Software Engineering Institute

Assuring Software Reliability

Robert J. Ellison

August 2014

SPECIAL REPORT
CMU/SEI-2014-SR-008

Program Name
CERT Division and Software Solutions Division

http://www.sei.cmu.edu

Carnegie Mellon University

Copyright 2014 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense 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.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution except as restricted below.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

DM-0001479

Table of Contents

Acknowledgments **vii**

Executive Summary **ix**

Abstract **xi**

1 Introduction **1**
- 1.1 Software and Hardware Differences 2
- 1.2 Software Reliability and Software Assurance 3
- 1.3 Security Example 4

2 Documenting Engineering Decisions **6**
- 2.1 Medical Infusion Pumps 6
- 2.2 Infusion Pump Assurance Case 8
- 2.3 Fault Reduction Example 9
- 2.4 Replace Fault Tolerance Report with an Assurance Case Justification 10

3 Causal Analysis of Software Failures **13**
- 3.1 2003 Power Grid Blackout 13
- 3.2 The Power Grid Failure 14
  - 3.2.1 Software Assurance Analysis 15
  - 3.2.2 Other Observations 16
- 3.3 Sustainment 17

4 Analyzing the Confidence of an Assurance Case **18**
- 4.1 Eliminative Induction 18
- 4.2 Reliability Validation and Improvement Framework 21
- 4.3 Incorporating into Design Reviews 22

5 Assuring Software Reliability **23**
- 5.1 Requirement Errors 23
- 5.2 Available Expertise 25
  - 5.2.1 Fault Tolerance Example 25
  - 5.2.2 Improving Availability of Security Expertise 26

6 Conclusion **28**

References/Bibliography **31**

List of Figures

Figure 1: Failure Distribution Curves 2

Figure 2: Security Controls 4

Figure 3: Vulnerability Management 5

Figure 4: Goal Structured Notation 8

Figure 5: Using GSN for Infusion Pump Assurance Case 8

Figure 6: Assurance Case for Fault Reduction Report 11

Figure 7: Alternate Ways to Provide Situational Awareness 15

Figure 8: Confidence Map 19

Figure 9: Light Bulb Example 20

Figure 10: Light Bulb Rebutters 20

Figure 11: Expanded Confidence Map 21

Figure 12: Error Leakage Rates Across Development Phases 23

Figure 13: Scoring Legend 29

Figure 14: KPP Scored Diagram 30

List of Tables

Table 1: Software Reliability Challenges 3
Table 2: Infusion Pump Hazards and Health Risks 7
Table 3: Evaluating Pump Hazard Recognition 9
Table 4: Fault Management Report 10
Table 5: Reliability Framework Technologies 21
Table 6: BSIMM Examples 27

Acknowledgments

This report draws extensively on the work done at the Software Engineering Institute on assurance cases and software reliability for safety-critical systems. Charles Weinstock provided material used in several of the examples and for the section that introduces confidence maps.

Executive Summary

The 2005 *Department of Defense Guide for Achieving Reliability, Availability, and Maintainability* (RAM) emphasized the importance of systems engineering design analysis over predicting software reliability based an analysis of faults found during integration. Requirements and design faults have accounted for 70 percent of the errors in embedded safety-critical software. The rework effort to correct such errors found during testing and system integration can be 300 to 1,000 times the cost of in-phase correction. The existing build and test paradigm for developing reliable systems is not feasible with the increasing complexity of software-intensive systems.

The different characteristics of hardware and software failures require analysis techniques distinct from those used for hardware reliability. For example, the cause of a hardware device failure can often be traced to a single event, e.g., the failure of a specific component. But a failure for a complex system is likely the result of a combination of events. None of those events can individually cause a failure, but the concurrence of all of them leads to a failure.

Improving software reliability can require significant changes in development and acquisition practices and will involve a learning curve for both acquirers and suppliers. Realistically, the changes have to be incremental. This document describes ways that the analysis of the impact of potential software failures (regardless of cause) can be incorporated into acquisition practices.

Software reliability is a statistical measure: the *probability* that a system or component performs its required functions under stated conditions for a specified period of time, i.e., no failures occur over that time period. It is not a measure of risk for a specific failure. For example, a highly reliable system is neither necessarily safe nor secure. Safety and security depend on mitigating specific kinds of faults.

Mitigating specific faults is more the province of system and software assurance. System assurance is defined as the confidence that a system behaves as expected. The term *assurance* is often associated with safety, but is increasingly applied to other attributes such as security and reliability. For security, the expected behavior is the desired system response to conditions created by an attacker.

A formal review of a design as recommended by the DoD RAM Guide requires that we can analyze how the engineering decisions support the reliability requirements. That analysis has to be described in a concise and understandable way. Software assurance provides mechanisms to do that. For example, an early design may be incomplete, may have overlooked some hazards or may have made invalid or inconsistent development or operating assumptions. An objective of an early design review should be to identify such concerns when they can be more easily fixed. An assurance case provides a systematic way for doing such a review.

The Software Engineering Institute has applied software assurance techniques in the early phases of the system development lifecycle for a large DoD system of systems. In addition to supporting technical analysis, the results of the assurance analysis can be displayed in a way that gives managers answers about the design progress that are demonstrably rooted in facts and data instead of opinions based on hope and best intensions. In addition, the analysis provides a way to show the effects of a specific development shortfall.

Abstract

The 2005 *Department of Defense Guide for Achieving Reliability, Availability, and Maintainability* (RAM) recommended an emphasis on engineering analysis with formal design reviews with less reliance on RAM predictions. A number of studies have shown the limitations of current system development practices for meeting these recommendations. This document describes ways that the analysis of the potential impact of software failures (regardless of cause) can be incorporated into development and acquisition practices through the use of software assurance.

1 Introduction

The limitations of current system development practices for meeting reliability requirements are evident in safety-critical systems where system-level faults due to software have increasingly dominated the rework effort. Several studies of safety-critical systems show that while 70 percent of errors in embedded safety-critical software are introduced in the requirements and architecture design phases [Feiler 2012], 80 percent of all errors are only found at system integration or later. In particular, these errors were not found in unit testing. The rework effort to correct requirement and design problems in later phases can be as high as 300 to 1,000 times the cost of in-phase correction, and undiscovered errors likely remain after that rework.

The 2005 *Department of Defense (DoD) Guide for Achieving Reliability, Availability, and Maintainability* (RAM) identified four steps required for RAM improvement.

- Step 1: Understand and document user needs and constraints
- Step 2: Design and redesign for RAM
- Step 3: Monitor field performance
- Step 4: Produce reliable and maintainable systems

The *Guide* noted that one of reasons for reliability failures in DoD systems was too great of reliance on predictions. For example, the recommendations listed in Step 2 of the *Guide* include

- Emphasize systems engineering design analysis and rely less on RAM predictions.

  Improving systems engineering designs for software reliability starts with an understanding for how the characteristics of software failures require analysis techniques distinct from those used for hardware reliability.

  An emphasis on systems engineering design analysis and less reliance on RAM predictions provides a way to reduce specific system risks. Software reliability is a statistical measure: the probability that a system or component performs its required functions under stated conditions for a specified period of time, i.e., no failures occur over that time period. It is not a measure of risk for a specific failure. The requirements for military systems typically identify specific faults that must be mitigated.

  System assurance provides the techniques for mitigating the risk of specific system failures. Its importance is recognized by its appearance in many of the recommendations that appear in The National Research Council report, *Critical Code: Software Producibility for Defense.*[1]

  The above items are introduced later in this section.

- Conduct formal design reviews for reliability and maintainability and in particular use an impartial, competent peer to perform the review.

---

[1] http://www.nitrd.gov/nitrdgroups/images/6/64/CritCodev27assurHCSS.pdf

An objective of a review is to confirm that an implementation based on a design is likely to meet requirements. For example, a review of a disk drive design would check how the techniques used manage read errors or surface defects. A review of a software design might need to verify if the applied software engineering choices sufficiently mitigate a fault. Hardware reliability such as for a disk drive can draw on documented design rules based on actual usage. Software reliability has not matured to the same state. A description of the specific engineering decisions and the justification for those choices has to be provided for the review. An assurance technique called an assurance case provides a way to document the reasoning and evidence that support the engineering choices.

Assurance cases are explained in Section 2 by the use of examples.

1.1 Software and Hardware Differences

The differences between software and hardware reliability are reflected in the associated failure distribution curves. A *bathtub curve* shown in Figure 1 describes the failure distribution for hardware failures. The bathtub curve consists of three parts: a decreasing failure rate (of early failures), a constant failure rate (of random failures), and an increasing failure rate (of wear-out failures) over time. Software defects exist when a system is deployed. Software's failure distribution curve, also shown in Figure 1, reflects changes in operational conditions that exercise those defects as well as new faults introduced by upgrades. The reduction of errors between updates can lead system engineers to make reliability predictions for a system based on a false assumption that software over time is perfectible. Complex software systems are never error-free.

Table 1 lists some of the challenges for implementing the DoD Guide recommendations for software reliability. These items are analyzed in the Software Engineering Institute (SEI) white paper *Evaluating Software's Impact on System and System of Systems Reliability* [Goodenough 2010]. This report suggests some techniques that address those challenges.

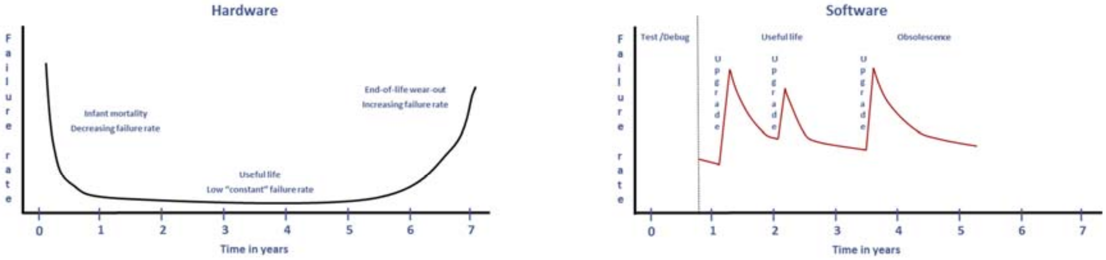


Figure 1: Failure Distribution Curves

Hardware failures increase over time because of wear. The risk of failures for a complex system can also increase over time but for different reasons. Leveson makes such a claim for the risk of safety failures [Leveson 2011]. She identified three reasons for this trend: (1) the impact of the operational environment, (2) unintended effects of design changes, and (3) changes in software development processes, practices, methods, and tools.

Table 1: Software Reliability Challenges

Software impact on systems	The recommended development activities depend on understanding the impact that software can have on platform reliability, availability, and maintainability. Improving development activities devoted to analyzing the potential impact of software failures (regardless of cause) is needed to minimize software's impact on system aborts (SA) and essential function failures (EFF) in complex stand-alone systems as well as in systems of systems.
Possible failure modes	Implement a software reliability improvement process similar to what is done for hardware. Software failure modes may be the loss or reduction in a capability. For each failure mode, additional analysis is needed to show what the recovery method will be, e.g., after a software-caused failure • Is a system reboot necessary? • Can the operator fall back to a previously saved "good" state and try again? • Is there an alternate method that might avoid the subsystem that isn't working?
Redesign analysis based on failure impact	The safety-critical, space-borne, and avionics systems where software dependence is understood give more attention to failure impact and redesign analysis activities than is commonly observed. But as noted by the DoD RAM *Guide*, improving the reliability of software intensive systems requires determining if the engineering decisions have sufficiently mitigated the effects of a set of hazards. As with the safety-critical systems such analysis requires redesign activities based on a concise and understandable description of the decisions made.
Integration of hardware and software failure analysis	Too often software and hardware techniques, as traditionally considered, are done independently of each other; each assumes the other is 100 percent reliable, and the analysis does not consider interactions between software and non-computer hardware. There may be an implicit assumption that software quality is always improving as once a problem is found, it can be removed and will never occur again. But such an assumption is a reliability risk. Software is never perfect so a system needs to be designed to recover from (currently unknown) faults whose effects are encountered only rarely.

1.2 Software Reliability and Software Assurance

Software reliability is a statistical measure: the *probability* that a system or component performs its required functions under stated conditions for a specified period of time, i.e., no failures occur over that time period. It is not a measure of risk for a specific failure. For example, a highly reliable system is neither necessarily safe nor secure. Safety and security depend on mitigating specific kinds of faults.

System and software assurance was identified as one of four key technologies required for addressing the challenges of qualifying increasingly software-reliant, safety-critical systems by software studies done by the National Research Council [Jackson 2007], NASA [Dvorak 2009], the European Space Agency (ESA) [Conquet 2008], the Aerospace Vehicle Systems Institute (AVSI) [Feiler 2009a], and AED [Boydston 2009].

The Open Group Real Time and Embedded Systems Forum reached a similar conclusion for dependability (includes reliability, availability, performance, security, integrity, and safety) [Open Group 2013]. Conventional technologies, such as software processes and/or formal methods, are not sufficient to meet dependability requirements for computing systems that are used for long period of times and are continually upgraded to reflect evolving technologies and changing regulations and standards.

The difficulties in meeting requirements for the attributes that The Open Group includes under dependability arise when a system encounters adverse rather than normal conditions. Security is an example. Valid input data does not compromise a system. Security attacks frequently succeed because of unexpected system behavior when it processes invalid input.

A reference to the application of software assurance to a specific system attribute often includes the attribute name. Software assurance applied to safety is referred to as *safety assurance*. The definition of software assurance in this document does not reference a specific system property.

> **Software Assurance:** The application of technologies and processes to achieve a required level of confidence that software systems and services function in the intended manner.

1.3 Security Example

Security is a good example for the importance of considering software failures during design. Security is a system property that has often been implemented by adding preventative security controls such as user authentication and authorization, data encryption, and network data flow control mechanisms. But attack tactics change, and systems now are frequently compromised exploiting a mistake made during the development of application software such as not validating input values. Such vulnerabilities enable attackers to go around the strongest security controls. The more than 900 known software vulnerabilities exceed the protective capabilities of security controls.

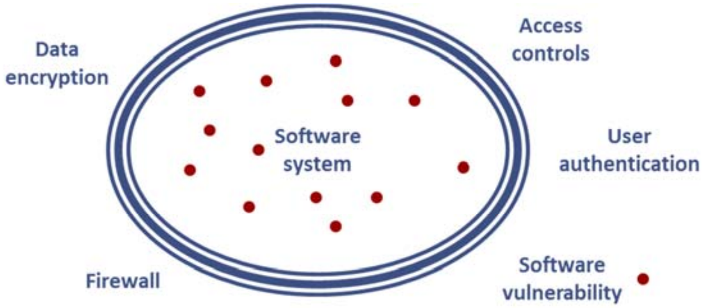


Figure 2: Security Controls

We needed to reduce the number of inadvertent vulnerabilities created during development. Vulnerability elimination based only testing as shown in Figure 3 was not satisfactory. Attackers were exploiting security risks created by the functional and operational requirements. Such risks had to be considered during the design. How could externally supplied data adversely affect a design of a function? For example, a detailed design could propose using a relational database query language to retrieve data required to meet a functional requirement. But the design also has to consider the security risks associated with that implementation. An attack tactic called a SQL-Injection has repeatedly exploited the use of a database query language to access supposedly inaccessible information or to make unauthorized changes to the data store. The detailed design has to incorporate the known mitigations for this risk, i.e., security has to be built into the software rather than being added later. This change in perspective is also reflected in version 4 of NIST 800-53, *Recommended Security Controls for Federal Information Systems and Organizations*, where “building it right” is an essential component for developing a secure system.

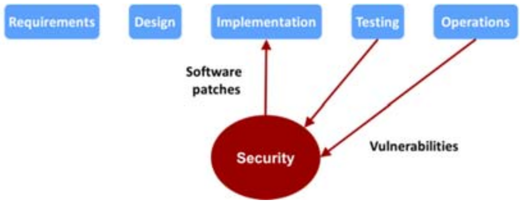

Requirements
Design
Implementation
Testing
Operations
Software patches
Vulnerabilities
Security

Figure 3: Vulnerability Management

Summary

Improving system assurance is the objective for many of the recommendations that appear in The National Research Council report, *Critical Code: Software Producibility for Defense*. The feasibility of achieving high assurance for a particular system is strongly influenced by early engineering choices. In particular, assessing assurance as a system is being developed had a high potential for improving the overall assurance of systems. Such an approach can require significant changes in development and acquisition practices. But realistically, the changes have to be incremental. This document describes ways that the analysis of the potential impact of software failures (regardless of cause) can be incorporated into acquisition practices.

2 Documenting Engineering Decisions

A formal engineering review requires more than a description of a design. The objective is to identify design errors during the design and at the latest in the design review. Reliability depends on identifying and mitigating potential faults. A design review should verify that faults associated with important business risks have been identified and mitigated by specific design features.

In this section we introduce the use of assurance cases to document engineering decisions. We begin by illustrating how the U.S. Food and Drug Administration (FDA) used an assurance case to improve the safety and reliability of a medical infusion pump.

2.1 Medical Infusion Pumps

A patient-controlled analgesia infusion pump is used to infuse a pain killer at a prescribed basal flow rate which may be augmented by the patient or clinician in response to patient need within safe limits. Infusion pumps in general have reduced medication errors and improved patient care by allowing for a greater level of control, accuracy, and precision in drug delivery than was obtainable using previous, more labor intensive techniques.

The FDA uses a premarket assessment to certify the safety and reliability of medical infusion pumps before they are sold to the public. In spite of the FDA's assessment, too many approved pumps exhibited hardware and software defects in the field, leading to death or injury of patients [FDA 2010].

The FDA requires that class III medical devices, which include infusion pumps, undergo a scientific and regulatory review to evaluate their safety and effectiveness. But infusion pumps that passed such a review had been associated with persistent safety problems. From 2005 through 2009, 87 infusion pump recalls were conducted by firms to address identified safety problems. The problems have been observed across multiple manufacturers and pump types. The FDA became aware of many of the problems only after they occurred in actual use. These defects had not been found during development by testing and other methods.

After an analysis of pump recalls and adverse events the FDA concluded that many of the problems appeared to be related to deficiencies in device design and engineering. To address this problem, the FDA has proposed to change the premarketing scientific review to identify such problems before a pump is marketed. The changes meet the DoD RAM *Guide* recommendations, i.e. revise the premarket assessment to evaluate how the engineering done by a manufacturer reduced the health risks in using such a pump.

The health risks associated with infusion pumps are shown in Table 2. That table also includes the hazards that could affect those health risks that were identified during the analysis of pump failures. A new FDA requirement for the premarket scientific review is for a pump manufacturer to make a convincing argument for why the engineering approach used sufficiently reduces the health risks.

With the large number engineering defects that had been identified, the FDA expected that most new infusion pumps would have new implementations of software or include changes in materials, design, performance, or other features. The manufacturers had to demonstrate that the new pumps were as safe as the ones they replaced and that changes made in the pumps and in the engineering methods did not raise different questions of safety and effectiveness compared to existing devices.

Table 2: Infusion Pump Hazards and Health Risks

Hazards		Health Risks	
Software	Environmental	Overdose	Trauma
Operational	Mechanical	Air embolism	Exsanguination
Electrical	Hardware	Infection	Electric shock
Biological and Chemical	Use	Allergic response	Underdose
		Delay of therapy	

Specific hazards include air in drug delivery line, tampering (for example, by a patient during home use to adjust drug delivery), network error, false alarm or lack of an alarm caused by a sensor that is out of calibration, alarm priorities improperly set, incorrect settings of alarm thresholds, and software runtime error. For serious hazards, an alarm should sound.

The mitigation of the spectrum of hazards identified by the FDA requires an integrated hardware and software solution. For example, a manufacturer has to demonstrate that a pump's software has been designed to analyze the data from multiple sensors to correctly identify and respond to adverse conditions.

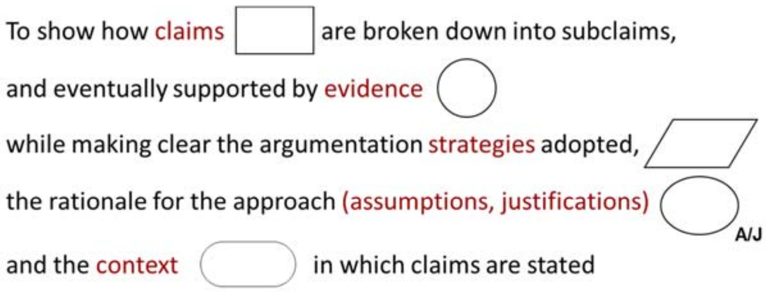
A manufacturer, on the submission of a pump, is to provide evidence and an argument that supports the claim that the pump has been engineered to safely manage the identified hazards. The FDA's task is to determine if the supplied materials are convincing so that the pump can be used by the public. To aid its analysis, the FDA suggested that the manufacturers use an assurance case as a way to structure the information in a concise and understandable manner.

> **Assurance case**: a documented body of *evidence* that provides a convincing and valid *argument* that a specified set of critical *claims* about a system's properties are adequately justified for a given application in a given environment.

An assurance case does not imply any kind of guarantee or certification. It is simply a way to document the rationale behind system design decisions. The FDA explicitly stated that pumps could not be designed to be error free.

Using only text to document and justify the engineering decisions for an infusion pump would involve 72 combinations of hazards and health risks. The aspects of the justification of a low risk of an overdose associated with environmental hazards may also support a claim for a low risk of an overdose for operational hazards. Showing such the relationships among the justifications is easier to do with a graphical notation than with text.

The Goal Structured Notation (GSN) shown in Figure 4 is widely used for assurance cases [Kelly 2004].



To show how claims are broken down into subclaims,

and eventually supported by evidence

while making clear the argumentation strategies adopted,

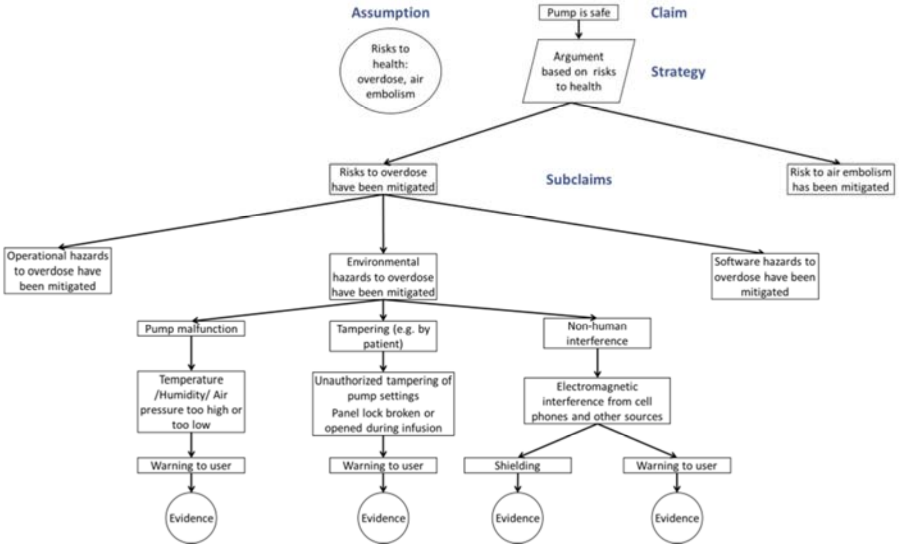
the rationale for the approach (assumptions, justifications)

and the context in which claims are stated

Figure 4: Goal Structured Notation

2.2 Infusion Pump Assurance Case

A sketch of an infusion pump assurance case is shown in Figure 5. The strategy box says that safety of the pump will be shown by demonstrating that each of the nine health risks has been mitigated, i.e., divide the initial claim into—it is hoped—simpler subclaims. The overdose subclaim is then divided into subclaims based on the hazards that could result in an overdose. For example, an overdose could be caused by environmental hazards such as a high temperature which led to a pump malfunction. It would also result from a patient tampering with unit or from cell phone interference.



Assumption
Risks to health: overdose, air embolism
Pump is safe
Claim
Argument based on risks to health
Strategy
Risks to overdose have been mitigated
Subclaims
Risk to air embolism has been mitigated
Operational hazards to overdose have been mitigated
Environmental hazards to overdose have been mitigated
Software hazards to overdose have been mitigated
Pump malfunction
Tampering (e.g. by patient)
Non-human interference
Temperature /Humidity/ Air pressure too high or too low
Unauthorized tampering of pump settings Panel lock broken or opened during infusion
Electromagnetic interference from cell phones and other sources
Warning to user
Warning to user
Shielding
Warning to user
Evidence
Evidence
Evidence
Evidence

Figure 5: Using GSN for Infusion Pump Assurance Case

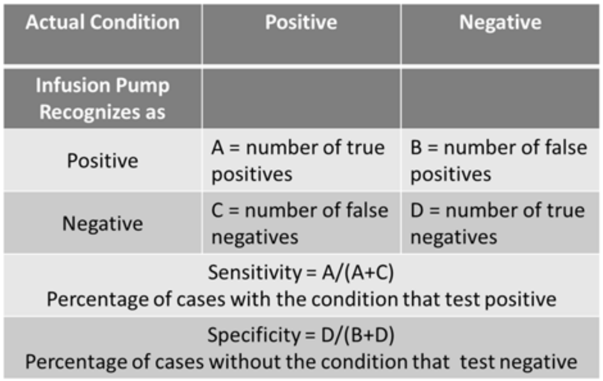
A manufacturer needs to show that the sensors have been engineered to monitor the state of physical configuration and the flow of the medicine and that the software has been designed to recognize hazards based on the aggregate of sensor values and take the action as defined by the safety requirements.

For example, if an overdose has a high health risk, then an alarm should sound. An overdose can be the result of a free flow of the medication because the valves in the delivery paths are broken or the delivery path is damaged, creating a vent in a line. The high health risk requires that an infusion pump be designed to be very sensitive to hazards that could cause an overdose, i.e., the percentage of such hazardous condition that are not recognized has to be low (see Table 3).

There are other cases where the recognition has to be what is called specific, i.e., there is a high probability that the condition recognized by the pump actually exists. A warning light that comes on too often when there are no problems will soon be ignored.

It should not be surprising that testing alone had not provided sufficient evidence to demonstrate acceptably safe infusion pump behavior. The justification had to consider any safety issues raised by how the infusion pump software responded to the interactions of the data from multiple sensors. The sensitivity and specificity of a pump's hazard tests had to be consistent with the associated health risks, but such analysis requires the data shown in Table 3 for each hazard considered. That analysis depends on applying techniques such as simulations, static analysis, and state machine modeling. The lack of that kind of engineering analysis during a design is consistent with the number of engineering and manufacturing defects identified by the FDA.

Table 3: Evaluating Pump Hazard Recognition



Actual Condition	Positive	Negative
Infusion Pump Recognizes as		
Positive	A = number of true positives	B = number of false positives
Negative	C = number of false negatives	D = number of true negatives
Sensitivity = A/(A+C) Percentage of cases with the condition that test positive		
Specificity = D/(B+D) Percentage of cases without the condition that test negative		

2.3 Fault Reduction Example

This example shows the benefits of using an assurance case to demonstrate a system met a fault reduction requirement. The system uses primary and secondary processors for some number of essential services. The primary and secondary processors must periodically exchange information to maintain consistency, and there is a risk that an exchange of faulty data could lead to the loss of both the primary and secondary servers.

The developer was asked to write a report that showed how the design had sufficiently reduced the risk of a failure caused by an exchange of faulty information. But a report such as the one in shown in Table 4 is unlikely to convince an expert reviewer of the sufficiency of the design. Many of the statements are ambiguous. The software safeguards listed in the third column are not specified. A reviewer would expect to see the subsystem design as a mitigation but would immediately question that functional requirements reduced faults. The latter are typically a source of faults. The imprecision continues in the verification column. No one is going to state a validation is not extensive or that reviews are superficial. Tests results are noted. But how do the results support the fault reduction claim?

Table 4: Fault Management Report

Cause or Fault	Effect/Severity/ Likelihood	Mitigation	Verification
Faulty data exchanged among redundant computers causes all computers to fail. This could occur because of Improper requirements, incorrect coding of logic, incorrect data definitions (e.g., initialized data), and/or inability to test all possible modes in the SW.	Effect: Loss of operation of system during critical phase, leading to loss of life. Severity: Catastrophic Likelihood: Improbable Class: Controlled	Software safeguards reduce, to the maximum extent feasible, the possibility that faulty data sent among redundant computers causes them to fail. Program Development Specifications and Functional SW Requirements Subsystem design and functional interface requirements are used in the design and development of the relevant SW.	Extensive validation and testing are in place to minimize generic SW problems. The contractors must perform rigorous reviews throughout the SW definition, implementation, and verification cycles. These review processes cover requirements, design, code, test procedures and results, and are designed to eliminate errors early in the SW life cycle. After completing system level verification, critical SW undergoes extensive integrated HW/SW verification at facility XXX. Extensive verification is independently performed at facility XXX, using hardware and software maintained to duplicate the configuration of the fielded system.

2.4 Replace Fault Tolerance Report with an Assurance Case Justification

An assurance case provides a more effective way to convey the engineering information in this report to both experts and non-experts. In this example, the primary claim is

> **Claim**: The likelihood of complete failure of primary and backup computers during a critical mission phase has been reduced as low as reasonably practicable (ALARP).

The assurance case shown in Figure 6 documents those engineering decisions and the role that the verification items, the evidence, have in justifying a claim.

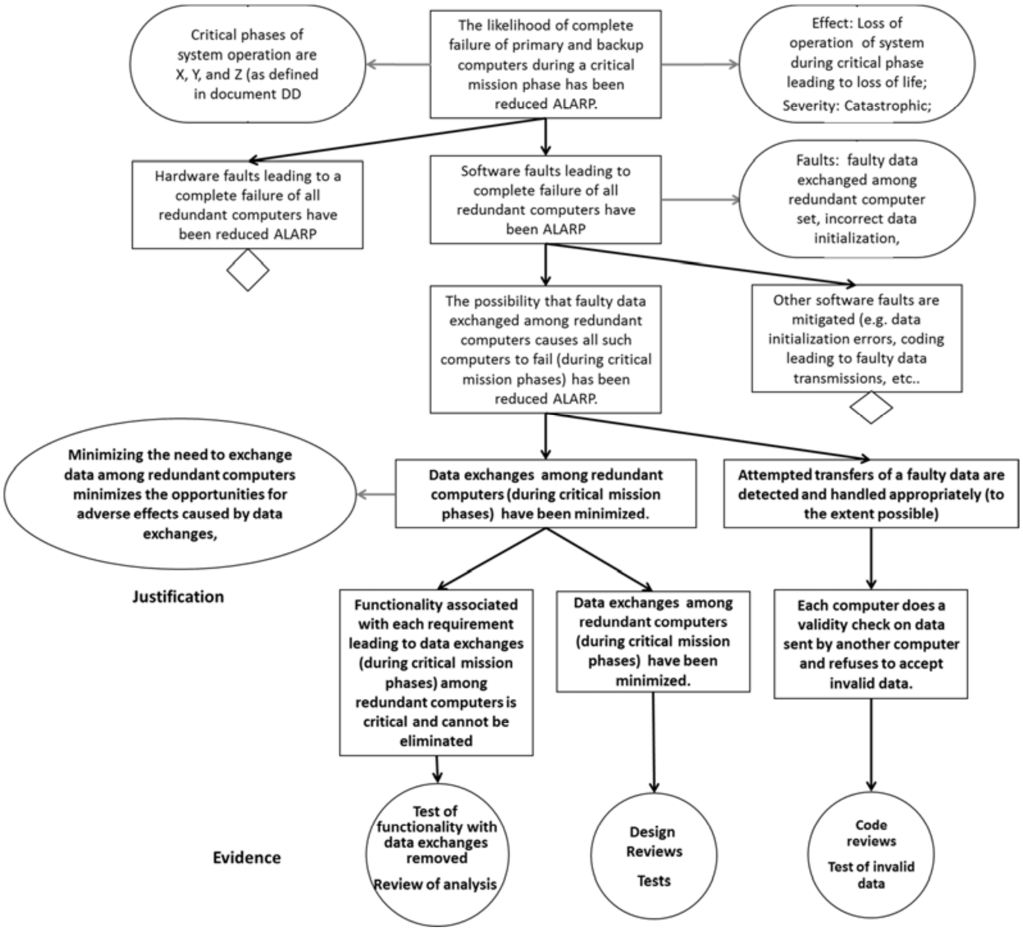

Critical phases of system operation are X, Y, and Z (as defined in document DD
The likelihood of complete failure of primary and backup computers during a critical mission phase has been reduced ALARP.
Effect: Loss of operation of system during critical phase leading to loss of life; Severity: Catastrophic;
Hardware faults leading to a complete failure of all redundant computers have been reduced ALARP
Software faults leading to complete failure of all redundant computers have been ALARP
Faults: faulty data exchanged among redundant computer set, incorrect data initialization,
The possibility that faulty data exchanged among redundant computers causes all such computers to fail (during critical mission phases) has been reduced ALARP.
Other software faults are mitigated (e.g. data initialization errors, coding leading to faulty data transmissions, etc..
Minimizing the need to exchange data among redundant computers minimizes the opportunities for adverse effects caused by data exchanges,
Justification
Data exchanges among redundant computers (during critical mission phases) have been minimized.
Attempted transfers of a faulty data are detected and handled appropriately (to the extent possible)
Functionality associated with each requirement leading to data exchanges (during critical mission phases) among redundant computers is critical and cannot be eliminated
Data exchanges among redundant computers (during critical mission phases) have been minimized.
Each computer does a validity check on data sent by another computer and refuses to accept invalid data.
Evidence
Test of functionality with data exchanges removed
Review of analysis
Design Reviews
Tests
Code reviews
Test of invalid data

Figure 6: Assurance Case for Fault Reduction Report

The claim for the reduction of failures for the redundant computers depends on two subclaims:

- attempts to transfer invalid data are detected and handled appropriately
- only essential exchanges are done

The software safeguards are explicitly identified as the data validity checks made when data is transferred. The functional requirements are not mitigations. The mitigation is removing unnecessary exchanges. Only those data exchanges required by the functional requirements are done. The assurance case also shows the role of the tests and reviews. For example, functionality is tested whenever data exchanges were deleted.

Summary

The infusion pumps' failures to meet the reliability requirements led to health risks. Verifying that those risks have been sufficiently mitigated during the FDA review required evidence that showed how the design of the pump safely mitigated each of the identified hazards. A formal design review of the reliability of a military system also needs to verify that a design adequately mitigates the possible system failures. Such a design analysis requires a concise and understandable way to describe the association between engineering decisions and mitigation of a hazard. As shown with the infusion pump example, an assurance case can convey that information to experts and non-experts.

3 Causal Analysis of Software Failures

The DoD RAM *Guide* recommendation for better engineering analysis applies to the causal analysis of a failure or potential failure. The effectiveness of a design in improving reliability depends on how well the causal analysis has identified and prioritized the threats.

An event-based causal analysis is often used for a hardware failure. Such an analysis identifies the sequence of events that preceded the failure, which are then analyzed to identify a root cause. But such an event-based analysis for failure with a complex system can be misleading. Leveson's analysis of safety failures showed that the causes were frequently the concurrent occurrence of several events. The absence of any one of those events would have prevented the failure.

As example of the weaknesses of an event-based analysis for complex systems, consider the release of methyl isocyanate (MIC) from a Union Carbide chemical plant in Bhopal, India in 1984. A relatively new worker had to wash out some pipes and filters that were clogged. MIC produces large amounts of heat when in contact with water, and the worker did close the valves to isolate the MIC tanks from the pipes and filters being washed. However, a required safety disk which backed up the valves in case they leaked was not inserted. The valves did leak—which lead to 2,000 fatalities and 10,000 permanent injuries. The analysis identified the root cause as an operator error. Charles Perrow's[2] analysis of the Bhopal incident concluded that there was no root cause, and that given the design and operating conditions of the plant, an accident was just waiting to happen. His argument was

> However [water] got in, it would not have caused the severe explosion
>
> - had the refrigeration unit not been disconnected and drained of Freon,
> - or had various steps been taken at the first smell of MIC instead of being put off until the tea break,
> - or had the scrubber been in service,
> - or had the water sprays been designed to go high enough to douse the emissions,
> - or had the flare tower been working and been of sufficient capacity to handle a large excursion.

3.1 2003 Power Grid Blackout

The 2003 power grid blackout was a reliability failure for the power grid control system for an Ohio utility. There had only been minor errors encountered with that control system, and hence reliability-based RAM predictions would have been high. This section describes the engineering review that followed the blackout that came to the opposite conclusion. That review considered the blackout as a software assurance failure and implicitly developed an assurance case to identify the system weaknesses.

---

[2] The Habit of Courting Disaster, Charles Perrow, The Nation (October 1986) 346-356.

3.2 The Power Grid Failure

On August 14, 2003, approximately 50 million electricity consumers in Canada and the northeastern U.S. were subject to a cascading blackout. The events preceding the blackout included a mistake by tree trimmers in Ohio that took three high-voltage lines out of service and a software failure (a race condition[3]) that disabled the computing service that notified the power grid operators of changes in power grid conditions. With the alarm function disabled, the power grid operators did not notice a sequence of power grid failures that eventually lead to the blackout [NERC 2004].

The technical analysis of the blackout explicitly rejected tree-trimming practices and the software race condition as root causes. Instead if we phrase the conclusion like Perrow's, it would be

> However the alarm server failed the blackout would not have occurred
>
> - if the operators had not been unaware of the alarm server failure,
> - or if a regional power grid monitor had not failed,
> - or if the recovery of the alarm service had not failed,
> - or …... [NERC 2004]

A basic understanding power grid reliability requirements and monitoring capabilities is required to analyze the causes and mitigations for the blackout. Power grid operators typically have 30 to 60 minutes to respond to an alarm raised because a generator is out of service or adverse conditions have led to transmission lines being automatically disconnected from the power grid. The technical analysis sponsored by the North American Electric Reliability Corporation (NERC) provides the following summary of the reliability requirements and power grid monitoring activities.

> **Reliability requirement:** The electricity industry has developed and codified a set of mutually reinforcing reliability standards and practices to ensure that system operators are prepared to deal with unexpected system events. The basic assumption underlying these standards and practices is that power system elements will fail or become unavailable in unpredictable ways. The basic principle of reliability management is that "operators must operate to maintain the safety of the system they have available."
>
> **Power grid monitoring:** It is common for reliability coordinators and control areas to use a state estimator to monitor the power system to improve the accuracy over raw telemetered data. The raw data are processed mathematically to make a "best fit" power flow model, which can then be used in other software applications, such as real-time contingency analysis, to simulate various conditions and outages to evaluate the reliability of the power system. Real-time contingency analysis is used to alert operators if the system is operating insecurely; it can be run either on a regular schedule (e.g., every five minutes), when triggered by some system event (e.g., the loss of a power plant or transmission line), or when initiated by an operator [NERC 2004].

---

[3] The software failure was caused by a race condition. An error in the implementation of the software controls that managed access to the data by multiple processes caused the alarm system to stall while processing an event. With that software unable to complete that alarm event and move to the next one, the alarm processor buffer filled and eventually overflowed.

3.2.1 Software Assurance Analysis

The software subsystem that provided audible and visual indications when a significant piece of equipment changed from an acceptable to problematic status failed at 14:14.

- The data required to manage the utility’s power grid continued to be updated on a power grid operator’s control computer.
- After the server failure the power grid operator’s displays did not receive any further alarms, nor were any alarms being printed or posted on the alarm logging facilities.
- The power grid operators assumed that alarm service was operating and did not observe that system conditions were changing.

A key observation by the technical reviewers was that the blackout would not have occurred if the operators had known the alarm service had failed. Instead of analyzing the details of the alarm server failure, the reviewers asked why the following software assurance claim had not been met.

> **Claim:** Power grid operators had sufficient situational awareness to be able to manage it in a manner that meets the reliability requirements.

The blackout analysis then identified multiple ways that the situational awareness claim could be satisfied. Figure 7 shows those possibilities as an assurance case where only one out of the six subclaims is required. For example, a 10-minute recovery time for the alarm server should be sufficient. Responses to adverse power grid conditions can often take an hour or longer. The level of confidence required for an electric utility requires concurrently available alternatives. An implementation of all six is realistic.

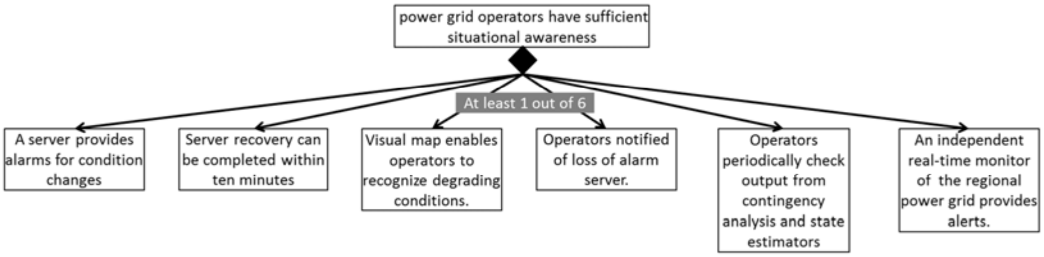

power grid operators have sufficient situational awareness
At least 1 out of 6
A server provides alarms for condition changes
Server recovery can be completed within ten minutes
Visual map enables operators to recognize degrading conditions.
Operators notified of loss of alarm server.
Operators periodically check output from contingency analysis and state estimators
An independent real-time monitor of the regional power grid provides alerts.

Figure 7: Alternate Ways to Provide Situational Awareness

The description of the alternatives and their status at the time of the blackout is as follows:

- The alarm server recovery service was designed for a hardware failure. The alarm service did fail over to the secondary server, but the primary server had stalled because of the race condition prevented it from accessing data. The secondary server was in the same stalled state. A warm reboot of the alarm service failed. The supplier of that control system told the computer support staff that a full restart of the energy management system was required which could take more than 30 minutes.
- A dynamic map board or other type of display that showed data relative to locations on the grid might have enabled the operators to recognize significant line and facility outages within the controlled area. Unlike many transmission system control centers, this utility power management center did not have a map board.

- The power grid operators could have been notified that the alarm service was not available by the computer support staff. But there was no formal procedure for such a notification. It appears that the operators only became aware of the lack of the alarm service about 90 minutes after its failure and only 20 minutes in advance of the final cascade of failures.
- The power grid operators could have learned of the change in conditions by looking at the output of the state estimators and real-time contingency analysis tools. But problems had been experienced with the automatic contingency analysis operation since the system was installed in 1995. As a result, the practice was for the operators or engineers to run contingency analysis manually as needed. Hence the operators did not access the contingency analysis results at any time that day.
- The government contracts with independent organizations to monitor regional sections of the national power grid. The state estimator and network analysis tools at that site for this segment of the power grid were still considered to be in development on August 14 and were not fully capable of automatically recognizing changes in the configuration of the modeled system. The state estimator at the independent monitor went out of service when it failed to deal correctly with the failure of two lines. An operator mistake after the state estimator had been fixed led to a second failure. It did not return to service until after the cascade of failures had started.

The analysis also said the lack of an automatic alarm failure system as one of the causes. An automatic notification existed, likely a heart-beat monitor that notified the secondary server when the primary one had failed. It should have been easy to use the same mechanism to also notify the computer support staff and the power grid operators in the event of a failure of one or both alert servers.

3.2.2 Other Observations

The power grid blackout is an example of a number of the software reliability challenges listed in Table 1.

- A recovery from a failure mode corresponding to a software fault in the alarm server had not been considered. The IT support staff only determined on the day of the blackout that a full control system reboot would be required rather than just a restart of the alarm server.
- The guidance in Table 1 recommends that the design for system recovery should assume that failures could arise from currently unknown or rarely occurring faults. Software perfection was implicitly assumed by the utility from two perspectives. There was likely over confidence in the alarm server software and implicitly in the commercial organization that developed it. The IT support staff said they had encountered only minor problems with the alarm server. But the alarm software supported multi-tasking which should automatically raise reliability concerns. Careful engineering is required to avoid race conditions when accessing and modifying shared data. If the engineering choice is to use semaphores, experience shows that race conditions are likely. Race conditions had not been observed with this software. A fault such as a race condition is easy to ignore as it will occur only for a very specific set of operating condition which may never occur. The only safe assumption is that unanticipated failures will occur.

- The utility had not explicitly considered how to continue operations if the alarm server recovery failed. The assurance case shown in Figure 7 demonstrated that a design that was very resilient, i.e., could tolerate the failure of any six of the claims.

3.3 Sustainment

As described in Section 1.2 the risk of a system failure can increase over time because of changes in operational conditions and work processes. An assurance case that documents the assumptions, argument, and evidence that justify a claim can be used to monitor how changes affect the confidence associated with a claim.

As an example, consider the assurance case for the utility shown in Figure 7. The utility did not provide a visual map, there was no requirement for the computer support staff to notify the operators of failures in the alarm service, and the alarm service recovery was designed only for a hardware failure. The three remaining alternatives, the alarm service, operators monitoring the contingency analysis, and the independent monitoring capability, were assumed by the utility to provide sufficient resiliency. But at some point after the installation of the control system, problems occurred with the automatic execution of the contingency analysis tool. The loss of that automatic analysis would leave the utility dependent on a single internal resource, the alarm service. But the resultant significant reduction in resiliency did not appear to be considered when the decision was made that the operators should manually run that analysis only when needed. Now only two of the alternatives listed in the assurance case remained. Both of those alternatives failed the day of the blackout. A simple analysis of the assurance case supports a conclusion that a blackout was just ready to happen. That conclusion was strengthened as the NERC analysis team found deficiencies other than those that caused the 2003 blackout that—under different circumstances—could also have led to a blackout.

Summary

The 2003 electric power grid blackout is a good example of the kind of causal analysis that is required to improve the reliability of complex systems. A failure analysis based just on events could have concluded that the primary cause of the blackout was the failure of the alarm server caused by a race condition. The reliability problems were far more serious as the operational resiliency was so poor that a blackout was just ready to happen.

4 Analyzing the Confidence of an Assurance Case

The participants in the recommended formal design reviews have to decide if they are confident that a proposed design will satisfy reliability requirements. An objective for incorporating software assurance into that review is for that judgment to be based on more than opinion. An assurance case provides a way to systematically do the analysis.

How can we determine the confidence that a system will behave as expected? As noted in the discussion of the power grid blackout, a combination of conditions is frequently the cause of a software system failure. It is impossible to examine every possible combination of conditions that could affect a system.

But achieving that confidence is important to those acquiring the system, those developing the system, and those using the system. Such confidence should be based on concrete evidence and not just on an opinion of the developers or reviewers. An assurance case provides the argument and evidence. Our level of confidence depends on understanding which evidence leads to an increase in the confidence that a system property holds and why specific evidence increases the confidence.

There are examples where confidence can be quantified. A Safety Integrity Level (SIL) is an example of a well-defined confidence measure that has been applied to hardware devices. An SIL for a device is based on a risk analysis relative to a specific dangerous failure. But as noted in Goodenough (2012), confidence for software intensive systems is a slippery subject [Goodenough 2012]. There is a subjective aspect to it, such as “I know that method is the best option.” We need a precise and understandable definition of confidence in order to know where apply scarce system development resources. Which aspects of a design most affect our level of confidence? Ongoing research has proposed several ways to analyze confidence for a software intensive system.

4.1 Eliminative Induction

One approach for confidence is implicitly applied during a system review and most likely during development. Instead of estimating the likelihood that a claim is true, consider the probability that the claim is false. For example, ask why the argument and evidence provided by a developer might be insufficient to justify the claim. For example,

- The test plans did not include all of the hazards identified during design.
- The web application developers had limited security experience.
- The acquirer did not provide sufficient data to validate the modeling and simulations.
- Integration testing did not adequately test recovery following component failures.

It is not at all obvious, but such an approach is constructing an alternate assurance case for the same claim. Instead of constructing an argument for the validity of a claim, we identify the various possibilities for why the claim is false. An assurance case consists of gathering evidence or performing analysis that removes those possibilities. The SEI refers to the graphical representation of the assurance case created by eliminating doubts as a confidence map. Each eliminated possibility removes a reason for doubt and thereby increases our confidence in the claim. The expectation during a review is that the developer is able to show how to eliminate the doubts that are raised.

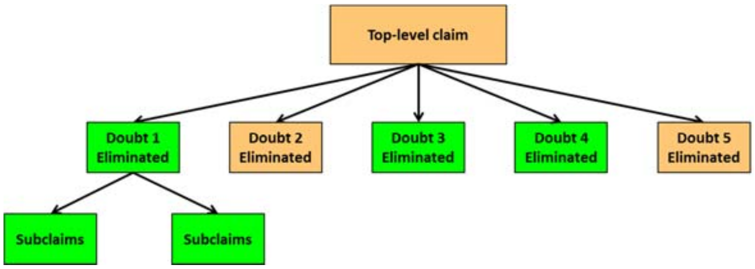


Figure 8: Confidence Map

A claim is tentative. We cannot deductively prove using the argument in an assurance case that the evidence **E** proves the claim **C** is true. Additional information could show that it is false. Rather our logic looks like

if **E** then (usually) **C** unless **R, S, T, …**

where **R,S,T…** are exceptions. The doubts identified during design review are the potential exceptions to the claim. Removing doubts about a claim is called eliminative induction. These exceptions are called defeaters. Each defeater is a source of doubt about the truth of a claim. The research done for confidence has identified three classes of defeaters that are applicable for a justification of an assurance claim. An assurance justification has the form

Argument shows *Evidence* confirms *Claim*

The three classes of defeaters are

1. **Doubt the claim:** There is information that contradicts or rebuts a claim. The cause can be a combination of a poor argument and insufficient evidence. Referred to as rebutters in the literature.
2. **Doubt the argument:** There are specific conditions under which the claim is not necessarily true even though the premises (i.e., evidence) are true. Such conditions create doubts or undercut the validity of the argument. We can doubt the inference among claims or between a claim and its supporting evidence. Referred to as undercutters in the literature.
3. **Doubt the evidence:** There are conditions that invalidate one or more of the premises. The argument is valid, but insufficient evidence weakens or undermines our confidence in the claim. Referred to as underminers in the literature.

Removing doubts just inverts an assurance case. Consider the assurance case shown in Figure 9 for the claim that flipping a switch will turn the light on.[4] The assurance argument now is if those failures are eliminated then the light will turn on.

---

[4] http://blog.sei.cmu.edu/post.cfm/eliminative-argumentation-a-means-for-assuring-confidence-in-safety-critical-systems

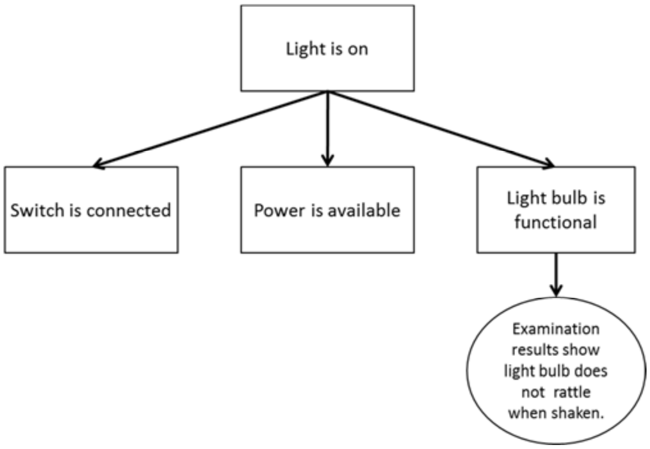

Light is on
Switch is connected
Power is available
Light bulb is functional
Examination results show light bulb does not rattle when shaken.

Figure 9: Light Bulb Example

The claim is invalid if there is no power, if the switch is not connected, or if the bulb is defective. Those conditions rebut the claim as shown Figure 10. The assurance argument now is if those failures are eliminated then the light will turn on. The confidence of the assurance now depends on our confidence in the statement that there are no other reasons for a failure.

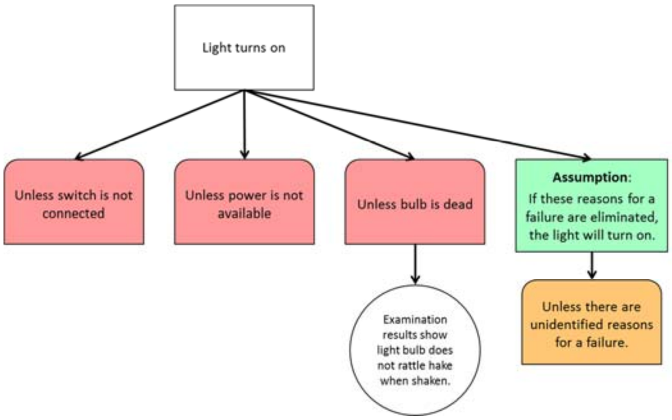

Light turns on
Unless switch is not connected
Unless power is not available
Unless bulb is dead
Assumption: If these reasons for a failure are eliminated, the light will turn on.
Examination results show light bulb does not rattle hake when shaken.
Unless there are unidentified reasons for a failure.

Figure 10: Light Bulb Rebutters

Doubts can also be raised on the evidence that a light bulb is functional as shown in Figure 11. The evidence given for a functional light bulb is that that it does not rattle when shaken, i.e., the filament is intact. The validity of that evidence can be undermined by an examiner who is hard of hearing or has headphones on. The validity of the evidence argument is undercut by an LED bulb as such does not rattle when defective.

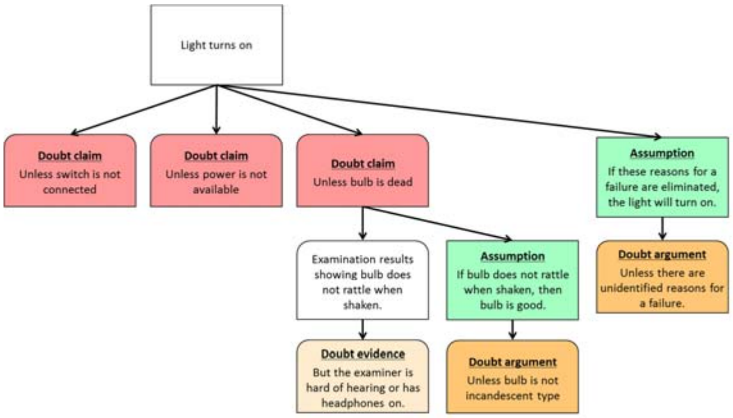

Light turns on
Doubt claim
Unless switch is not connected
Doubt claim
Unless power is not available
Doubt claim
Unless bulb is dead
Assumption
If these reasons for a failure are eliminated, the light will turn on.
Examination results showing bulb does not rattle when shaken.
Assumption
If bulb does not rattle when shaken, then bulb is good.
Doubt argument
Unless there are unidentified reasons for a failure.
Doubt evidence
But the examiner is hard of hearing or has headphones on.
Doubt argument
Unless bulb is not incandescent type

Figure 11: Expanded Confidence Map

4.2 Reliability Validation and Improvement Framework

Embedded software responsible for system safety and reliability is experiencing exponential growth in complexity and size [Leveson 2004a, Dvorak 2009], making it a challenge to qualify and certify the systems [Boydston 2009] and exceeding the capabilities of experts to justify a level of confidence based on their analysis. Confidence is increased when expertise can be replaced by formal analysis.

The SEI has developed a reliability validation and improvement framework to provide a foundation for addressing the challenges of qualifying software-reliant safety-critical systems.[5] The framework draws on multiple software studies to identify four technologies that are incorporated into the framework.

Table 5: Reliability Framework Technologies

Specification of system and software requirements in a manner to allow for completeness and consistency checking as well as other predictive analyses: For example, group requirements into mission requirements (operation under nominal conditions) and safety-criticality requirements (operation under hazardous conditions) rather using the more traditional grouping of functional and nonfunctional requirements.
Use of architecture-centric, model-based engineering to model intended (managed) interactions between system components, including interactions among the physical system, the computer system, and the embedded software system
Use of static analysis in the form of formal methods to complement testing and simulation as evidence of meeting mission and safety-criticality requirements. Such analysis can validate completeness and consistency of system requirements, architectural designs, detailed designs, and implementation and ensure that requirements and design constraints are met.
Use of system and software assurance throughout the development lifecycle to provide justified confidence in claims supported by evidence that mission and safety-criticality requirements have been met by the system design and implementation. Assurance cases systematically manage such evidence (e.g., reviews, static analysis, and testing) and take into consideration the context and assumptions.

The next section suggests how aspects of these four technologies can be incorporated into a software reliability improvement plan for non-safety critical systems.

---

[5] Funded by U.S. Army Aviation and Missile Research Development and Engineering Center (AMRDEC) Aviation Engineering Directorate (AED)

4.3 Incorporating into Design Reviews

Activities in the design phase of a custom developed system should identify possible failure modes and how they might affect operations. The causes of such failure modes can be a combination of those from the operational environment and those associated with software defects.

The design for a software-intensive system is usually an incremental process. A proposed design that mitigates a particular failure mode can introduce new ones. The expense of a high-assurance mitigation may lead to requirement changes or require additional time to evaluate less-expensive alternatives.

The conclusion of a formal design review should be based on more than opinion. An assurance case provides a way to systematically do the analysis. Software intensive systems are complex, and it should not be surprising that the analysis done by even an expert designer could be incomplete and has overlooked a hazard or made simplifying but invalid development and operating assumptions. The use of eliminative induction as described in Section 4.1 provides a systematic way to look for exceptions for the claim, the evidence, or the arguments used to justify the engineering decisions incorporated in a design.

Summary

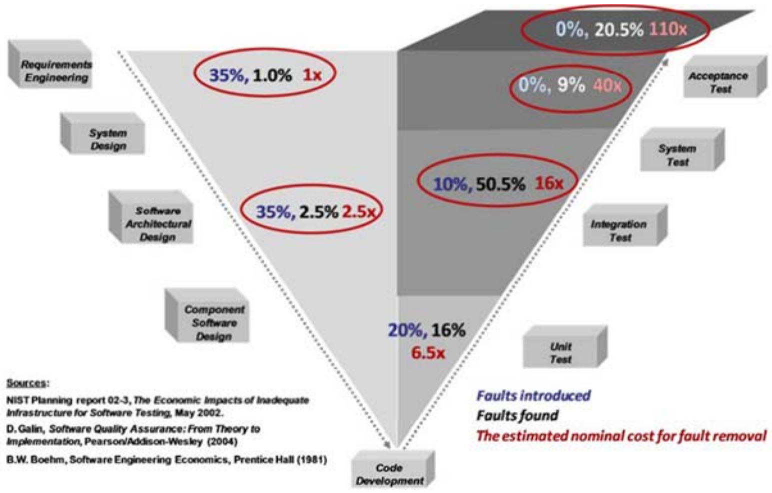
The organization of an assurance case is often organized around claims that are derived from requirements. The subclaims are positive statements such as “The health risk of a drug overdose has been sufficiently mitigated.” To find the mistakes that might have occurred in constructing the assurance case, raise doubts about the subclaims, the arguments for the subclaims, and about the evidence provided.

5 Assuring Software Reliability

Meeting software reliability requirements requires a different approach than that used for hardware reliability. The failure distribution curves as shown in Figure 1 are quite different as hardware failures over time are associated with wear while software failures result from changes in usage, in operating conditions, or new features added by a software upgrade. A software system with defects can operate perfectly for long period of time until what might be a rare combination of conditions lead to a failure. The analysis a software failure should not assume there is a root cause. Reliability requirements often involve both hardware and software as with infusion pumps. The pump’s software also has to manage faults created by a sensor failure. Software reliability for military systems typically involves both hardware and software faults.

There are lessons that can be learned from hardware reliability analysis. Techniques such as modeling and simulations are frequently applied for hardware reliability. The objective is to identify potential stress points during the design of the hardware component and not after its assembly. The same techniques can be applied to eliminate software reliability defects during requirements and design phases of the system development lifecycle.

A summary of the results of a number of studies on where errors are introduced in the development lifecycle, when they are discovered, and the cost of the resulting rework is shown in Figure 12 [Feiler 2012]. Requirement and design errors dominate.


0%, 20.5% 110x
Requirements Engineering
35%, 1.0% 1x
0%, 9% 40x
Acceptance Test
System Design
System Test
10%, 50.5% 16x
Software Architectural Design
35%, 2.5% 2.5x
Integration Test
Component Software Design
20%, 16% 6.5x
Unit Test
Faults introduced
Faults found
The estimated nominal cost for fault removal
Code Development

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 12: Error Leakage Rates Across Development Phases

5.1 Requirement Errors

Requirement problems are not unique to reliability. One study [Hayes 2003] showed that the top six categories of requirements errors are

1. omitted/missing requirements (33%)
2. incorrect requirements (24%)
3. incomplete requirements (21%)

4. ambiguous requirements (6.3%)
5. over specified requirements (6.1%)
6. inconsistent requirements (4.7%).

The DoD RAM *Guide* warning on unrealistic or poorly defined requirements is also applicable. Recovery from a failure for a software intensive system can depend on determining the cause which may not be feasible under the operating conditions. An acquisition could require automated diagnostics, but complex systems unfortunately can fail in complex ways that can be beyond the capabilities of automatic recovery mechanisms. Requirements for alternatives for a disabled service such as those shown for the power grid failure should also be considered. The development of detailed mission scenarios such as those for close air support that identify known system failures assists both the acquirer and developer [Ellison 2010].

An example of missing requirement could be omitting fault tolerance requirements for a user application. For example, consider the design of an application that supports multiple user actions, each displayed in its own window. A web browser is a good example. Reliability is degraded if a failure in any one of those activities leads to a failure for all the user actions. Google's Chrome browser improves reliability by using a separate process thread for each window. A failure now should only affect one user activity.

A user application with an availability requirement should also include a requirement verifying that requirement. We know how to use simulations and models to justify that a software design satisfies performance claims. Such evidence would most likely be required when computing resources are constrained, but we might not ask for such a justification for a user application. For example, computing resources are allocated for each instance of a browser window. Multiple windows could be competing for access to the same service. How well are resources managed as windows are created and deleted? A requirement for a justification such as by simulation of the proposed approach during design avoids surprising behavior when the application is deployed.

As part of a requirement elicitation, an acquisition should consider the impact on the mission for possible system failure states and the desired recovery. A recommendation that was noted in Section 4.2 of Feiler (2012) is to group requirements into mission requirements (operation under nominal conditions) and requirements for operating under adverse conditions, rather using the more traditional grouping of functional and nonfunctional requirements [Feiler 2012]. Specifying operations for normal and adverse conditions is essential for mission threads such as close air support (CAS) and time sensitive targeting (TST). An analysis described in Ellison identified a gap between theory and practice [Ellison 2010]. The DoD mission thread documentation represented an "idealized" view of the operational environment; the documentation rarely considered possible failures and often assumed significant homogeneity of computing infrastructure and military hardware. In practice, a successful execution of these mission threads depended on using available equipment and often on ad hoc measures to work around resource limitations. Recovery requirements had to reflect the resources available.

5.2 Available Expertise

Contracting decisions can avert effort from the approaches that can increase software reliability. For example, the U.S. Army Materiel Systems Analysis Activity (AMSAA) provides funding for hardware reliability-improvement programs that use modeling, analysis, and simulation to identify and reduce design defects before the system is built. No such reliability improvement programs existed for software. Instead AMSAA funding for software focuses on finding and removing code faults through code inspection and testing [Goodenough 2010].

But funding a software reliability improvement program is realistic only if the recipient knows what could supplement code inspections and testing. For example, how could we apply modeling, an analysis technique, or simulation to evaluate a software design before a system is implemented? The technologies listed in Table 5 improve software reliability for safety-critical systems but at this time are on the very leading edge of practice.

In all likelihood, most of manufacturers of infusion pumps thought they were doing a good job and had difficulty finding examples for how software reliability for such devices could be improved in advance of testing.

5.2.1 Fault Tolerance Example

The domain expertise required to analyze the power grid failure was readily available. Equivalent expertise required to analyze the reliability for complex system faults is likely not as prevalent.

The objectives of a NASA website[6] that described actual electronic systems failure scenarios suggests that for some aspects of system assurance, that expertise can be hard to find.

The objective of that effort is to

- Provide a publicly accessible body of knowledge that can be used to transfer experience among designers.
- Identify failures which are addressed or not addressed by current formal methodologies and tools.

The scenarios selection criteria include

- scenarios that many designers believe cannot happen
- scenarios not published elsewhere or with lessons learned omitted

The importance of experience is demonstrated in the following example of a NASA controller failure.

The controller required fault tolerance for a processor failure. The design used redundancy in the form of four processors and a voting mechanism. But a failure occurred when a technician replaced a resistor with one with the wrong value in a bus terminator for the bus used by the four control computers. The incorrect resistor value led those controllers to process the data differently depending on where they were connected to that bus. The result was a 2-2 voting tie.

---

[6] https://c3.nasa.gov/dashlink/resources/624/

The root cause was not the technician's error, but a software failure. All processors produced correct output based on their input, but two had been eliminated in the voting. The voting mechanism was designed to handle processor failures and had not considered faults which could lead to different inputs to the four processors. This could be an example of a fault that a designer believed could not happen.

According to the analysis provided by HP, formal analysis methods existed to resolve such conflicts (the well-known Byzantine fault). If the software design had include one of those methods, the processors would have stayed in agreement and the failures would not have occurred. Fortunately, the failure occurred in the laboratory.

It is likely that only those with extensive experience with such failures would have identified the omission of such faults during a design review. This example supports the DoD RAM *Guide's* recommendation for formal design reviews and emphasizes the importance of using reviewers with extensive domain experience.

5.2.2 Improving Availability of Security Expertise

Efforts to improve software security in acquired software also encountered problems with limited sources for the necessary expertise. But there are now resources available that enable a software development organization to reduce security risks during development.

An organization that wants to learn which software defects could be exploited only has to review MITRE's Common Weakness Enumeration (CWE), which has cataloged more than 900 software weaknesses that have been used in successful attacks. In addition, the CWE suggests ways to mitigate each of the weaknesses.

Building Security In Maturity Model (BSIMM) provides examples of current secure software development practices used by a group of large corporations [BSIMM 2013]. By the fall of 2013, the BSIMM had surveyed 52 large corporations who were executing security improvement initiatives. The BSIMM objective was not to find the best practices but simply to document current practice and track the changes in those practices over time. The BSIMM collection of practices is characterized more by diversity than commonality. The commonality among the surveyed organizations was more in terms of objectives for training, for maintaining knowledge of attacks, and for capabilities required for architecture analysis and configuration management. Examples of some of the objectives are shown in Table 6.

The entries in that table suggest equivalent contractor capabilities that could be appropriate for software reliability. For example, the items listed under Strategy and Metrics are applicable to software reliability. The reliability equivalent to Attack Model activities could be building and maintaining a repository of design, implementation, testing, and integration guidance for reliability issues that could occur in the types of systems an organization builds. The objectives for Security Features and Design in Table 6 are applicable for reliability. Extensive experience and expertise is necessary to resolve a number of the reliability issues for complex systems.

Table 6: BSIMM Examples

Objective	Activities
Strategy and Metrics	
Attain a common understanding of direction and strategy	Publish process Establish checkpoints compatible with existing development practices and begin gathering the input necessary for making a go/no go decision
Align behavior with strategy and verify adherence	Publish data about software security internally Enforce checkpoints with measurements and track exceptions
Attack Models	
Create attack and data asset knowledge base	Build and maintain a top N possible attacks list Collect and publish attack stories – increases awareness
Provide information on attackers and relevant attacks	Build attack patterns and abuse cases tied to potential attackers Create technology-specific attack patterns
Security Features and Design	
Publish security features and architectures	Build and publish security features – build it once and reuse it
Build and identify security solutions	Create capability to solve difficult design problems

Summary

We do not have the same breadth of information on hazards and mitigations for software reliability as we have for software security, which means that improving the reliability of software systems will have a learning curve for both acquirers and suppliers. Descriptions and analysis of real system failures are valuable, such as those that appear on the NASA website referenced in section 5.2.1.

6 Conclusion

Software reliability is the probability that no failures occur over a period of time. Mitigating specific high-risk faults is the province of system and software assurance. The overall assurance of systems can be improved by assessing it during the development of a system. Requirement and design errors not found until testing and system integration are expensive to rectify. Showing during a design review how an engineering decision mitigates a specific hazard should reduce the occurrence of design errors being found late in the development lifecycle.

Doing such engineering analysis depends on having a concise and understandable way to describe the associations among engineering decisions and fault management. An assurance case provides a concise and understandable way to describe them to experts and non-experts, and assurance case analysis techniques such as eliminative induction can provide specific reasons for a design weakness.

Improving the reliability of software systems will have a learning curve for both acquirers and suppliers. The failure of a complex system is likely the result of a concurrence of multiple events. There are currently only limited resources available to help developers analyze and mitigate such failures. The limited knowledge base also increases the difficulties for an acquirer to determine the feasibility of reliability requirements.

This section concludes with a discussion of SEI experience applying assurance case techniques to the early phases of the system development lifecycle of a DoD system. That experience suggests that the assurance case technique is a powerful tool for analyzing systems. Assurance cases give managers answers about design progress that are demonstrably rooted in facts and data instead of opinions based on hope and best intentions. Techniques such as the confidence map described in Section 4 provide a concise and understandable way to show the effects of a specific development shortfall and to track progress between reviews.

The SEI has applied assurance case techniques in the early phases of system development life cycle for a large DoD system of systems (SoS) as described in Blanchette [Blanchette 2009]. The general approach is applicable for less complex systems. The SEI team analyzed the software contributions to the definitive characterization of operational needs – the SoS key performance parameters (KPPs). Within the DoD, KPPs are the system characteristics essential for delivery of an effective military capability. All DoD projects have some number of KPPs to satisfy in order to be considered acceptable from an operational perspective. For example, any DoD system that must send or receive information externally is required to fulfill the Net-Ready KPP (NR-KPP). The top claim is that the SoS supports Net-Centric military operations. The subclaims of that node are

- The SoS is able to enter and be managed in the network.
- The SoS is able to exchange data in a secure manner to enhance mission effectiveness.
- The SoS continuously provides survivable, interoperable, secure, and operationally effective information exchanges.

When performing an assurance case analysis of a completed design, the outcome is rather black-and-white: either design artifacts are complete and sufficient, or they are not. Reviewing an in-progress design requires a more nuanced approach, one that reflects relative risk, since the design artifacts will necessarily be in different stages of completion. For this example, the SEI used a simple and familiar stoplight approach to scoring (so named for the red-yellow-green coloring), where the color red designates a relatively high risk area, the color yellow designates a relatively medium risk area, and the color green indicates a relatively low risk area. The rules for assigning colors are slightly different at the evidence level than they are at the level of the claims, as is shown in Figure 13.

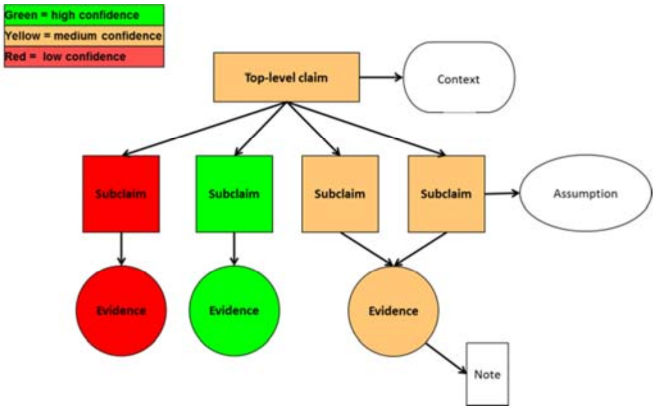


Figure 13: Scoring Legend

When the subclaims are not all uniformly the same color, an analyst must make a subjective decision on the risk to assign to a node. For example, an analyst might conclude a medium risk given the following doubts raised about the evidence and arguments.

1. Only a subset of information exchanges has been implemented to date.
2. The noted risks are, at best, medium at this time.
3. The security architecture has not been completely propagated across the SoS.
4. An evaluation of the security architecture revealed some design choices that will prevent system accreditation.
5. Preliminary field tests indicate some information exchanges are exceeding prescribed timelines for completion.

The overall analysis tree might appear as shown in Figure 14 in a confidence map. The color assigned represents an analyst’s judgment on the seriousness of the doubts identified for a specific claim. It can provide both program and developer managers a sort of roadmap for prioritizing and addressing the issues.

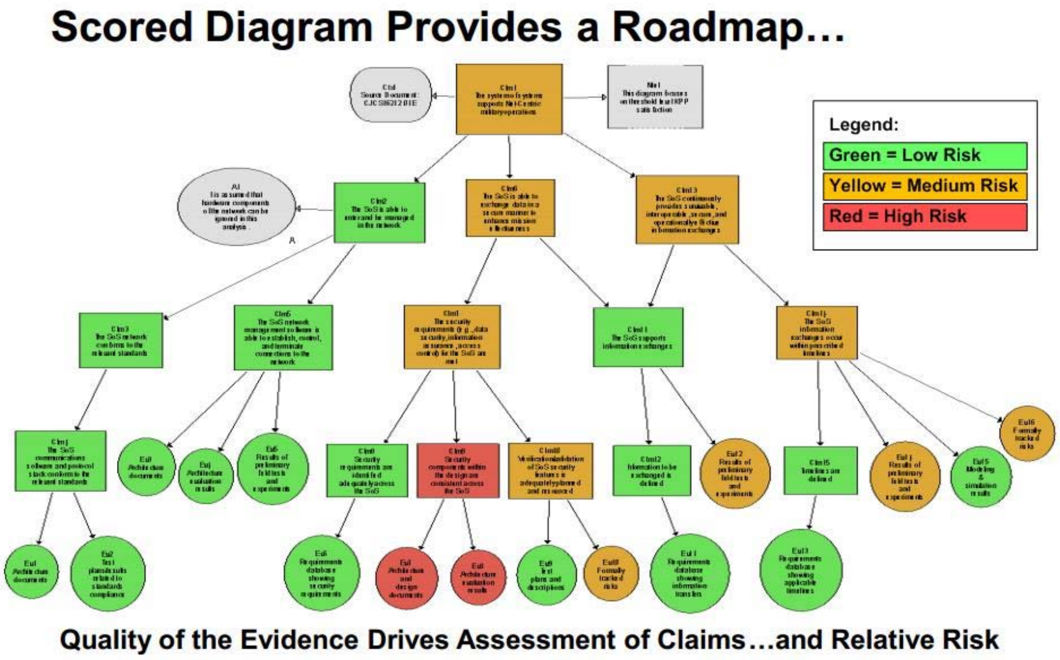

Scored Diagram Provides a Roadmap…
Legend:
Green = Low Risk
Yellow = Medium Risk
Red = High Risk
Quality of the Evidence Drives Assessment of Claims…and Relative Risk

Figure 14: KPP Scored Diagram

For example, an item may be red because

- It is scheduled to be addressed at a later date.
- The contractor is significantly behind schedule.
- A redesign is required because of changes in requirements.
- The problem is harder than anticipated.
- There is a significant risk that the current approach will not meet requirements: The reason can be a poor design or unrealistic requirements.

Such a confidence map provides a concise and understandable way to show the effects of a specific development shortfall and to track progress between reviews.

Experience with actual projects suggests that the assurance case technique is a powerful tool for analyzing a large and complex SoS software design. It provides a means of taking a crosscutting look at a SoS, a perspective often achieved only with great effort even in less complex development projects. Assurance cases give managers answers about design progress that are demonstrably rooted in facts and data instead of opinions based on hope and best intentions.

References/Bibliography

URLs are valid as of the publication date of this document.

[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://resources.sei.cmu.edu/asset_files/WhitePaper/2009_019_001_29066.pdf

[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.
https://wiki.sei.cmu.edu/aadl/images/e/e6/Qualification_and_Reliability_of_Complex_Rotorcraft_Systems-A.pdf

[BSIMM 2013]
The Building Security In Maturity Model. 2013. http://bsimm.com/

[Conquet 2008]
Conquet, Eric. "ASSERT: A Step Towards Reliable and Scientific System and Software Engineering." *Proceedings of 4th International Congress on Embedded Real-Time Systems* (ERTS 2008). Toulouse (France), January–February 2008. Societe des Ingenieurs de l'Automobile, 2008.
http://www.sia.fr/dyn/publications_detail.asp?codepublication=R-2008-01-2A04

[Dvorak 2009]
Dvorak, Daniel L., ed. *NASA Study on Flight Software Complexity* (NASA/CR-2005-213912). Office of Chief Engineer Technical Excellence Program, NASA, 2009.

[Ellison 2010]
Ellison, Robert; & Woody, Carol. *Survivability Analysis Framework* (CMU/SEI-2010-TN-013). Software Engineering Institute, Carnegie Mellon University, 2010.
http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9323

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

[Feiler 2009a]
Feiler, Peter; Hansson, Jörgen; de Niz, Dionisio; & Wrage, Lutz. *System Architecture Virtual Integration: An Industrial Case Study* (CMU/SEI-2009-TR-017). Software Engineering Institute, Carnegie Mellon University, 2009.
http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9145

[Feiler 2012]
Feiler, Peter; Goodenough, John; Gurfinkel, Arie; Weinstock, Charles; & Wrage, Lutz. *Reliability Improvement and Validation Framework* (CMU/SEI-2012-SR-013). Software Engineering Institute, Carnegie Mellon University, 2012.
http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=34069

[Goodenough 2010]
Goodenough, J. B. *Evaluating Software's Impact on System and System of Systems Reliability*. Software Engineering Institute, Carnegie Mellon University, March 2010.
http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=28933

[Hayes 2003]
Hayes, J. H. "Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project," 49–59. *Proceedings of the IEEE International Symposium on Software Reliability Engineering* (ISSRE). Denver, CO, November 2003. IEEE, 2003.

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

[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.

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

[Leveson 2011]
Engineering a Safer World, Nancy G. Leveson, The MIT Press, Cambridge, MA, 2011

[NERC 2004]
North American Electric Reliability Corporation. *Technical Analysis of the August 14, 2003 Blackout.* http://www.nerc.com/docs/docs/blackout/NERC_Final_Blackout_Report_07_13_04.pdf

[Open Group 2013]
The Open Group. *Dependability Through Assuredness™ (O-DA) Framework*. 2013.
https://www2.opengroup.org/ogsys/catalog/C13F

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
August 2014

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Assuring Software Reliability

5. FUNDING NUMBERS
FA8721-05-C-0003

6. AUTHOR(S)
Robert J. Ellison

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2014-SR-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFLCMC/PZE/Hanscom
Enterprise Acquisition Division
20 Schilling Circle
Building 1305
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

The 2005 Department of Defense Guide for Achieving Reliability, Availability, and Maintainability (RAM) recommended an emphasis on engineering analysis with formal design reviews with less reliance on RAM predictions. A number of studies have shown the limitations of current system development practices for meeting these recommendations. This document describes ways that the analysis of the potential impact of software failures (regardless of cause) can be incorporated into development and acquisition practices through the use of software assurance.

14. SUBJECT TERMS
Software reliability, software assurance, assurance case, software failure

15. NUMBER OF PAGES
47

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102