Aerospace, Programming Languages, and Information Technology Co-Development: Safety-Critical Software

Brendan Sechter

2026-07-21

This ninth article of the twelve-part series covers safety-critical software as a distinct engineering discipline. The Space Shuttle software program treated in A244 established the empirical proposition that safety-critical software at industrial scale can be built to airline-comparable reliability targets. The engineering discipline that emerged in the subsequent decades codified the practices that make the proposition operational and extended them beyond aerospace into automotive, medical device, nuclear power, and industrial automation domains. This article treats the engineering discipline, the industry consensus process standards that codify it, the technical methods including formal verification and model-based development that supplement traditional testing, and the ongoing tension between waterfall and iterative process models in aerospace contexts.

The framing established in A237 treats safety-critical software as one of the artifacts of the aerospace-computing coupling that neither field alone would have produced. Traditional software engineering, developed for commercial applications where failure produces inconvenience rather than catastrophic outcomes, does not naturally optimize for the reliability targets that safety-critical systems require. Traditional aerospace engineering, developed for mechanical and hydraulic systems whose failure modes are relatively well-characterized, does not naturally provide the tools for reasoning about software failure modes. The engineering discipline of safety-critical software emerged from the intersection of the two traditions and continues to develop as both underlying disciplines evolve.

Emergence of the Discipline

Safety-critical software as a distinct engineering discipline emerged in the 1970s and 1980s from the practical experience of large aerospace software programs including the Space Shuttle software treated in A244 and its predecessors including the Apollo Guidance Computer software treated in A242. The Nancy Leveson analysis of the Therac-25 radiation therapy machine incidents between 1985 and 1987 in Leveson and Turner 1993 became one of the load-bearing case studies for the emergence of software safety engineering, demonstrating that software errors in a safety-critical medical device could and did cause patient injury and death. The analytical framework that Leveson developed, integrating software engineering with hazard analysis techniques inherited from aerospace and nuclear engineering, became one of the founding methodologies of the discipline.

The book-length treatment in Leveson 1995 and its successor Leveson 2011 together supply the standard textbook exposition of software safety engineering. The analytical techniques include Systems-Theoretic Accident Model and Processes hereafter STAMP, a framework that treats accidents as failures of control rather than as sequences of component failures, formalized in Leveson 2004 in Safety Science. Its associated hazard-analysis technique is STAMP-

based Process Analysis hereafter STPA. STAMP and STPA are widely used in contemporary safety-critical software programs and represent one of the methodological contributions of the discipline that later spread to non-safety-critical applications through the recognition that control-based reasoning applies broadly to complex system behavior.

The emergence of the discipline was substantially driven by mishap-driven learning. Each major accident involving software produced investigation reports, root-cause analyses, and process improvements that fed back into industry consensus process standards and organizational practices. Notable cases documented in the aerospace software literature include the 1996 Ariane 5 maiden flight failure caused by unhandled integer overflow in reused Ariane 4 code, treated in the official investigation report by Lions et al. 1996, the 1999 Mars Climate Orbiter loss caused by unit-conversion errors between imperial and metric systems, the 1999 Mars Polar Lander loss involving software interpretation of landing-gear deployment sensor data, and the 2018 and 2019 Boeing 737 MAX Maneuvering Characteristics Augmentation System failures. Each case contributed evidence to the discipline's ongoing understanding of the failure modes that safety-critical software must guard against.

The reliability targets that safety-critical software programs are expected to achieve are typically expressed as probabilities of catastrophic failure per operational hour. For the highest-consequence category commonly used across safety-critical domains, the target is

$$\lambda_{\text{catastrophic}} \leq 10^{-9} \text{ per operational hour}$$

corresponding to a mean time between catastrophic failures exceeding approximately 10^9 operational hours or roughly 10^5 years of continuous single-unit operation. Achieving this target through testing alone is infeasible because a test program that observes zero failures would need to run for a substantial fraction of the target mean-time-between-failure to provide meaningful statistical confidence, which motivates the complementary role of formal verification, structural coverage analysis, and process-based reasoning treated in the sections below.

Industry Consensus Process Standards

Industry consensus process standards codify the engineering practices that safety-critical software programs are expected to follow. The standards are organized by industry domain, with each domain producing its own standards through the consensus body appropriate to that domain. The RTCA maintains the primary civil-aviation software process guidance in a series of consensus documents produced through the joint work of aviation regulators, aircraft manufacturers, avionics suppliers, and academic researchers. The European Organisation for Civil Aviation Equipment hereafter EUROCAE issues parallel guidance harmonized with the RTCA documents. Automotive standards including ISO 26262 for road vehicles were developed through the International Organization for Standardization drawing on the aerospace precedents. Medical device software standards including IEC 62304 were developed through the International Electrotechnical Commission with similar heritage. Nuclear power software standards including IEC 60880 have their own consensus process. Industrial automation and process control software standards including IEC 61511 for process safety are developed through similar consensus bodies.

The common structure across these standards is a risk-based classification of software components into categories corresponding to the severity of failure consequences, with each category associated with process requirements that scale with the severity. The number of categories varies from three to five depending on the domain, but the general pattern is that the most safety-critical category requires the most stringent process. The process requirements typically include specification traceability, design and code review, testing to coverage criteria, configuration management, quality management systems, and independent verification

and validation. The strongest coverage criterion commonly required at the highest severity category is modified condition and decision coverage, formalized in the primary aerospace-industry paper by Chilenski and Miller 1994, which for a Boolean expression of N conditions requires a minimum test count of

$$N_{\text{tests, MC/DC}} \geq N + 1$$

designed so that each condition independently affects the decision outcome. This coverage criterion contrasts with statement coverage which requires only that every source line execute at least once, and with branch coverage which requires that each conditional branch take both true and false values, both of which are substantially weaker than modified condition and decision coverage for complex Boolean expressions. The verification effort required scales approximately as

$$E_{\text{V and V}}(k) \approx E_0 \cdot \alpha^k$$

with k the risk category index counted from the least stringent and multiplier α of order 2 to 5 per category increase, reflecting the engineering-economics reality that each successive category increase requires substantially more thorough verification than the previous.

The engineering value of consensus standards is that they codify the industry’s collective experience with what practices produce acceptable reliability at each risk level. The limitation of consensus standards is that they codify past experience and therefore lag current engineering practice, particularly for newer techniques including formal verification, model-based development, and machine learning components that the standards were not originally written to accommodate. The revision cycle for major consensus standards is typically five to ten years, which is fast relative to major aircraft development timescales but slow relative to computing technology change.

Formal Verification

Formal verification of software is the application of mathematical proof techniques to establish properties of programs with respect to specifications. The techniques include model checking, theorem proving, abstract interpretation, and symbolic execution. Each technique addresses a different class of properties and imposes different scalability constraints on the software artifacts it can analyze.

Model checking exhaustively explores the state space of a system model to verify that a temporal-logic property holds in all reachable states. The technique was introduced independently by Clarke and Emerson 1981 at the Workshop on Logic of Programs and by Queille and Sifakis in 1982 at the International Symposium on Programming, with all three of Clarke, Emerson, and Sifakis later receiving the 2007 ACM Turing Award for this contribution. The tools including SPIN developed by Gerard Holzmann at Bell Labs, NuSMV developed at Fondazione Bruno Kessler in Italy, and TLA+ developed by Leslie Lamport at Microsoft Research have been used extensively for aerospace applications. Model checking is limited by the state-space explosion problem in which the number of reachable states grows exponentially with the number of concurrent processes and the depth of exploration. For $N_{\text{processes}}$ processes each with $|S|$ local states and D possible values per shared variable across V shared variables, the composite state space is bounded above by

$$|S_{\text{composite}}| \leq |S|^{N_{\text{processes}}} \cdot D^{V}$$

which for realistic aerospace software of even modest complexity exceeds the computational capacity of any physical model checker. The engineering discipline of model checking includes techniques for abstraction, symmetry reduction, and partial-order reduction that permit useful analysis of systems whose full state space is infeasible to enumerate.

Theorem proving applies human-guided proof-construction techniques with computer-assisted proof checking to establish that a program implementation satisfies its formal specification. The tools including Coq, Isabelle, HOL, ACL2, PVS, and Lean have all been used for aerospace safety-critical software verification. Theorem proving scales further than model checking for property classes but requires substantial specialized expertise. The comparative engineering effort per line of verified code across the three technique classes satisfies approximately

$$E_{\text{static analysis}} \ll E_{\text{testing}} < E_{\text{model checking}} \ll E_{\text{theorem proving}}$$

with theorem-proving effort typically one to two orders of magnitude above tested-module effort and static-analysis effort typically one to two orders of magnitude below tested-module effort. The engineering trade-off among verification techniques therefore depends on the required assurance strength, the size of the software artifact, and the cost tolerance of the program.

Abstract interpretation, formalized by Cousot and Cousot 1977 as a theoretical framework, permits the static analysis of programs by computing sound over-approximations of program behavior in abstract mathematical domains. For a program P with concrete-semantics state set S_P and computed abstract-semantics state set $S_P^{\#}$, the soundness relation is

$$S_P \subseteq \gamma(S_P^{\#}), \qquad S_P^{\#} = \alpha(S_P)$$

where α is the abstraction function that maps concrete states to abstract states and γ is the concretization function that maps abstract states back to sets of concrete states. This soundness guarantee means that any property proved on the abstract semantics necessarily holds on the concrete semantics, though the converse fails and the abstract analysis produces false alarms at a rate that depends on the choice of abstract domain and analysis precision. The commercial application in the Astrée analyzer developed at École Normale Supérieure and later commercialized by AbsInt, described in Delmas and Souyris 2007, demonstrated that abstract interpretation could be used to prove the absence of classes of runtime errors including division by zero, integer overflow, and array-bounds violation across the entire Airbus A380 flight-control software. This industrial demonstration became one of the load-bearing empirical results for the practical viability of formal verification in industrial safety-critical software.

Model-Based Development

Model-based development is the engineering practice of building software from executable models rather than from hand-written source code. The engineer specifies system behavior in a high-level modeling language, and automatic code generation produces the equivalent source code that is then compiled and integrated into the target system. The value of model-based development is that the engineer works at a level of abstraction closer to the domain problem, and that the automatic code generation reduces the class of errors that hand-coding introduces.

Two commercial modeling environments dominate aerospace model-based development. MATLAB Simulink from MathWorks, extended by the Stateflow state-machine modeling capability and the Embedded Coder code generator, is widely used for control-system development including flight-control software. SCADE from ANSYS, based on the synchronous

programming language Lustre described in the primary language design paper by Halbwachs et al. 1991 in Proceedings of the IEEE with a mature commercial toolchain that generates production-ready code from formally defined models, is widely used for safety-critical avionics software with strong formal-verification integration. Both environments include qualification kits that support the use of the generated code in high-severity safety-critical categories under the industry consensus process standards.

The engineering trade-off with model-based development is that the model becomes the primary engineering artifact rather than the source code. The engineering discipline of model maintenance, model version control, model review, and model verification becomes the analog of the corresponding practices for hand-written source code. The tool-chain qualification requirement, meaning the requirement that the code generator itself be shown to produce correct code for the modeling constructs it accepts, becomes a substantial engineering artifact whose cost is shared across all users of the tool rather than borne by each program individually.

Process Tensions and Iterative Development

The traditional aerospace software development process is a variant of the waterfall model, with distinct phases for requirements analysis, architectural design, detailed design, implementation, testing, and integration each with formal signoff and configuration management between phases. The engineering argument for the waterfall approach in safety-critical aerospace software is that it produces the documentation trail that industry consensus process standards require, that it enforces disciplined thinking about requirements before implementation, and that it minimizes the class of coupling errors that arise from partially-defined interfaces between concurrently developed subsystems.

The counterargument, drawing on the commercial software engineering experience of the 1990s and 2000s in which iterative development approaches including Extreme Programming, Scrum, and continuous delivery replaced the waterfall model as the dominant commercial practice, was that waterfall requires perfect foresight about requirements and design decisions that no engineering team actually possesses, and that iterative approaches with short feedback cycles produce better outcomes than long-cycle waterfall approaches for essentially all software problems including safety-critical ones. The aerospace software community's response to the iterative-development case has been mixed, with some programs adopting hybrid approaches that combine iterative development within phases with waterfall-style formal signoffs between phases.

The V-model, a visualization of the waterfall model that pairs each requirements or design phase with a corresponding verification or testing phase, remains the standard reference architecture for safety-critical software processes. The V-model captures the bidirectional relationship between top-down decomposition of requirements into implementations and bottom-up integration of implementations against requirements. Contemporary aerospace software programs use the V-model as an organizing framework while implementing individual phases with iterative techniques where the engineering constraints permit. Continuous integration and continuous delivery pipelines have become standard within-phase engineering practice for essentially all contemporary aerospace software, though the between-phase controls that industry consensus process standards require remain waterfall-derived.

Structural complexity of software modules is commonly quantified using the McCabe cyclomatic complexity metric introduced in McCabe 1976

$$V(G) = E - N + 2P$$

for a control-flow graph G with E edges, N nodes, and P connected components, giving the

number of linearly independent paths through the module. Industry consensus process standards typically recommend cyclomatic complexity below approximately 10 to 15 per function or module to bound the engineering difficulty of comprehensive testing and human review, and safety-critical software programs typically enforce cyclomatic complexity limits through coding-standard automated checks that reject modules exceeding the specified threshold.

Framework Application to Safety-Critical Software

The six-axis framework introduced in A237 applies to safety-critical software as an engineering discipline with axis weightings reflecting its cross-cutting character.

The first axis is numerical computation demand. Safety-critical software itself does not impose numerical demand distinct from the applications it implements. The verification and analysis tooling for safety-critical software, however, imposes substantial computational demand. Model checking a moderate-sized flight-control system can consume weeks of computer time on a large workstation. Theorem proving a substantial software module can require months of engineer time supported by hours of interactive prover computation per proof step.

The second axis is real-time control. Safety-critical software commonly implements real-time control functions, drawing on the real-time operating system tradition treated in A241. The engineering integration of safety-critical software analysis with real-time scheduling analysis remains an active research area, with formal methods for combined timing-and-functional verification becoming available in the 2010s and 2020s.

The third axis is reliability and verification. This is the primary axis for safety-critical software as a discipline. The reliability targets, verification techniques, and process standards that the discipline codifies are the engineering artifacts that define it. The load-bearing empirical results include the Shuttle software 0.004 defects per thousand lines figure treated in A244, the Astrée analysis of the A380 flight-control software cited above, and the operational reliability records of commercial aircraft flight-control software across three decades of fly-by-wire operation.

The fourth axis is networking and distribution. Safety-critical software increasingly operates in distributed configurations, with networked avionics buses replacing point-to-point wiring in modern aircraft. The engineering integration of network protocol correctness with safety-critical software correctness draws on both the network protocol tradition treated in A243 and the real-time systems tradition treated in A241.

The fifth axis is software engineering as a discipline. Safety-critical software engineering is a sub-discipline of software engineering that has developed practices, tools, and process standards. Its diffusion into general software engineering has been substantial, with the industry consensus process standards influencing quality management systems in essentially all software domains, and formal verification techniques originally developed for aerospace increasingly applied to security-critical and business-critical commercial software.

The sixth axis is semiconductor economics and dual-use. Safety-critical software drives demand for the class of processors, memory technologies, and platform components that support high-reliability operation. The market for safety-critical computing platforms is small relative to commercial computing but sustains a dedicated supplier base including radiation-hardened processors, high-reliability memory, and formally qualified operating systems.

Conclusion

Safety-critical software emerged as a distinct engineering discipline in the 1970s and 1980s from the practical experience of aerospace software programs including Apollo, the Space Shuttle, and the early fly-by-wire commercial and military aircraft. The discipline codified its practices in industry consensus process standards issued by domain-specific bodies including

the RTCA for civil aviation. The technical methods that supplement traditional testing include formal verification, model-based development, and mishap-driven learning. The ongoing tension between waterfall and iterative process models reflects the difficulty of adapting commercial software engineering advances to the constraints of safety-critical development, with contemporary practice generally adopting hybrid approaches that combine iterative techniques within phases with waterfall-style formal signoffs between phases. The discipline continues to develop as underlying software engineering and formal methods techniques advance.

The next article in the series treats Silicon Valley from its defense-contracting origins, including the Stanford Industrial Park and the Terman relationship with the Department of Defense, the transition to commercial computing markets in the 1970s and 1980s, and the residual defense presence in contemporary Silicon Valley.

References

Books

- Leveson 1995
- Leveson 2011

Reference

- Astrée Analyzer
- MISRA C
- RTCA
- SCADE

Related Posts

- A237 Framing and the Co-Development Mechanism
- A241 Aerospace Simulation and Real-Time Systems
- A242 The Apollo Guidance Computer
- A243 ARPANET and Networking Origins
- A244 Space Shuttle Software as Engineering Landmark

Research

- Chilenski and Miller 1994
- Clarke and Emerson 1981
- Cousot and Cousot 1977
- Delmas and Souyris 2007
- Halbwachs et al. 1991
- Leveson 2004
- Leveson and Turner 1993
- Lions et al. 1996
- McCabe 1976