Software Engineering Institute
Carnegie Mellon University

FAA RESEARCH PROJECT ON SYSTEM COMPLEXITY EFFECTS ON AIRCRAFT SAFETY: IDENTIFYING THE IMPACT OF COMPLEXITY ON SAFETY

Sarah Sheard, Chuck Weinstock, Michael Konrad, and Donald Firesmith

July 2015

Executive Summary

This draft report organizes our work on this topic to date. The work will continue. Hence, the bulk of this report is an unfinished appendix. We have organized this work on the impact that software complexity has for aircraft safety by looking in two directions. Backward tracing asks the question, “What aspects of safety, V&V, and certification can be complicated by complexity?” Forward tracing asks the question, “What about complexity can lead to problems with certification, V&V, and flight safety?”

Backward tracing

Flight safety and assurance: Flight safety depends on assurance. We define *assurance* as justified confidence that a system will function as intended in its environment of use. Confidence is justified only if there is believable evidence supporting that confidence. A study by the National Research Council promotes the use of a carefully constructed assurance case to augment testing when exhaustive testing is infeasible or too costly. The case establishes a relationship between the tests (and other evidence) and the properties claimed, and is useful for supporting qualification (and requalification) decisions, managing resources and activities, and estimating the impact of design and requirements.

Quantification of assurance: Standards for safety-critical systems such as DO-178B/C focus on specifying qualification criteria. Criticality levels are identified for different system components, and qualification criteria are assigned to each—a larger set and more stringent criteria for higher criticality levels. The underlying assumption is that, by satisfying these criteria, software will have reached a level of quality sufficient to be an acceptable risk. These criteria can be mapped into an assurance case framework.

Cognitive aspects of certification and V&V: An assurer or certifier needs to establish and sustain an adequate-fidelity mental representation of the aircraft system and its environment sufficiently long and flexible to enable answering questions about what state the aircraft is in, what responses it produces, how parts interact, what it does to the environment (control), and how, in turn, the environment reacts. There is a question about whether something can be too complex to establish an adequate mental representation of the situation. Such a mental representation will likely be tool-aided, but then there

is a tradeoff: understanding what the tool does is necessary, but such understanding requires more learning.

Forward tracing

Later identification of problems: Identifying problems later in the software development life cycle leads to costlier repairs. Complexity makes it difficult to identify issues early in the life cycle because it multiplies or obscures interactions between components. This is important because a number of studies have shown conclusively that the later in the life cycle an error is discovered, the more difficult or costly it is to rework (rework of requirements errors alone makes up 78% of total rework cost). Data from these studies shows that 70% of all errors are introduced during requirements, system design, and software design, while 80% of the errors are not discovered until integration testing or later.

Invalid or incomplete assumptions about use or environment: System engineers are concerned about the interface between the system and its operator, and about the interaction of the system with its operational environment. In many cases, the system consists of a system under control and a control system that monitors, controls, and manages the operation of the system. In the process of specifying the requirements for the system, assumptions are made about the way operators interact with the system and about the operational environment. Increased complexity makes it difficult to determine all of the possible use cases, leading to more opportunities for human error. Similarly, assumptions are made about the physical system under control when specifying the requirements for the control system. These assumptions may not always be valid and may be violated over time due to changes in the operational context or in the system itself.

Concurrency and redundancy: Application software is integrated into a runtime architecture. Typically, the runtime architecture has concurrently executing and communicating tasks and supports multiple operational modes, with different modes involving different subsets of active tasks and communication channels. This can lead to race conditions in the way tasks interact and to a nondeterministic sequence of actions. Problems such as these can result from the application software making assumptions about the runtime environment. The assumptions may be violated when migrating to different runtime systems and computer hardware, such as multicore processors.

Scheduling real-time systems: Embedded applications may process time-sensitive data and may process data in a time-sensitive manner. For example, a control system makes assumptions about the latency of a data stream from a sensor to an actuator. Differences in the task execution and communication timing of different runtime architectures, and their particular hardware deployment, can affect latency, sampling jitter, and latency jitter.

Safety, reliability, and security: Safety-critical systems have reliability, safety, and security requirements, all of which become more difficult to achieve and guarantee as complexity increases. These requirements are typically addressed as part of system engineering. Understanding the role of software in reliability and safety has been a challenge, largely addressed to date by assuming perfectible software. Safety-critical systems contain both mission software and fault management software. Errors in the fault management logic make up as much as 60% of all system errors.

Managing complexity through abstraction: Current practice in reliability improvement of software focuses primarily on finding and removing bugs through review and testing. A complicating factor is that embedded software executes as an interacting set of concurrent tasks that operate on time-sensitive data and events and may encounter race conditions, unexpected latency jitter, and unanticipated resource contention, which appear to occur randomly and are difficult to test for in a systematic manner. Given the exponential growth in software size and interaction complexity, the concept of exhaustive testing has often turned into testing until the budget or schedule has been exhausted.

Migration toward states of higher risk: Systems tend to migrate toward states of higher risk. There are several reasons for this trend. First, the operational environment impacts the safety of systems. Second, compared to hardware, software experiences rapid design evolution and requires managed interaction complexity in order to limit the impact of such changes. Third, risk is increased by changes in the process, practices, methods, and tools used.

Limited confidence in modeling and analysis results: Model-based engineering is considered a key to improving system engineering and, specifically, embedded software system engineering. Engineers have practiced modeling, analysis and simulation for a number of years. Now there is a need in software engineering for an architecture-centric approach to reference models as a common source for system analysis and system generation.

Summary

Current best practice—which relies on process standards and safety culture—is unable to cope with the exponential growth in size and interaction complexity of embedded software in today’s increasingly software-reliant systems. Traditional reliability engineering has its roots in hardware, assumes slowly evolving system designs, and focuses reliability metrics on physical wear. During system design, reliability growth is achieved by having modeling and analysis identify failure modes as they become apparent. In contrast, software failure modes are primarily design errors, and software is often assumed to be perfectible and to show deterministic behavior. Software faults are addressed through design and code reviews as well as testing.

Moving beyond textual specification of requirements is essential to validating completeness and consistency of such specifications. Understanding the impact of architectural decisions, and identifying potentially mismatched assumptions in system interactions on nonfunctional mission and safety-critical requirements early in the life cycle, requires architectural models with well-defined semantics that support analysis of such system properties. Software-induced hazards must be taken into account in system safety analysis. Fault management has increasingly become the responsibility of the embedded software, requiring increased scrutiny in order to achieve reliability and safety goals. The complexity and the nondeterministic nature of software interaction requires formal static analysis methods to complement testing, with consistency across analysis models.

Future work

For the rest of this fiscal year, we will finish previously open tasks. We need to establish a working relationship with FAA technical experts who can help analyze the effect of complexity on safety and begin technical exchange as mentioned in the plan.

Our first task beginning in FY2016 is Task 3.5, "Quantify the Effects of System Complexity on Aircraft Safety and Identify Relevant Complexity Measures." We will prepare for this task in the remainder of FY2015 by reviewing available data sources and tracing the concept of complexity relevant to existing standards.

Impact of Complexity on Safety

To date on this task we have looked at the impact of complexity on safety from two viewpoints: (1) from the forward-looking viewpoint, asking "When complexity occurs, what happens that can affect safety?" and (2) from the backward-looking viewpoint, asking "In order for safety to be compromised, what had to happen, and how is that related to complexity?" We have documented substantial thought in these two sections, and plan to complete the work in our final report. In particular, we still need to tie the first viewpoint to the specific causes and effects of complexity from our first report [Konrad 2015].

Additional perspectives must be taken for a topic as broad as complexity. We have identified the following three views as important:

1. *Life-cycle view* (also *Process view*). In this view we plan to look at the kinds of complexity that become apparent during the various life cycle phases, including early system concept definition, requirements development, system and software architecture and high-level design, coding (including reviews and low-level testing), and verification and validation (V&V) phases. We have not yet determined the kinds of complexity appearing in each life cycle phase that are most germane to assurance, certification, and safety. We expect that a few kinds will dominate.
2. *Technology view*. In this view we plan to look at the new kinds of technology being implemented in a system, both new small apps and any significant architectural redesign or restructuring (e.g., what might happen in response to a new security threat). Some technology may increase complexity (e.g., if it increases interoperability), while some may simplify (e.g., if it isolates portions of the system from other portions, with a known communication gateway).
3. *Assurance-case, certification-case,* and *safety-case* views. These we have addressed for the most part in Appendix A.

Finally, we are looking at how to measure complexity, starting with [Nichols 2015]. To the software complexity measures from the literature, as discussed there, we will add system complexity measures. Our current thinking is that uncertainty due to complexity propagates and, when systems get too complex, the uncertainty dominates "knowability," and the system cannot be assured.

Future work

For the rest of this fiscal year we will finish up previously open tasks including those mentioned above. We need to establish a working relationship with FAA technical experts and begin technical exchange as mentioned in the plan.

Our first task beginning in FY2016 is Task 3.5, “Quantify the Effects of System Complexity on Aircraft Safety and Identify Relevant Complexity Measures,” described as:

> *Quantify the effect of system complexity on system safety, as possible using available data sources. Identify relevant measures of complexity for avionics systems and select measures that are of particular interest for system validation and certification. Show how the metrics relate (e.g., how the code quality impacts the system certification) to existing standards for avionics system development (e.g., DO-178C, ARP-4754A, ARP-4761A).*
>
> *Deliverables:*
> *White paper containing final selection of complexity and safety measures as well as analysis and quantified contributions of the various types of complexity to system safety.*

We will prepare for this task in the remainder of FY2015, by reviewing available data sources and tracing the concept of complexity relevant to existing standards.

Conclusion

This Task 3.4 report exposes our mid-term thinking on the effect of complexity on assurance, V&V, and flight safety to the FAA. A broad array of topics have been identified to date. While some thoughts about how complexity affects flight safety are well developed, our writings on the topics that are still the subject of active research are not yet mature; complete documentation will be included in the final report. Comments related to this draft report will also be considered in the context of the final project report.

Appendix

This appendix provides the current state of the content for the draft report being worked on in Task 3.4: Identify the impact of complexity on safety. As “complexity” and “safety” are both broad concepts, and the connections are varied, as shown in Figure 1, the conclusions will need to be prioritized.

As a first attempt to organize such an inquiry, we have looked at the many ways in which complexity could affect safety and assurability (or certifiability, for an aircraft). We have looked in two directions. Backward tracing asks the question, “What aspects of safety, V&V, and certification can be complicated by complexity?” Forward tracing asks the question, “What about complexity can lead to problems with certification, V&V, and flight safety?” This report includes considerations of both views.

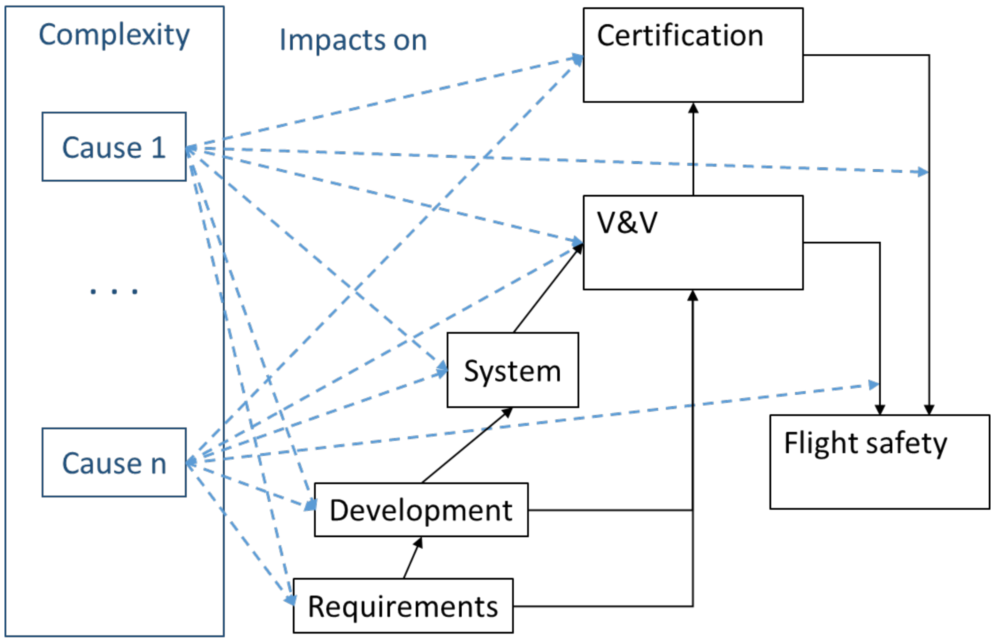

Figure 1: Impacts of Complexity on Assurance and Safety

A.1 Backward tracing

What aspects of safety and assurance can be hurt by complexity?[1]

A.1.1 Flight safety depends on assurance

We define assurance to be justified confidence that a system will function as intended in its environment of use. When we dissect this seemingly simple definition into its component parts, we find that it is actually quite complex, as shown in Figure 2. Confidence is justified only if there is believable evidence supporting that confidence. A system functions as intended only if it does what its ultimate users intend for it to do as they are actually using it, even though usage patterns will differ among individual users. It also functions as intended only if it properly mitigates the possible causes (intentional or unintentional) of critical failures to minimize their impact. Finally, a systems environment of use includes the actual environment of use, not just the intended environment of use. A shutdown system might work perfectly at sea level but totally fail 5,000 feet under the surface where it is actually being used.

1 Much of Section A.1 is modified from [Weinstock 2009] and [Feiler 2010].

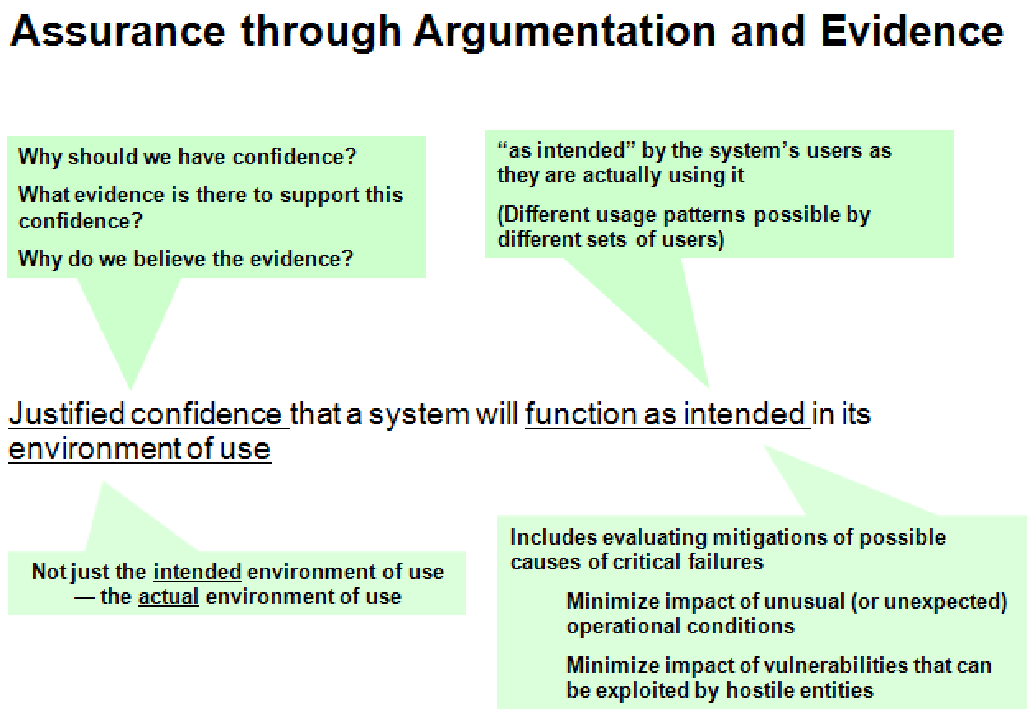
Assurance through Argumentation and Evidence
Why should we have confidence?
What evidence is there to support this confidence?
Why do we believe the evidence?
"as intended" by the system's users as they are actually using it
(Different usage patterns possible by different sets of users)
Justified confidence that a system will function as intended in its environment of use
Not just the intended environment of use — the actual environment of use
Includes evaluating mitigations of possible causes of critical failures
Minimize impact of unusual (or unexpected) operational conditions
Minimize impact of vulnerabilities that can be exploited by hostile entities

Figure 2: Assurance Through Argumentation and Evidence

How do we achieve this justified confidence? Historically we have relied on rigor in the development process and on extensive testing. However, a study by the National Research Council (NRC) titled "Dependable Systems: Sufficient Evidence?" says that testing and good development processes, while indispensable, are not by themselves enough to ensure high levels of dependability [Jackson 2007]:

> *… it is important to realize that testing alone is rarely sufficient to establish high levels of dependability. It is erroneous to believe that a rigorous development process, in which testing and code review are the only verification techniques used, justifies claims of extraordinarily high levels of dependability. Some certification schemes, for example, associate higher safety integrity levels with more burdensome process prescriptions and imply that following the processes recommended for the highest integrity levels will ensure that the failure rate is minuscule. In the absence of a carefully constructed dependability case, such confidence is misplaced.*

Such a dependability case augments testing when exhaustive testing is infeasible or too costly. The case establishes a relationship between the tests (and other evidence) and the properties claimed.

What the NRC report calls a dependability case, the community at large is calling an assurance case, or when used to show safety, a safety case. Under any of these names, it is somewhat similar to a legal case. In a legal case, there are two basic elements. The first is evidence, such as witnesses, fingerprints, or DNA. The second is an argument given by the attorneys as to why the jury should believe that the evidence supports (or does not support) the claim that the defendant is guilty (or innocent). A jury presented with only an argument that the defendant is guilty, with no evidence that supported that argument, would certainly have reasonable doubts about the defendant's guilt. A jury presented with evidence without an argument explaining why the evidence was relevant would have difficulty deciding how the evidence relates to the defendant.

The goal-structured assurance case is similar. Affirming evidence is associated with a property of interest (e.g., safety), attesting that it fulfills its claim. For instance, test results might be collected into a report. Without an argument as to why the test results support the claim of safety, an interested party could have difficulty seeing its relevance or sufficiency. With only a detailed argument that depends on test results to show that a system was safe, but does not provide those results, again it would be hard to establish the system's safety. So a goal-structured assurance case, as shown in Figure 3, specifies a claim (or goal) regarding a property of interest, evidence that supports that claim, and a detailed argument explaining how the evidence supports the claim.

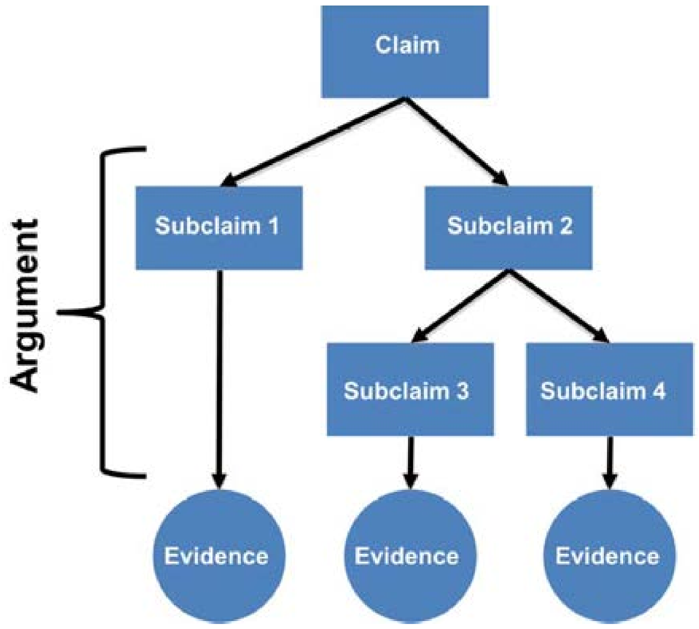

Figure 3: A Goal-Structured Assurance Case

The goal-structured assurance case [Kelly 2004] has been used extensively outside of the United States for a number of years to assure the safety of nuclear reactors, railroad signaling systems, avionics systems, and so forth. Assurance cases are now being developed for other attributes (e.g., security of a software supply chain [Ellison 2008]) and other activities (e.g., acquisition [Blanchette 2009]).

As evidenced by the NRC report, there is increasing interest in assurance cases in the United States. International Standards Organization (ISO) standard (15026-2) for assurance cases is now under development. The U.S. Food and Drug Administration (FDA) recently began to suggest their inclusion in regulatory submissions [FDA 2010].

In the best practice, an engineering organization will develop an assurance case in parallel with the system it assures. The case's structure will be used to influence assurance-centered actions throughout the life cycle. The co-development of the assurance case has several important advantages:

- It can help determine which claims are most critical and, hence, what evidence and assurance-related activities are most needed to support such claims.
- It can help guide design decisions that will simplify the case and, thus, make it easier to develop a convincing argument that important properties have been met.
- It serves as documentation as to why certain design decisions have been made, making it easier to revisit these decisions should the need arise, helping to communicate engineering expertise, and allowing for more efficient reuse in subsequent systems.
- It can help management make a more accurate determination of whether the development is on track to produce a system that meets its requirements.
- It can modify development so that identified hazards are reduced, when such design changes are much easier and less costly than later.

Whether co-developed or not, the resulting product is useful for supporting qualification (and requalification) decisions, managing resources and activities (by showing which activities have the most payoff for claims of particular importance), and estimating the impact of design and requirements changes (by showing which portions of the case may be affected).

A.1.2 Requirements and claims

There are basically two approaches for structuring an assurance case: (1) focus on identifying requirements, showing that they are satisfied or (2) focus on hazards to fulfilling those requirements, showing that the hazards have been eliminated or adequately mitigated. The approaches are not mutually exclusive—to show that a requirement is met, one often has to show that hazards defeating the requirement have been eliminated or mitigated—but each has a different flavor. Each type has a role to play in developing an assurance case.

This section addresses the first approach. The second approach is discussed in Section A.1.3.

Because developers are used to stating nonfunctional requirements (e.g., safety, availability, performance) and then ensuring that they are satisfied, top-level claims in an assurance case often have a requirements flavor (e.g., "X is safe," which might be decomposed into sub-claims that the "X is electrically safe," "X is safe to operate," etc.). Typically, these nonfunctional requirements arise from an understanding of hazards that need to be addressed; each such requirement, if satisfied, mitigates one or more hazards. But if the case only addresses derived requirements whose satisfaction implies safety, (e.g., "timeout is 5 seconds"), the link to the hazards mitigated by the requirement can be lost; it can become difficult to decide if the requirement is adequate to address the underlying hazard(s).

To see how a focus on requirements can obscure underlying hazards, let's consider an example. Suppose we have a safety-critical system that must monitor its operational capability and can either run connected to an electrical outlet or using a battery. An obvious hazard is loss of battery power; one might therefore state a safety requirement aimed at helping to ensure that the system is plugged into an electrical power source prior to battery exhaustion. Such a requirement might be worded as follows:

> *When operating on battery power, visual and auditory alarms are launched at least 10 minutes prior to battery exhaustion but no more than 15 minutes prior.*

To demonstrate that this claim holds for a particular system, we could provide test results showing that warnings are raised at least 10 minutes prior to battery exhaustion but no more than 15 minutes prior. In addition, we could present arguments showing that we have confidence in such test results because the structure of the tests has taken into account various potential causes of misleading results. For example, since the battery discharge profile changes depending on the age of a battery, we would need to show that all the tests were run with both new and well-used batteries. Similarly, since the electrical load might affect the time to battery exhaustion, we would need to show that the tests were run with different electrical loads on the system.

We can represent such a safety requirement as a claim in an assurance case together with the evidence and other arguments needed to show that the requirement is satisfied (see Figure 4). One set of evidence is the aforementioned testing results. However, we can increase confidence in the validity of the claim by also showing that pitfalls to valid testing have been adequately mitigated. Going a step further, our confidence in the validity of the claim would also be increased by an argument asserting the accuracy of the algorithm used to estimate remaining battery life. The combination of testing results and algorithm analysis makes the case stronger than if just test results alone were presented. To support the algorithm-accuracy claim, a developer might reference design studies and literature, as well as an analysis of the actual design.

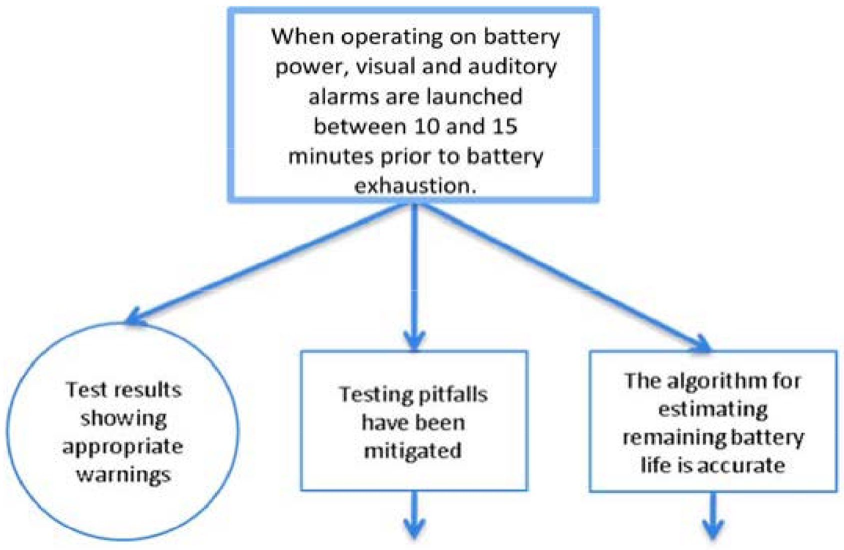
When operating on battery power, visual and auditory alarms are launched between 10 and 15 minutes prior to battery exhaustion.
Test results showing appropriate warnings
Testing pitfalls have been mitigated
The algorithm for estimating remaining battery life is accurate

Figure 4: Confirming That a Safety Requirement Has Been Satisfied

Such tests and analyses are fine for demonstrating that the requirement is satisfied. But from a safety viewpoint, we have little documentation about what hazard the requirement is mitigating. In addition, how do we know that 10 minutes is the appropriate warning interval for every setting? Is 10 minutes enough time for someone to respond to the alarm? Will the alarm be heard in every possible setting? How accurate does the measure of remaining power need to be (e.g., is it unacceptable if the alarms are launched when 20 minutes of power remains)? How does this requirement fit with other safety requirements? In short, to fully understand and validate the requirement, we need to establish the larger context within which the requirement exists.

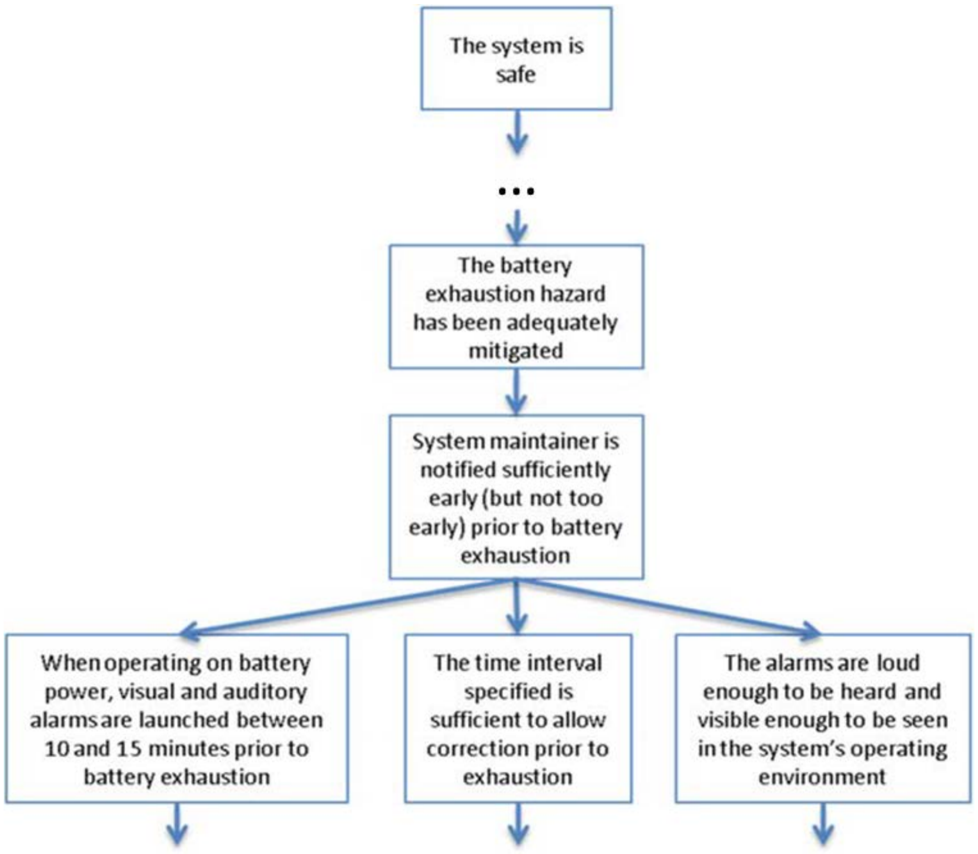
The system is safe
…
The battery exhaustion hazard has been adequately mitigated
System maintainer is notified sufficiently early (but not too early) prior to battery exhaustion
When operating on battery power, visual and auditory alarms are launched between 10 and 15 minutes prior to battery exhaustion
The time interval specified is sufficient to allow correction prior to exhaustion
The alarms are loud enough to be heard and visible enough to be seen in the system's operating environment

Figure 5: *Context for Raising an Alarm About Impending Battery Exhaustion*

Figure 5 presents claims related to a battery-exhaustion warning system and the context for such claims.

Directly below the first statement "The system is safe" we have removed, for simplicity's sake, reference to other safety hazards requiring mitigation that would normally appear beneath such a claim. We also state that any hazard of system shutdown due to battery exhaustion has been mitigated. With these matters settled, we proceed to the timing considerations that surround raising an alarm that warns of impending battery exhaustion.

The proposed mitigation for battery exhaustion is to notify a system maintainer in a timely manner that the battery is about to become exhausted. This is shown in the case by making the claim of notifying the maintainers "sufficiently soon" but not "too soon." We are now in a position to state the safety requirement about when an alarm needs to be raised. In addition, we can now readily deal with other hazards not addressed directly by the safety requirement; namely, we can consider whether the warning time is sufficient to allow human response and also whether the alarm is sufficiently noticeable that the human will be unlikely to ignore it.

Taken altogether, the exhaustion mitigation and notification claims establish the context and validity of what was originally an isolated safety requirement. Whether all this argumentation is necessary depends on the importance of dealing with battery exhaustion and the extent to which there is a standard way of dealing with it. Less argument (and evidence) is needed to support mitigations of less important hazards or where there is consensus on adequate ways of addressing a particular hazard.

A benefit of focusing on safety requirements is that stating the safety requirements and demonstrating that they have been met seems straightforward from a user viewpoint. But a safety assurance case that only addresses whether safety requirements are met will focus primarily on what tests and test conditions or other analyses are considered sufficient to show the requirements are met. The case is likely to be less convincing when it does not deal explicitly with all relevant hazards (i.e., when the reasoning leading from the hazards to the requirement is not part of the case).

Another problem with a pure requirements-based approach is that it can be difficult to specify fault-tolerant behavior. For example, consider a high-level requirement such as “The system does X.” Satisfying this requirement would certainly seem to satisfy a higher level claim that the system is safe. But the requirement, as stated, implies that the system always does X, and there may be factors outside the system’s control that can prevent this from happening. From a safety viewpoint, we want to ensure the system minimizes the chances of becoming unsafe. Stating a claim that is unachievable in the real world doesn’t allow the case to adequately address safety hazards and their mitigations.

From a safety argument perspective, instead of focusing on safety requirements, per se, it is more convincing to state (and satisfy) hazard mitigation claims. For example, a claim such as “The possibility of not doing X has been mitigated” allows the assurance case to discuss the possible hazards to doing X and then to explain the mitigation approaches, which can include raising alarms to cause a human intervention.

A.1.3 Assurance over the life-cycle

Evidence-based arguments start with a single claim and then build an argument out of subclaims at multiple levels. Eventually the lowest level claims are supported by evidence, and the end result is that the high-level claim is substantiated. The nature of the argument and the nature of the evidence will necessarily change as development moves through the different stages of the life cycle. At early stages, an argument consisting of broad strokes supported by informal “hand waving” evidence will allow design decisions to be made and development to continue. As the development continues, the arguments needed to allow continued development become significantly more detailed, and the supporting evidence becomes much more precise.

As an example of the above, consider a system that has a requirement to restart within one minute of a system failure. Early in the life cycle, designers are faced with making decisions on how to best meet this requirement. Obvious choices include hot standby, warm restart, and cold restart. Each has its costs and benefits, and tradeoffs must be made.

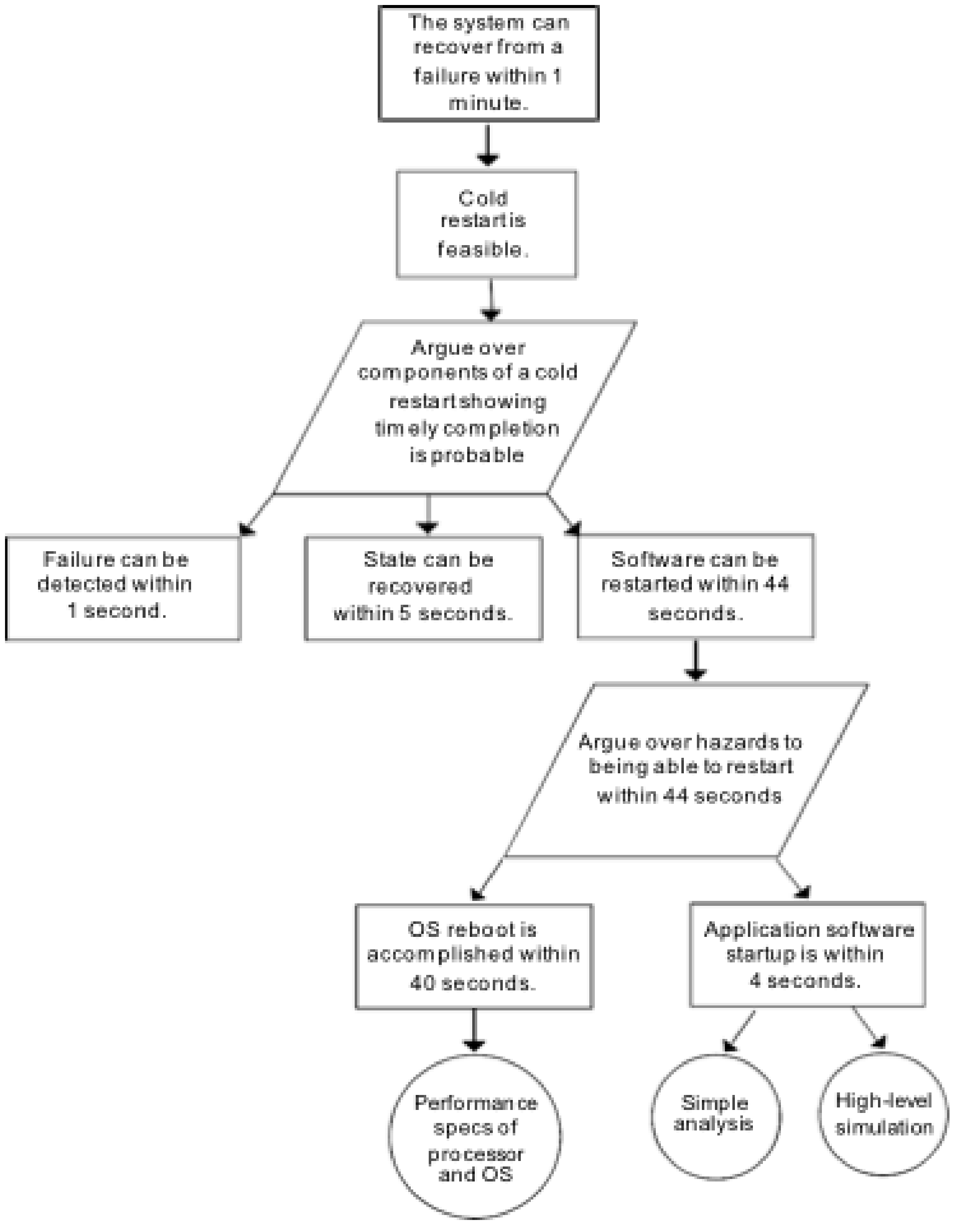
The system can recover from a failure within 1 minute.
Cold restart is feasible.
Argue over components of a cold restart showing timely completion is probable
Failure can be detected within 1 second.
State can be recovered within 5 seconds.
Software can be restarted within 44 seconds.
Argue over hazards to being able to restart within 44 seconds
OS reboot is accomplished within 40 seconds.
Application software startup is within 4 seconds.
Performance specs of processor and OS
Simple analysis
High-level simulation

Figure 6: An Assurance Case in Early Design

Figure 6 shows a partial assurance case for such a design. Only the case for cold restart has been expanded because that alternative has proven, at this early stage, to likely be able to accomplish the goal of restarting within one minute. If there were any question about this goal being met, the other alternatives would have also been expanded to enable a more informed decision. An assurance case for the same system later in the life cycle would be both simpler (rejected alternatives have been removed from the case) and more complex (the analysis of the cold restart alternative is supported by additional evidence) than here. As the system is developed further, the case is augmented with actual test results as evidence, as well as more details—simulations, AADL models, and so on.

We expect to see much more detailed evidence as the development of the system continues. Thus it is important to develop and maintain the assurance case in parallel with the system being assured. This has a multitude of benefits, including:

- An assurance case fully documents the system being developed, leading to more confidence in the quality of the system and making it more likely that the design will be understood as new personnel are brought onboard.
- An assurance case developed in parallel with development of the system can lead to more insight into system quality earlier in the development cycle and can take less expensive corrective action if problems begin to surface. Identified hazards can be mitigated with design corrections.
- The opportunities for reuse of a design documented with an assurance case are significantly greater than for one developed without it. This is especially true if assurance case patterns are used. An assurance case pattern is a template that captures acceptable ways of structuring generic arguments [Kelly 1998].

A.1.4 Quantification of assurance

Standards for safety-critical systems such as DO-178B/C focus on specifying qualification criteria. Criticality levels are identified for different system components, and qualification criteria are assigned to each—a larger set and more stringent criteria for higher criticality levels. The criticality level is determined by (1) the cause of a software component's anomalous behavior or (2) that behavior's contribution to the failure of a system function that results in system hazards and failure conditions of varying severity. Five severity levels combined with likelihood of occurrence (expressed qualitatively as likeliness of occurrence or quantitatively as probability of occurrence) act as a system safety management decision matrix [Boydston 2009].

The underlying assumption is that, by satisfying these criteria, software will have reached a level of quality sufficient to be an acceptable risk. These qualification criteria are a combination of de- sign- and implementation-related criteria, as well as development-process- and qualification-process- related criteria. Examples of system- and implementation-related criteria are requirement specification and design documentation guidance, and coverage ranging from dead code to Modified Condition/Decision Coverage (MC/DC). Examples of qualification-process-related criteria are requirements traceability, and correct implementation and application of test suites.

These criteria can be mapped into an assurance case framework, and we will use such a frame- work to drive a qualification-evidence metric. This is conceptually illustrated in Figure 7. Claims represent qualification criteria on the system and its subsystems, that is, requirements that must be satisfied by the system design and implementation as shown by the evidence. The operational context and the assumptions for each claim are documented. The evidence takes the form of a V&V activity, ranging from review and testing to formal analysis. The process of producing the evidence has its own set of claims and evidence, such as validity of the model or test case implementation and correct application of the evidence-generating method. The risk-level annotations reflect the criticality levels of different subsystems and different requirements on those subsystems. Whether the evidence for meeting the qualification criteria is sufficient is reflected in the argument and represents a risk assessment by the qualification authority.

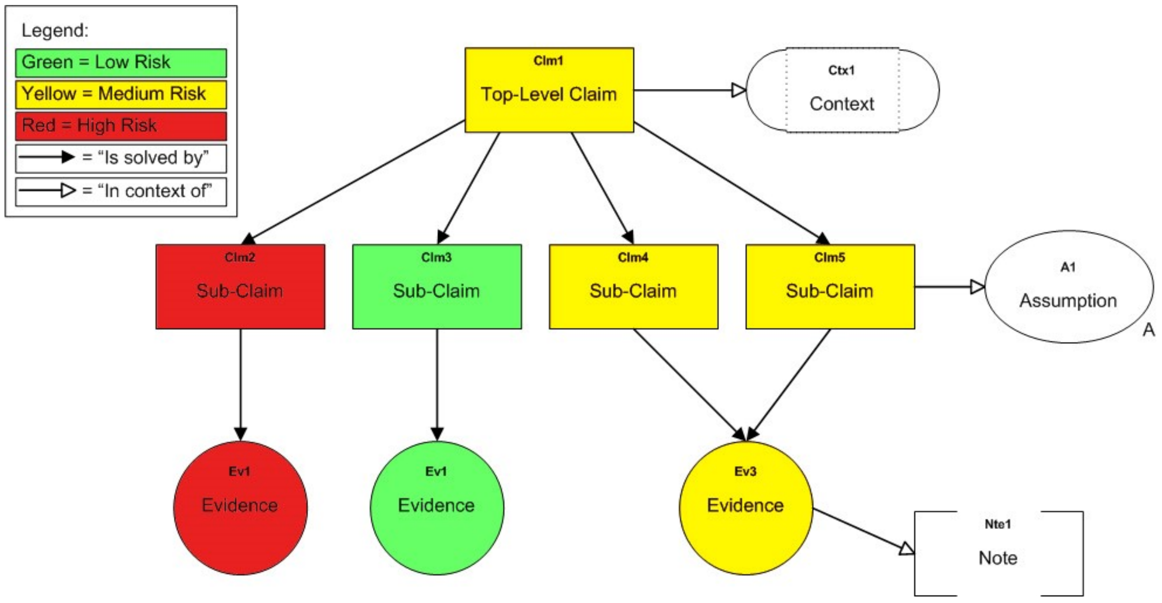

Figure 7: Qualification Evidence Through Assurance Cases

We have several contributors to the qualification-evidence metric. The first contributor focuses on the claim hierarchy. We identify the significance of each subclaim's contribution to a claim in order to reflect the potential impact of an unsubstantiated subclaim, and we weight it with the criticality of the subsystem for which the claim is made. We determine the degree of claim coverage by subclaims, using a technique similar to the one used for requirements decomposition coverage.

We take into account the context in which the evidence was produced (e.g., the assumptions about the operational environment made during the tests) when assessing the risk of deploying the sys- tem in various deployment scenarios.

The second contributor identifies the degree to which specific evidence contributes to the substantiation of a claim to reflect the impact that the lack of particular evidence has on the confidence in the claim. We take into account the assumptions made in the evidence-producing process (i.e., the fidelity of the model abstraction with respect to the actual system, the consistency of the model with respect to the actual system and other models, and the proper execution of the evidence-producing process). In this context we may also take into account work on the use of a strategy- based risk model to assess the impact of different steps in the validation process of Research and Development satellites in terms of expected risk [Langenbrunner 2010].

The third contributor reflects the effectiveness of the method used to produce the evidence. A de- fect removal efficiency metric is intended to reflect the effectiveness of specific validation meth- ods in discovering errors (i.e., to achieve fault prevention). Boehm, Miller, and Jones provide source examples for this metric [Madachy 2010, Miller 2005b, Jones 2010]. Rushby discusses an approach to probabilistically assess the imperfection of software in the context of software verifi- cation as part of system assurance [Rushby 2009]. We can consider incorporating this idea of probability of imperfection or uncertainty to claims and evidence.

A.1.5 Dependence of the above on complexity

Complexity in the system and its software contributes to problems with assurance and certification in a number of ways.

A.1.6 Cognitive aspects of certification and V&V

This section addresses what aspects of cognition are important to considerations of assurance, certification, and V&V. Although cognitive aspects are primarily out of scope for this project, there are some important considerations for system design, hazard identification, and the assurance process.

An assurer or certifier needs to establish and sustain an adequate-fidelity mental representation of the aircraft system and its environment sufficiently long and flexibly to enable answering questions about what state the aircraft is in, what responses is it producing, how parts are interacting, what it does to the environment (control), and how, in turn, the environment reacts. Such a model of how humans reason and evaluate a situation is key to understanding the qualities of the resulting judgment, as well as identifying causes of judgment failure, on the one hand, and how to improve its capability, on the other hand. Apart from biases, anchors, and things that compromise self-control (hunger, fatigue), there is a question about whether something can be too complex to even establish an adequate mental representation of the situation.

Such a mental representation will likely be tool-aided, but then there is a tradeoff: understanding what the tool does (and hopefully not how it does it) is necessary, but it also takes more learning. If that learning hasn't occurred already, then adds to the task of the assurer or certifier. If the learning has occurred, and the assurer or certifier has learned to trust the tool, then the tool and what it does both become part of the cognitive mechanism available to the assurer or certifier. If the tool is difficult to learn, then the amount of capacity needed to use the tool may exceed the capacity that its use frees up.

Note that one key challenge here is to know what can be simplified from the actual situation and yet retain sufficient essence (fidelity) to be able to reason correctly about the situation.

This knowledge may not be easy to obtain and as a result, especially in situations fraught with risk, the assurer or certifier may feel compelled either to retain high fidelity with the existing situation, adding to modeling effort because of a fear of the consequences of cutting any corners, or to start small, modeling at first smaller versions of the problem, and going on a path towards iteratively making it resemble more closely the actual situation. The challenge here is that the topology of solution spaces is not well enough known to be sure the that solution will have the same general shape on the next iteration as it has had on the last few iterations.

The following are types of information that the assurer or certifier needs to integrate into the mental representation in order to enable correct evaluation:

- Purpose for reasoning about the system
- System, its parts, interfaces, interactions, and behavior
- Artifacts from developing, verifying, and validating the system (with a caveat that emergent properties are not necessarily evident when looking at artifacts related to individual parts and connections)

- Domain and Environment, including their parts and interfaces.
- Novel technologies used in the aircraft or the environment that need to be considered, including their capabilities and limitations
- Tools that aid analysis (e.g., that evaluate impacts on valuable resources, or that track progress)
- Methods that we need to follow to make an appropriate evaluation (deductive, analytical, etc.) and their properties (scope, correctness, precision, false-positives, etc.)

A.2 Forward tracing

What about complexity can lead to problems with certification, V&V, and flight safety?

In this section we look from the point of view of the major causes of complexity and what effect those have on aspects of assurance and certification.

A.2.1 Current metrics and expanded metrics (metrics related to systems, including hardware; i.e. not just software)

Metrics that address how complex a system is were provided in the last report, [Nichols 2015].

- Explicitly discuss impact of component complexity and integration complexity
- Systems engineering related measures
- Computer hardware related measures applicable to avionics systems

A.2.2 Later identification of problems leads to costlier repairs

Complexity makes it difficult to identify issues early in the life-cycle because it multiplies or obscures interactions between components. This is important because a number of studies have been performed to determine where in the development life cycle errors are introduced, when they are discovered, and that the resulting rework cost is. They show conclusively that the later in the life-cycle an error is discovered the more difficult or costly it is to rework. We limit ourselves here to work by NIST, Galin, Boehm, and Dabney [NIST 2002, Galin 2004, Boehm 1981, Dabney 2003]. The NIST data primarily focuses on IT applications, while the other studies draw on safety-critical systems.

Figure 8 shows rework cost factors based on the Galin and Boehm studies [Galin 2004, Boehm 1981]. There are three data points for each development phase: % errors introduced in the phase; % errors discovered in the phase; and a rework cost factor normalized to the cost of repair in the phase. The rework cost figures include the cost of retest.

The percentages of errors introduced and discovered were quite consistent across the three studies. The figure shows that 70% of all errors are introduced during requirements, system design, and software design, while 80% of the errors are not discovered until integration testing or later. The 70% is evenly split between requirements errors and design errors. The figure shows that 20% of the errors are introduced during code development and unit testing and 16% of the errors are discovered in that phase – indicating that current practice is relatively good at addressing coding errors. In other words, the data shows that we need to do a better job at getting the requirements specified and at managing the interaction complexity between system components not only in terms of system functionality but also in terms of non-functional system properties. Note that the studies provide more detailed data in terms of leakage rates between phases.

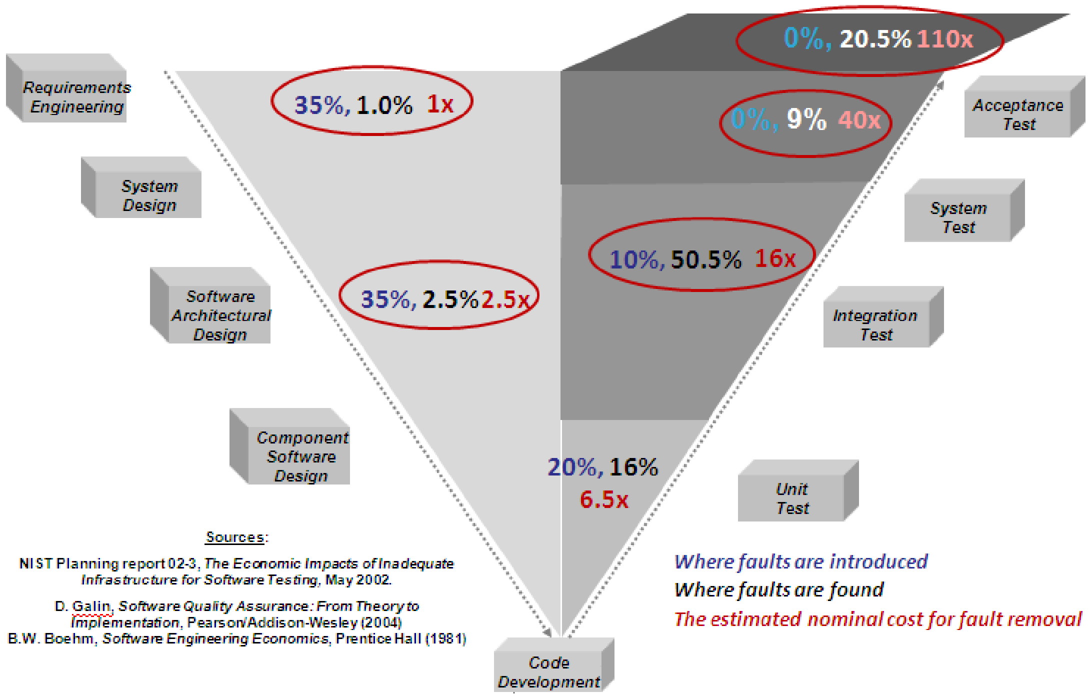
Requirements Engineering
System Design
Software Architectural Design
Component Software Design
Code Development
Unit Test
Integration Test
System Test
Acceptance Test
35%, 1.0% 1x
35%, 2.5% 2.5x
20%, 16% 6.5x
10%, 50.5% 16x
0%, 9% 40x
0%, 20.5% 110x
Where faults are introduced
Where faults are found
The estimated nominal cost for fault removal
Sources:
NIST Planning report 02-3, The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002.
D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson/Addison-Wesley (2004)
B.W. Boehm, Software Engineering Economics, Prentice Hall (1981)

Figure 8: Error Leakage Rates Across Development Phases

The rework cost for requirements errors alone make up 78% of the total rework cost. In other words, there is high leverage in cost reduction by focusing on earlier discovery of requirements and system design related errors. However, complexity makes such early discovery extremely difficult.

A.2.3 Invalid or incomplete assumptions about use or environment

System engineers are concerned about the interface between the system and its operator as well as the interaction of the system with its operational environment. In many cases, the system consists of a system under control and a control system that monitors, controls, and manages the operation of the system. In the process of specifying the requirements for the system assumptions are made about the way operators interact with the system and about the operational environment. Increased complexity makes it difficult to determine all of the possible use cases, leading to more opportunities for human error. Similarly, assumptions are made about the physical system under control when specifying the requirements for the control system. These assumptions may not always be valid and may be violated over time due to changes in the operational context or in the system itself. Several examples illustrate the point.

Operators are expected to have situational awareness of the operational environment with sometimes limited or misleading information or guidance. The result can by an accident as was the case in the Comair crash when one of the taxiways was under construction [Nelson 2008].

An example of assumptions made about the interaction between the operator and the system is an incident in which a subway train left the platform without the operator present in the operator console. One of the doors on the first car had difficulty closing. The operator stepped out of the operator cabin to close the door and the semi-automated system – sensing that all doors were closed – departed with the operator standing on the platform.

Similarly, assumptions exist when a control system is designed for a system under control. For example, assumptions are made about the lift generated by aircraft wings in determining the maximum load and in identifying the operational envelope. The interaction complexity between several system parameters can lead to violation of assumptions about system parameters and result in incidents or accidents. For example, Air France flight 447 [Spiegel 2010], which crashed on route from Brazil to France, was loaded to within 240kg of maximum capacity and its fuel consumption was based on a majority of the flight occurring at 39000 feet. At that altitude the operational speed for maintaining the required lift is quite narrow, increasing the risk of operating outside a safe flight envelope in non-nominal situations, such as severe turbulences due to storms. This example illustrates the challenge of understanding all system hazards and specifying appropriate safety and reliability requirements.

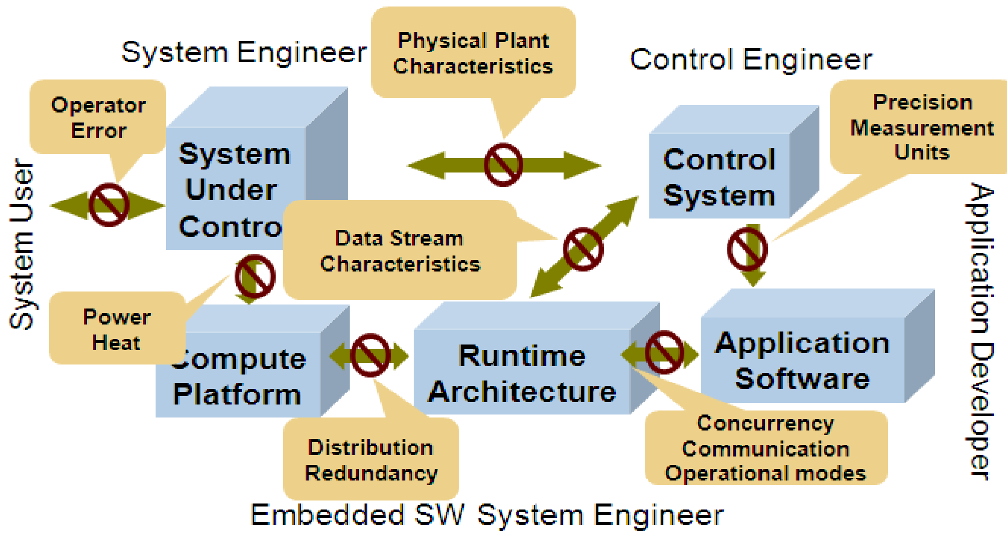

Figure 9: Interaction Complexity and Mismatched Assumptions with Embedded Software

The upper half of Figure 9 illustrates some of this interaction complexity and potential for mismatched assumptions. As these systems have become software-reliant new areas of interaction complexity and potential for mismatched assumptions are introduced, as shown on the lower half of Figure 9.

As system functionality is implemented in software, monitored and controlled variables in the environment are translated into input and output variables that the embedded software operates on. In the translation into software variables the data is often assumed to be expressed in a particular measurement unit, which may not have been communicated to the software engineer when system requirements were translated into software requirements. Similarly, the expected range of values and the degree of precision in which they are represented is affected by the base type chosen for the variable. For example, one of the contributing factors to the Ariane 5 accident [Nuseibeh 1997] was the use of a

16 bit integer variable, which could handle the range of values for Ariane 4, but resulted in negative values on Ariane 5 due to wrap-around.

A.2.4 Concurrency and redundancy

Application software is integrated into a runtime architecture. Typically the runtime architecture has concurrently executing and communicating tasks and supports multiple operational modes, with different modes involving different subsets of active tasks and communication channels. This can lead to race conditions in the way tasks interact and in a non-deterministic sequence of actions. This can result from the application software making assumptions about the runtime environment. For example, it may assume that two tasks may not require explicit synchronization because execution of both tasks on the same processor using a non-preemptive scheduling protocol ensures mutual exclusion. These assumptions may be violated when migrating to different runtime systems and computer hardware, such as multi-core processors. Apple experienced such an issue the first time it released a multi-core system: iTunes would run into a deadlock because of a scheduling error that did not show up on a single-core system because of the inherent serialization in that situation.

The computer platform is typically a distributed networked set of processors with redundancy to provide reliable system operation. This means that replicated instances of the embedded software execute on different processor instances and communicate over different network instances. A change in the deployment configuration of the embedded software may lead to replicated software components being allocated to the same physical processor and eliminating physical redundancy. Similarly, migration embedded software to a partitioned runtime architecture such as ARINC653 can result in reduced reliability if the mapping of embedded software to partitions and the binding of partitions to physical hardware is not performed consistent with the redundancy requirements for the system. Finally, virtualization leads to unplanned resource contention and slower than expected performance.

A.2.5 Scheduling real-time systems

Embedded applications may process time-sensitive data and may process data in a time-sensitive manner. For example, a control system makes assumptions about the latency of a data stream from a sensor to an actuator. Different control algorithms have various degrees of sensitivity to sampling jitter, which if exceeded can result in unstable control behavior [Feiler 2008]. Differences in the task execution and communication timing of different runtime architectures and their particular hardware deployment can affect latency as well as sampling and latency jitter.

Similarly, it is common practice to implement processing of events by periodically sampling a data variable whose value changes to represent the occurrence of an event and after a given duration changes back to its original value. Variation of sampling such a variable beyond a certain threshold results in the recipient missing the observation of an event, while the application logic assumes that all events are communicated to the recipient. Such an unanticipated loss of events can result in inconsistent system states and deadlock in system interaction protocols.

A.2.6 Safety, reliability, and security

Safety-critical systems have reliability, safety, and security requirements all of which become more difficult to achieve and guarantee as complexity increases. These requirements are typically addressed as part of system engineering. The reliability of a system and its components is driven by availability requirements and by the safety implications of failing system components. The FAA has introduced five levels of criticality and has associated reliability figures in terms of mean time between failures (MTBF). Typically the required reliability numbers are achieved through redundancy, e.g., through dual or triple redundancy in flight control systems.

Similarly, safety of a system is assured through a series of analyses that identifies hazards and their manifestation through Functional Hazard Analysis (FHA), followed by Preliminary System Safety Analysis (PSSA) and System Safety Analysis (SSA) for the top-level system design, and Common Cause Analysis (CCA) to identify system components that violate the independence of failure occurrence assumption of many reliability predictions, taking on the form of fault tree analysis (FTA), and failure mode and effects analysis (FMEA) as the system design evolves [SAE 1996, FAA 2000].

Experience has shown that such safety analysis must take into account interactions with operators and the operational environment as well as the development environment as sources of contributing and systemic hazards [Leveson 2004, Leveson 2005]. A key is the translation of the results of such safety hazards into safety requirements on the system and the validation of these requirements for completeness, consistency, and enforcement or management when violated. This has led to the formalization of requirements, often expressed as discrete state behavior and boundary conditions on physical system and environmental state [Parnas 1991, Leveson 2000, Tribble 2002] and their validation and verification through formal methods [Miller 2005a]. It has also led to an understanding that system safety and reliability are emergent system properties. System reliability can be achieved with unreliable components, and reliable system components do not guarantee system reliability or system safety [Leveson 2009]. Again, complexity makes it more challenging to build the models to be formally verified. It is easy to develop a model that is too abstract, that misses important details introduced by system complexity.

Understanding the role of software in reliability and safety has been a challenge and largely addressed by assuming perfectible software. Errors in software are design errors that present both reliability and safety hazards and must be treated as such. Software reliability cannot be addressed in terms of MTBF, because software errors either exist or they do not. A software function will always produce the same result for a given input. Interacting software may be sensitive to execution order and timing, but for a given execution order and schedule will produce the same result. Instead we need to understand the hazards introduced by possible errors in software and their impact on system reliability and safety. This requires an understanding of the roles of software in system reliability and safety. Embedded software may contribute to desired mission capability; to reliability by implementing fault management of physical system components, of the computer platform, and of the mission software; and to safety by monitoring for violation of safety requirements.

A.2.7 Fault management and “rainy day” testing

Safety-critical systems contain mission software and fault management software with errors in the fault management logic making up as much as 60% of all system errors. In other words, the system portion responsible for the reliable operation is itself unreliable. One reason for this is limited understanding of faults and hazards of software vs. hardware due to assumptions made about the operational environment, system components, and system interactions. A second reason is the interaction complexity in systems – particularly interaction complexity of the embedded software. A third reason is the challenge of testing the fault management portion of a system. Fault management software is only exercised when the system fails, and fault management errors are only triggered when the system is already dealing with an erroneous condition (a “rainy day scenario”). There is a need for explicitly specifying and validating safety and reliability requirements as well as hazards and assumptions, and architecting fault mitigation into the system in a traceable and verifiable manner.

A.2.8 Managing complexity through abstraction

Current practice in reliability improvement of software is primarily focused on finding and removing bugs through review and testing. Testing focuses on exercising the code with various inputs to ensure that all code statements execute as expected and produced expected results. Many of the test coverage approaches reflect the assumption that software behaves deterministically, i.e., for given inputs the software executes the same statements and produces the same result. Black box testing focuses on mapping sets of input data to expected output data, and white box testing focuses on exercising the program logic reflected in the source code statements. Since there are many potential interactions between source code statements, programming language abstractions have been introduced to manage this complexity. These include data abstraction and object orientation, strong typing, modularity with well-defined interfaces (Ada being an excellent example of a programming language for reliable software), and restrictions such as static memory allocation and no application-level manipulation of pointers (as found in high-integrity subset profiles of programming languages such as the Ada Ravenscar profile). Despite these capabilities, the challenge is to find test coverage approaches that bound the amount of necessary testing.

Standards for safety-critical systems, such as DO-178B/C, provide guidance on the degree of coverage necessary for software with different levels of criticality. For example, the most critical (Level A) software, which is defined as that which could prevent continued safe flight and landing of an aircraft, must satisfy a level of coverage called *Modified Condition/Decision Coverage* (MC/DC), while in other cases Decision Coverage (DC), branch coverage, or statement coverage is sufficient. Quoting the FAA Certification Authorities Software Team (CAST) [FAA 2010]:

> *The issue is that at least some industry participants are not applying the “literal” definition of decision. I.e., some industry participants are equating branch coverage and decision coverage, leading to inconsistency in the interpretation and application of DC and MC/DC in the industry. Tool manufacturers, in particular, tend to be inconsistent in the approach, since many of them come from a “traditional” testing background (using the IEEE definitions), rather than an aviation background.*

A complicating factor is that embedded software executes as an interacting set of concurrent tasks that operate on time-sensitive data and events, and may encounter race conditions, unexpected latency jitter, and unanticipated resource contention, which appear to occur randomly and are difficult to test for in a systematic manner. Given the exponential growth in software size and interaction complexity, the concept of exhaustive testing has often turned into testing until the budget or schedule has been exhausted.

A.2.9 Migration toward states of higher risk

Leveson observes that systems will tend to migrate toward states of higher risk [Leveson 2009]. There are several reasons for this trend. First, the operational environment impacts the safety of systems. For example, the system may be deployed in new operational settings that may introduce new hazards and may violate assumptions made about the operational environment. Similarly, changes to operational procedures and guidelines, whether as result of operational budget reductions or as work around to compensate for known and correctable system faults may also contribute to increased risk.

The second reason for this trend towards higher risk is that software corrections and changes are design changes. Compared to hardware, software experiences rapid design evolution and requires managed interaction complexity in order limit the impact of such changes. In particular, the addition of new mission capability and operational features is a common occurrence since software can be easily changed. Similarly, technology upgrades to the computer system, such as migration from deterministic network protocols in a federated architecture to a publish-subscribe paradigm on top of a high-speed Ethernet with non-deterministic network protocols, or the introduction of multi-core processors, impact the embedded application software. This results in unintended feature interactions and violation of assumptions due to paradigm shifts. A study of 15 operating system releases from ten vendors [Koopman 1999] shows that failure rates stay at a high rate and may even be increasing over multiple major releases. The result is a failure density curve across multiple releases, whose shape is illustrated categorically in Figure 10.

A third reason is changes in the process, practices, methods, and tools used in the development of embedded software. For example, although Ada has shown to be an excellent choice for the development of highly reliable software, today’s market place demands the use of Java due to the fact that Ada programming skills are scarce and Java programming skills are plentiful. Similarly, object-oriented design methods based on UML were originally not developed with safety-critical embedded software systems in mind; retro-fitting such notations to address real-time, reliability and safety concerns is an ongoing slow process with its pitfalls.

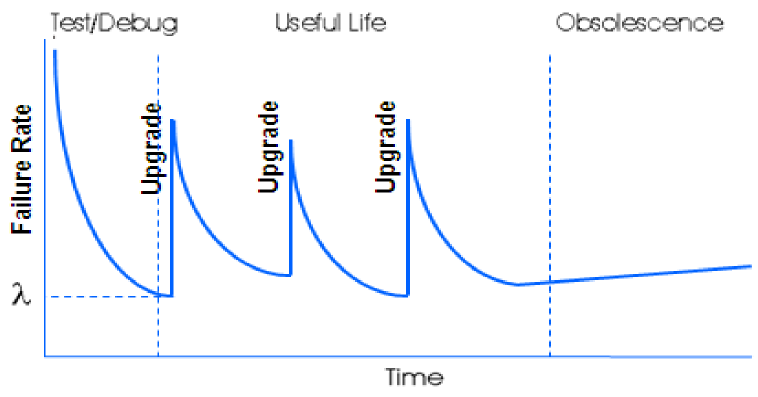

Figure 10: Failure Density Curve across Multiple Software Releases

In summary, there is a need for monitoring leading indicators of increased risk in evolving software-reliant systems as well as for investigation into potential new problem areas and hazards due to major capability and technology upgrades and revision the process, practices, methods and tools to address them proactively.

A.2.10 Limited confidence in modeling and analysis results

Model-based engineering is considered to be a key to improving system engineering and embedded software system engineering. Modeling, analysis and simulation has been practiced by engineers for a number of years. For example, computer hardware models have been created in Very High Speed Integrated Circuits Hardware Description Language (VHDL) [VHDL NG 1997] and validated through model checking. Control engineers have used modeling languages such as Simulink for years to represent the physical characteristics of the system to be controlled and the behavior of the control system. Characteristics of physical system components, such as thermal properties, fluid dynamics, and mechanics, have been modeled and simulated at various levels of fidelity.

Even for software, systems analysis models and simulation have been used to predict various operational aspects. However, aircraft industry experience has shown that independently maintained analysis models by different teams results in a “multiple truth” problem (Figure 11). Such models are created at different times during development, based on one version of an evolving design document, and are rarely kept up-to-date with the evolving system design. The inconsistency between the analysis models, with respect to the system architecture and the system implementation renders the analysis results of little value in the qualification of the system.

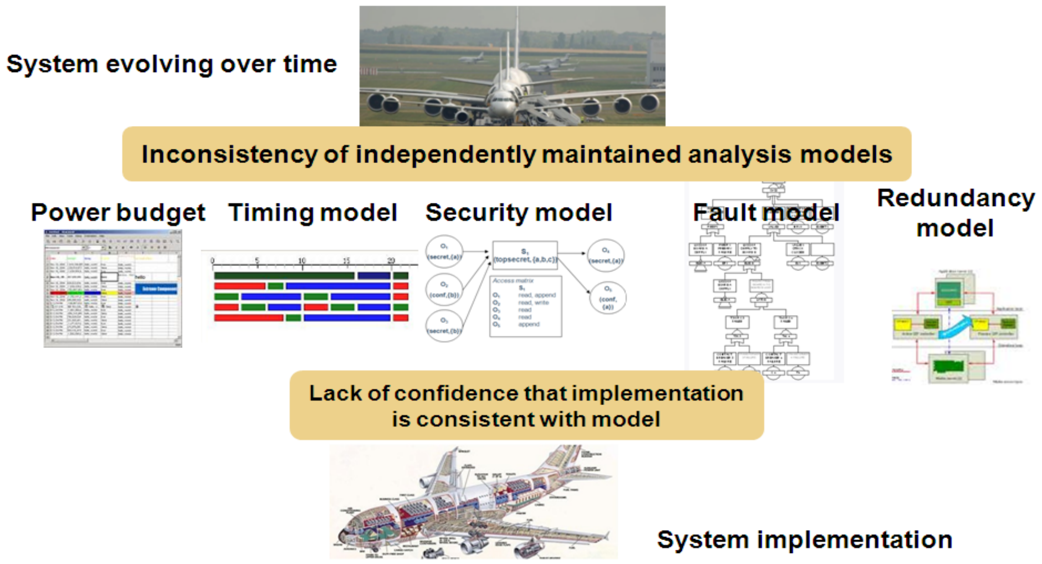
System evolving over time
Inconsistency of independently maintained analysis models
Power budget
Timing model
Security model
Fault model
Redundancy model
Lack of confidence that implementation is consistent with model
System implementation

Figure 11: Pitfalls in Modeling and Analysis of Systems

There is a need for an architecture-centric reference model approach as common source for system analysis and system generation.

A.3 Summary

Current best practice, which relies on process standards, best practices, and safety culture, is unable to cope with the exponential growth in size and interaction complexity of embedded software in today's increasingly software-reliant systems. Traditional reliability engineering has its roots in hardware, assuming slowly-evolving system designs, with reliability metrics focusing on physical wear. During system design, reliability growth is achieved by having modeling and analysis surface failure modes as they are identified. In contrast, software failure modes are primarily design errors and software is often assumed to be perfectible and show deterministic behavior. Software faults are addressed through design and code reviews as well as testing.

Moving beyond textual specification of requirements is essential to validating completeness and consistency of such specifications. Understanding the impact of architectural decisions, and identifying potentially mismatched assumptions in system interactions on non-functional mission and safety-criticality requirements early in the life cycle, require architectural models with well-defined semantics that support analysis of such system properties. Software-induced hazards must be taken into account in system safety analysis. Fault management has increasingly become responsibility of the embedded software requiring increased scrutiny in order to achieve reliability and safety goals. The complexity and the non-deterministic nature of software interaction requires formal static analysis methods to complement testing, with consistency across analysis models.

A.4 Impact of complexity on V&V

A.4.1 Negative Impacts with Rationales

Increasing complexity causes the following negative impacts:

Analysis

- Impact: Static analysis tools will be less likely to be effective and efficient.

  Rationale: Analysis tools may not scale to the level required to address the increased complexity.

Demonstration

- Impact: Demonstrations become less effective at identifying defects.

  Rationale: The demonstrated behavior is more complex and therefore more difficult to understand.

Review (including Inspections and Walk-throughs)

- Personnel Performing Testing (both testers and developers):
  - Impact: More staff are needed to perform the reviews.

    Rationale: Each review will take longer to perform.
  - Impact: Those performing testing need more expertise and experience in the application domain, the technology used, and testing.

    Rationale: Testing will become more complex and require more sophisticated methods and approaches.
- Resources (schedule and budget):
  - Impact: Requirement and architecture reviews take longer to achieve the same level of defect identification.

    Rationale: The requirements and architecture are more complex and therefore more difficult for the reviewers to understand.
  - Impact: Times allowed for reviews will need to be increased.

    Rationale: Each review will take longer to provide the same level of defect identification.
- Review Process:
  - Impact: The review process needs to be more complete and rigorous (e.g., Fagan Inspections instead of highly informal peer reviews) to achieve the same level of defect identification.

    Rationale: The increased complexity makes the work produced under review harder to understand.
- Review Inputs:
  - Impact: Requirement and architecture documents under review contain more defects.

    Rationale: The requirements and architecture are more difficult for the requirements engineers and architects to understand.
- Review Results:
  - Impact: Requirement and architecture reviews become less effective at identifying defects.

Rationale: The requirements and architecture are more difficult for the reviewers to understand.

  - Impact: Reviews produce more false-positive and false-negative results.

    Rationale: Reviewers are more likely to misunderstand the more complex requirements and architecture documents and models under review.

Testing

- Personnel Performing Testing (both testers and developers):
  - Impact: More staff are needed to perform testing.

    Rationale: There are more tests to create, execute, and report.
  - Impact: Those performing testing need more expertise and experience in the application domain, the technology used, and testing.

    Rationale: Testing will become more complex and require more sophisticated methods and approaches.
- Testing Resources (schedule and budget):
  - Impact: Testing schedules will need to be extended.

    Rationale: Testing will take longer to provide the same level of testing.
  - Impact: Testing schedules will be insufficient to achieve desired level of defect identification.

    Rationale: Other activities will take longer, leaving less time for testing.
  - Impact: Testing budgets will need to be increased.

    Rationale: Testing will take longer, will require more staffing, and more sophisticated test tools will need to be acquired.
- Test Inputs:
  - Impact: The requirements are harder for the testers to understand and analyze.

    Rationale: The requirements are more complex.
  - Impact: The requirements contain more defects.

    Rationale: The requirements are more complex.
  - Impact: The architecture is harder for the testers to understand and analyze.

    Rationale: The architecture (and architectural models) are more complex.
  - Impact: The architecture contains more defects.

    Rationale: The architecture (and architectural models) are more complex.
  - Impact: The software is harder for the testers to understand and analyze.

    Rationale: The requirements, architecture, and software are more complex.
  - Impact: The software contains more defects to uncover.

    Rationale: The requirements and architecture driving the implementation are more complex.
- Test Process:

  - Impact: The testing process (especially test design including test case selection and completion criteria) needs to be more formal and rigorous to achieve the same level of coverage.

    Rationale: The increased complexity makes informal testing less effective and uncovering defects.
  - Impact: The testing process needs to be more complete to achieve the same level of coverage.

    Rationale: The increased complexity results in more paths, states and state transitions, or input spaces.
- Test Planning (Test Strategy and Test Plan):
  - Impact: Test Strategies and Plans will need to address how testing will handle the complexity.

    Rationale: TBD.
- Test Environment/Tools: TBD.
- Test Design (Test Development and Design of Experiment):
  - Impact: The test cases including test preconditions, test scripts, test data, and test postconditions (expected behavior in terms of outputs and state) have more defects.

    Rationale: The complexity of the test source material (such as requirements and architecture) are more difficult for the testers to understand.
  - Impact: The number of tests (in terms of test suites and test cases) needs to increase to achieve the same level of defect identification.

    Rationale: The increased complexity results in more paths, states and state transitions, or input spaces.
  - Impact: Testing (in terms of test suites and test cases) is more incomplete.

    Rationale: The increased complexity results in more paths, states and state transitions, or input spaces. There is typically less time to design and develop the tests.
  - Impact: Test case selection needs to be prioritized.

    Rationale: Testing becomes less complete and limited testing resources need to be used.
  - Impact: Test design in terms of test case selection is more difficult to do.

    Rationale: The test design drivers are harder to understand.
  - Impact: Informal test design approaches (e.g., Exploratory Testing) become less effective.

    Rationale: Complexity overwhelms the tester’s ability to informally identify and construct a complete and correct set of test suites and test cases.
  - Impact: The test design approaches (including test case selection criteria) need to be less informal and more sophisticated.
  - Rationale: Informal and simple test design approaches produce incomplete test suitesand test cases that have more defects.
  - Impact: Test completion criteria will be harder to set and achieve.

    Rationale: TBD.

  - Impact: Test automation becomes more difficult to perform.

    Rationale: TBD.
- Test Execution:
  - Impact: Tests take longer to execute.

    Rationale: There should be more tests and individual tests are more complex.
- Test Reporting: TBD
- Test Outputs:
  - Impact: Testing work products (e.g., test cases, test data, test scripts, test reports) will contain more defects.

    Rationale: TBD.
  - Impact: Testing produces more false-positive and false-negative results.

    Rationale: TBD.
  - Impact: Test results are harder to understand and interpret.

    Rationale: TBD.
  - Impact: The system and software will contain more residual defects after testing.

    Rationale: Testing will be less effective and efficient.

A.4.2 Additional inputs

The following additional inputs were, in part, crowd-sourced from LinkedIn groups on testing and quality assurance. They have not yet been fully evaluated.

Small changes to complex systems can have a much larger effect within the overall system. Complexity vastly increases the challenge of keeping documentation up-to-date.

Complexity makes it difficult to assess the effects of interactions among different parts of the system due to timing, concurrency, and emergence of unforeseen behavior.

What complexity does to the V&V phases:

- Complexity can cause a loss of control over the V&V program, e.g., test coverage, test result reporting, test data. There are also more test cases, and when writing tests, general automatic tests are not sufficient; specific tests have to be written (this is time-consuming and error-prone)
- Time consuming **-** in general, V&V for complex systems consumes time and productivity – E.g., testers need far more effort to understand the code first. Also, increased system complexity requires more effort to define, create, and maintain representative test environments. As a result, deliverables promised on timescales appropriate for less complex software will be late.
- Effort: The efforts to generate V&V activities will increase: the time/effort required for generating test cases for a complex system will be very much higher than that of a simple system. Larger budgets should be allocated. Generally more test cases are needed to achieve adequate coverage.
- Flawed design and implementation. The code being tested may not be well-architected and designed. During development and test, complexity causes incomplete or siloed domain

knowledge, leading to reliance on assumptions and to flaws in requirements and design Complex code is also more likely to have errors: more bugs are introduced and they are less likely to be found.

- Usability testing becomes both more important and more difficult.
- Because the subject being reviewed is complex, reviews raise too few errors per unit time and too many errors slip through.
- Lack of tools: there may not be system/data models, utilities, etc., that are capable of handling a system with unprecedented complexity.
- Skill Set: The skill set of the testers (i.e. knowledge of architecture and design of the system under test) is important. If the testers lack understanding of the system, then they may not write effective and optimized tests. They need modelling, quantitative analysis, statistics knowledge. Also, staff can become “Head Monopoles” and when they are on holiday and it is difficult for another colleague to take over. The more complex the software is, the harder it for a "newbie".
- Inefficient measures. Measures are complex, leading to too many meetings for clarification surrounding complex subject matters.
- Inability to assess “well enough tested?” - What if 95% of the logic is in 5% of the code? We don’t know that the complex part is well enough tested.
- “Sunny day” V&V, showing functionality under normal operational conditions, becomes more difficult. Additionally, “rainy day” V&V, showing that unexpected conditions are dealt with appropriately, is much more important and also more difficult.

What we must do because the software being tested is complex:

- We need a better approach during development to reduce the complexity of the software in the first place, with proper quality gates, and a good unit/integration coverage during build.
- We need behavior-driven story creation, and the customer behavior / usage should be prioritized.
- We need a smart test strategy. For example, we need smart testers who test the system with its complexities, instead of adding extraneous verification and validation points
- We need specific test plans targeting different areas of releases, considering the impact and risk involved.
- We need more testing. We need to invest extra resources in both V&V documentation and the V&V process.
- We need to move testing to the left: incorporate more testing at the time of development, i.e., we need test-driven development (TTD) We should test applications when they are young, before they become complex. Even complex applications are initially simple and easy, one class, one method, etc.
- We need to improve our testing speed with appropriate test automation tools, and we need to start scripting early, before the code is in for testing
- We need a constructive approach to automation, to verify application flows with minimal intervention and good confidence.

- We need to obtain documentation covering every aspect of the complex system. Complexity can come from many sources. Those sources must have clarity so as to avoid ambiguity, assumptions along with any other decision point relative to understanding the complexity and delivering against it with firm measures.

References

[Blanchette 2009]
Blanchette, S. "Assurance Cases for Design Analysis of Complex System of Systems Software." *American Institute for Aeronautics and Astronautics (AIAA) Infotech@Aerospace Conference*. Seattle, Washington, U.S.A., April 2009. http://www.sei.cmu.edu/library/abstracts/whitepapers/ Assurance-Cases-for-Design-Analysis-of-Complex-System-of-Systems-Software.cfm

[Boehm 1981]
Boehm, B.W. *Software Engineering Economics*. Prentice Hall, 1981.

[Boydston 2009]
Boydston, A. & Lewis, W. "Qualification and Reliability of Complex Electronic Rotorcraft Systems," Presented at the *American Helicopter Society (AHS) Symposium*. Quebec, Canada, October 2009.

[Dabney 2003]
Dabney, J. B. Return on Investment of Independent Verification and Validation Study Preliminary *Phase 2B Report*. NASA, 2003.

[Ellison 2008]
Ellison, R., Goodenough, J., Weinstock, C., & Woody, C. *Survivability Assurance for System of Systems* (CMU/SEI-2008-TR-008). Software Engineering Institute, Carnegie Mellon University, May 2008. http://www.sei.cmu.edu/reports/08tr008.pdf

[FAA 2010]
FAA Certification Authorities Software Team (CAST) Position Paper CAST-10, What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage *(DC)?* June 2002. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/ cast_papers/media/cast-10.pdf

[FAA 2000]
Federal Aviation Administration. System Safety Handbook, 2000. http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/

[FDA 2010]
U.S. Food and Drug Administration. Guidance for Industry and FDA Staff – Total Life Cycle: *Infusion Pump – Premarket Notification [510(k)] Submissions*. 2010. http://www.fda.gov/ MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm206153.htm

[Feiler 2008]
Feiler, Peter. “Efficient Embedded Runtime Systems through Port Communication Optimization,” 294-300. *13th IEEE International Conference on Engineering of Complex Computer Systems*. Belfast, Northern Ireland, March 2008. IEEE Computer Society, 2008.

[Feiler 2010]
Feiler, P., Wrage, L., & Hansson, J. “System Architecture Virtual Integration: A Case Study.” *Embedded Real-time Software and Systems Conference (ERTS 2010).* May 2010. http://www.erts2010.moonaweb.com/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%20 2/ERTS2010_0105_final.pdf

[Galin 2004]
Galin, D. *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison-Wesley, 2004.

[Jackson 2007]
Jackson, Daniel, Thomas, Martyn, & Millett, Lynette I., eds. *Software for Dependable Systems: Sufficient Evidence?* National Research Council of the National Sciences, 2007.

[Jones 2010]
Jones, C. “Software Quality and Software Economics.” *Software Tech News 13*, 1, April 2010. https://softwaretechnews.thedacs.com/stn_view.php?stn_id=53&article_id=154.

[Kelly 1998]
Kelly, T. “Arguing Safety—A Systematic Approach to Safety Case Management.” PhD diss., University of York, Department of Computer Science, 1998.

[Kelly 2004]
Kelly, T. & Weaver, R. “The Goal Structuring Notation: A Safety Argument Notation.” *Proceedings of International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC 2004).* Edinburgh, Scotland, May 2004. IEEE Computer Society, 2004.

[Konrad 2015]
Konrad, M., and S. Sheard. 2015 “FAA Research Project: System Complexity Effects on Aircraft Safety: Literature Review Task 3.2: Literature Search to Define Complexity for Avionics Systems.” Special Report CMU/SEI-2015-SR-006. Pittsburgh: Carnegie Mellon Software Engineering Institute.

[Koopman 1999]
Koopman, P. & DeVale, J. “Comparing the Robustness of POSIX Operating Systems,” 30-37. *Digest of Papers: 29th Fault Tolerant Computing Symposium*. Madison, Wisconsin, June 1999. IEEE, 1999.
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=781027

[Langenbrunner 2010]
Langenbrunner, A. J. & Trautwein, M. R. “Extending the Strategy-Based Risk Model: Application to the Validation Process for R&D Satellites.” *2010 IEEE Aerospace Conference Proceedings* (CD-ROM). Big Sky, MN, Mar. 6–13, 2010. IEEE Computer Society Press, 2010.

[Leveson 2000]
Leveson, Nancy G. “Intent Specifications: An Approach to Building Human-Centered Specifications.” *IEEE Transactions on Software Engineering 26*, 1 (January 2000): 15–35.

[Leveson 2004]
Leveson, Nancy. “A New Accident Model for Engineering Safer Systems.” *Safety Science 42*, 4 (April 2004): 237–270.

[Leveson 2005]
Leveson, Nancy G. “A Systems-Theoretic Approach to Safety in Software-Intensive Systems.” *IEEE Transactions on Dependable and Secure Computing 1*, 1 (January 2005): 66–86.

[Leveson 2009]
Leveson, Nancy G. *Engineering a Safer World: System Thinking Applied to Safety*. MIT Press, 2011. http://sunnyday.mit.edu/safer-world/safer-world.pdf

[Madachy 2010]
Madachy, Raymond, Boehm, Barry & Houston, Dan. “Modeling Software Defect Dynamics,” *DACS SoftwareTech*, March 2010.
https://softwaretechnews.thedacs.com/stn_view.php?stn_id=53&article_id=157

[Miller 2005a]
Miller, S. P., Anderson, E. A., Wagner, L. G., Whalen, M. W., & HeimDahl, M. “Formal Verification of Flight Critical Software.” Presented at the *AIAA Guidance, Navigation and Control Conference*. San Francisco, CA, August, 2005.
http://shemesh.larc.nasa.gov/fm/papers/FormalVerificationFlightCriticalSoftware.pdf

[Miller 2005b]
Miller, S., Whalen, M., O’Brien, D., Heimdahl, M. P., & Joshi, A. *A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures* (NASA/CR-2005-213912). NASA, 2005.

[Nelson 2008]
Nelson, P. S. “A STAMP Analysis of the LEX ComAir 5191 Accident.” Master's thesis, Lund University, Sweden, 2008.

[Nichols 2015]
Konrad, M., and S. Sheard. 2015 “FAA Research Project: System Complexity Effects on Aircraft Safety: Task 3.3: Candidate Complexity Metrics.” Special Report CMU/SEI-2015-SR-012. Pittsburgh: Carnegie Mellon Software Engineering Institute.

[NIST 2002]
National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). NIST, 2002.

[Nuseibeh 1997]
Nuseibeh, Bashar. “Ariane 5: Who Dunnit?” *IEEE Software 14*, 3: 15–16, 1997. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=589224

[Parnas 1991]
Parnas, D. L. & Madey, J. *Functional Documentation for Computer Systems Engineering, Version 2* (Technical Report CRL 237). McMaster University, Ontario, 1991.

[Rushby 2009]
Rushby, John. “Software Verification and System Assurance,” 3–10. *Proceedings of the Seventh IEEE International Conference on Software Engineering and Formal Methods*. Hanoi, Vietnam, Nov. 2009. IEEE Computer Society Press, 2009.

[SAE 1996]
Society of Automotive Engineers International. *Recommended Practice: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (ARP4761). 1996. http://standards.sae.org/arp4761/

[Spiegel 2010]
Spiegel Online. “The Last Four Minutes of Air France Flight 447,” 2010. http://www.spiegel.de/international/world/0,1518,679980,00.html

[Tribble 2002]
Tribble, A. C., Lempia, D. L., & Miller, S. P. *Software Safety Analysis of a Flight Guidance System*. http://shemesh.larc.nasa.gov/fm/papers/Tribble-SW-Safety-FGS-DASC.pdf (2002).

[VHDL NG 1997]
VHSIC Hardware Description Language Newsgroup. *Frequently Asked Questions and Answers*. http://www.vhdl.org/comp.lang.vhdl/ (1997).

[Weinstock 2009]
Weinstock, Charles B. and John B. Goodenough. 2009. *Towards an Assurance Case Practice for Medical Devices.* CMU/SEI-2009-TN-018. Pittsburgh: Carnegie Mellon Software Engineering Institute.

Acknowledgments

The authors would like to acknowledge the excellent and timely technical editing performed by Erin Harper and continued support from other team members, Bill Nichols and Greg Such, and SEI technical experts Peter Feiler and John Goodenough.

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu | www.cert.org
Email: info@sei.cmu.edu

This material is based upon work funded and supported by Federal Aviation Administration under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM-0004271