.1–5) • Limited use of recursion (Table B.1–6) • Limited use of C-like unions • Limited use of Ada or C++-like exceptions • No unstructured control flow in programs in higher level languages (Table B.1–7) • One entry/one exit point in subroutines and functions (Table B.9-5) • No automatic type conversion • Limited use of side effects not apparent from functions signatures (e.g., of static variables). No side effects in evaluation of conditions and all forms of assertions. Limited or documented-only use of compiler-specific features. Limited use of potentially misleading language constructs. Rules to be applied when these language features are used nonetheless.

Godd Programming Practice

Good programming practice (7.4.4.13)	When applicable: • Coding guidelines to ensure that, when necessary, floating point expressions are evaluated in the right order (e.g., “a-b+c” is not always equal to “a+c-b”) • In floating point comparisons: use only inequalities (less than, less or equal to, greater than, greater or equal to) instead of strict equality • Guidelines regarding conditional compilation and “pre-processing” • Systematic checking of return conditions (success / failure) Documentation, and, when possible, automation of the production of executable code (makefiles). Avoidance of side effects not apparent from functions signatures. When such side effects exist, guidelines to document them. Bracketing when operators precedence is not absolutely obvious. Catching of supposedly impossible situations (e.g., a “default” case in C “switches”). Use of “wrappers” for critical modules, in particular to check pre- and post-conditions and return conditions. Coding guidelines to comply with known compiler errors and limits set by compiler assessment.

Reuse

- The standard considers how reuse can be promoted and considers:
  - Use of trusted/verified components:
    - Proven in-use – here this is the practice e of reusing components that have extensive evidence from use in another system – this is not generally acceptable because use in a new context may reveal new faults.
    - Considering V&V evidence, the following must be considered:
      - that the element's design is known and documented;
      - the element has been subject to verification and validation using a systematic approach with documented testing and review of all parts of the element's design and code;
      - that unused and unneeded functions of the element will not prevent the new system from meeting its safety requirements;
      - that all credible failure mechanisms of the element in the new system have been identified and that appropriate mitigation has been implemented.

Traceability - forward

Forward traceability is broadly concerned with checking that a requirement is adequately addressed in later lifecycle stages. Forward traceability is valuable at several points in the safety lifecycle:

- from the system safety requirements to the software safety requirements;
- from the Software Safety Requirements Specification, to the software architecture;
- from the Software Safety Requirements Specification, to the software design;
- from the Software Design Specification, to the module and integration test specifications;
- from the system and software design requirements for hardware/software integration, to the hardware/software integration test specifications;
- from the Software Safety Requirements Specification, to the software safety validation plan;
- from the Software Safety Requirements Specification, to the software modification plan (including reverification and revalidation);
- from the Software Design Specification, to the software verification (including data verification) plan;
- from the requirements of IEC 61508-3 Clause 8, to the plan for software functional safety assessment.

Traceability - backward

Backward traceability is broadly concerned with checking that every implementation (interpreted in a broad context, and not confined to code implementation) decision is clearly justified by some requirement. If this justification is absent, then the implementation contains something unnecessary that will add to the complexity but not necessarily address any genuine requirement of the safety-related system. Backward traceability is valuable at several points in the safety lifecycle:

- from the safety requirements, to the perceived safety needs;
- from the software architecture, to the Software Safety Requirements Specification;
- from the software detailed design to the software architecture;
- from the software code to the software detailed design;
- from the software safety validation plan, to the Software Safety Requirements Specification;
- from the software modification plan, to the Software Safety Requirements Specification;
- from the software verification (including data verification) plan, to the Software Design Specification.

Architecture

- Considers a wide range of architecture that facilitate the control of some features:
  - Fault detection and diagnosis
  - Error detecting and correcting codes
  - Failure assertion programming
  - Diverse monitor
  - Software diversity (diverse programming)
  - Backward recovery
  - Re-try fault recovery mechanisms
  - Graceful degradation
  - Artificial intelligence fault correction
  - Dynamic reconfiguration
  - Safety and Performance in real time: Time-Triggered Architecture
  - UML

Verification and Testing

- This is covered in section C.5 of the standard and covers many techniques falling into these broad categories:
  - Blackbox testing: Probabilistic and systematic
  - Whitebox testing: Various flow analysis and coverage criteria
  - Error seeding to evaluate test sets
  - Formal proof and proof checking
  - Metrics, inspections, reviews
  - Simulation
  - Stress testing
  - Performance testing
  - Model-based testing
  - Regression testing

Summary

- This has been an overview of the range of techniques considered in IEC 61508-7
- The standard is somewhat old now.
- The sections on coding, reuse, architecture and V&V are still quite relevant.
- The aim of the lecture is to illustrate the range of possible techniques that can be used in a compliant approach
- This section of the standard is in need of significant updating.