Software Engineering Institute

Reliability Validation and Improvement Framework

Peter H. Feiler
John B. Goodenough
Arie Gurfinkel
Charles B. Weinstock
Lutz Wrage

November 2012

SPECIAL REPORT
CMU/SEI-2012-SR-013

Research, Technology, and System Solutions Program

http://www.sei.cmu.edu

Carnegie Mellon

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

This report was prepared for the

SEI Administrative Agent
AFLCMC/PZE
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0000070

.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work was performed under funding from the U.S. Army Aviation and Missile Research Development and Engineering Center (AMRDEC) Aviation Engineering Directorate (AED).

Executive Summary

Rotorcraft and other military and commercial aircraft rely increasingly on complex and highly integrated hardware and software systems for safe and successful mission operation as they undergo migration from federated systems to Integrated Modular Avionics (IMA) architectures. The current software engineering practice of "build then test" is proving unaffordable; software costs for the latest generation commercial aircraft, for example, are expected to exceed $10B [Feiler 2009a] despite the use of process standards and best practices and the incorporation of a safety culture. In particular, 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].

The U.S. Army Aviation and Missile Research Development and Engineering Center (AMRDEC) Aviation Engineering Directorate (AED) has funded the Carnegie Mellon® Software Engineering Institute (SEI) to develop a reliability validation and improvement framework. The purpose of this framework is to provide a foundation for addressing the challenges of qualifying increasingly software-reliant, safety-critical systems. It aims to overcome the limitations of current reliability engineering approaches, leverage the best emerging engineering technologies and practices to complement the process focus of current practice, find acceptance in industry, and lead to a new set of reliability improvement metrics. In this report, we

- summarize the findings of the background research for the framework in terms of key challenges in qualifying safety-critical, software-reliant systems
- discuss an engineering framework for reliability validation and improvement that integrates several engineering technologies
- outline a new set of metrics that focus on cost-effective reliability improvement
- describe opportunities to leverage ongoing industry and standards efforts and potential follow-on activities specific to the U.S. Army, to accelerate adoption of the changes in engineering and qualification practice described above

Reliability engineering has its roots in hardware reliability assessment that uses historical data from slowly evolving system designs. Hardware reliability is a function of time, accounting for the wear of mechanical parts. In contrast, software reliability is primarily driven by design defects—resulting in a failure distribution curve that does not adhere to the bathtub curve common for physical systems.[1]

Often the reliability of the software is assumed to be perfect and to behave deterministically—that is, to produce the same result given the same input [Goodenough 2010]. Therefore, the focus in software development has been on testing to discover and remove bugs using various test coverage metrics to determine test sufficiency. However, time-sensitive software component interact-

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[1] 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. For more information, go to http://en.wikipedia.org/wiki/Bathtub_curve.

tions may encounter race conditions, unexpected latency jitter, and unanticipated resource contention—all of which can occur randomly. In attempts to predict sufficiency, engineers have used a failure-probability density function based on code metrics such as source lines of code and cyclomatic (or conditional) complexity. However, this function is not a good measure of system-level interaction complexity and nonfunctional properties such as performance or safety.

Insertion of corrections into operational systems to address software-related problems requires recertification. Frequently, operational work-arounds must be accepted in lieu of correction due to high recertification cost. As a result, operators spend up to 75% of their time performing work-arounds. Clearly, a need exists to improve system recertification.

As with hardware, a reliability improvement program for software-reliant systems is needed that includes modeling, analysis, and simulation. This type of improvement program can identify design defects before a system is built and design for robustness to counter unplanned usage and hazard conditions [Goodenough 2010].

Studies 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] have identified four key technologies in addressing these challenges:

1. specification of system and software requirements in terms of both a mission-critical system perspective (function, behavior, performance) and safety-critical system perspective (reliability, safety, security) in the context of a system architecture to allow for completeness and consistency checking as well as other predictive analyses
2. architecture-centric, model-based engineering using a model representation with well-defined semantics to characterize the system and software architectures in terms of intended (managed) interactions between system components, including interactions among the physical system, the computer system, and the embedded software system. When annotated with analysis-specific information, the model becomes the primary source for incremental validation with consistency along multiple analysis dimensions through virtual integration.
3. use of static analysis in the form of formal methods to complement testing and simulation as evidence of meeting mission and safety-criticality requirements.[2] Analysis results can validate completeness and consistency of system requirements, architectural designs, detailed designs, and implementation and ensure that requirements and design constraints are met early and throughout the life cycle.
4. use of system and software assurance throughout the development life cycle 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.

Research and industry initiatives are integrating and maturing these technologies into improved

[2] In this report, we group requirements into mission requirements (operation under nominal conditions) and safety-criticality requirements (operation under hazardous conditions) rather than the more traditional grouping of functional and nonfunctional requirements. See Section 3.1 for more detail.

software-reliant system engineering practice. The SAE International[3] Architecture Analysis and Design Language (AADL) standard has drawn on research funded by the Defense Advanced Research Project Agency (DARPA) in architecture description languages (ADLs) [SAE 2004-2012].The Automated proof-based System and Software Engineering for Real-Time applications (ASSERT) initiative was led by the European Space Agency (ESA) and focused on representing two families of satellite architectures in AADL, validating them, and generating implementations from the validated architectures [Conquet 2008]. The European Support for Predictable Integration of mission Critical Embedded Systems (SPICES) initiative integrated AADL with formalized requirement specification, the Common Object Request Broker Architecture (CORBA) Component Model (CCM), and SystemC. The result was an engineering framework for formal analysis and generation of implementations [SPICES 2006]. The Correctness, Modeling, and Performance of Aerospace SystemS (COMPASS) project focused on a system and software co-engineering approach through a coherent set of specification and analysis techniques to evaluate correctness, safety, dependability, and performability in aerospace systems [COMPASS 2011]. The System Architecture Virtual Integration (SAVI) initiative led by the international aircraft industry consortium called Aerospace Vehicle Systems Institute (AVSI) is maturing and putting into practice an architecture-centric, model-based engineering approach. Using a single-truth model repository/bus based on AADL, this approach uncovers problems in the system and embedded software system early in the life cycle to address exponential development and qualification cost growth [Feiler 2009a]. In particular, the SAVI initiative provides an opportunity of leveraged cooperation [Redman 2010].

Applied throughout the life cycle, reliability validation and improvement leads to an end-to-end Virtual Upgrade Validation (VUV) approach [DeNiz 2012]. This approach builds the argument and evidence for sufficient confidence in the system throughout the life cycle, concurrent with development. The framework keeps engineering efforts focused on high-risk areas of the system architecture and does so in a cost-saving manner through early discovery of system-level problems and resulting rework avoidance [Feiler 2010]. In support of qualification, the assurance evidence is collected throughout the development life cycle in the form of formal analysis of the architecture and design combined with testing the implementation.

The architecture-centric framework provides a basis for a reliability validation and improvement program of software-reliant systems [Goodenough 2010]. Building software-reliant systems through an architecture-centric, model-based analysis of requirements and designs allows the discovery of system-level errors earlier in the life cycle than system integration time, when the majority of such errors are currently detected.

The framework also provides the basis for a set of metrics that can drive cost-effective reliability validation and improvement. These metrics address shortcomings in statistical fault density and reliability growth metrics when applied to software. They are architecture-centric metrics that focus on a major source of system-level faults: namely requirements, system hazards, and architectural system interactions. They are complemented by a qualification-evidence metric that (1) is based on assurance case structures, (2) leverages the DO-178B model of qualification criteria of different stringency for different criticality levels, and (3) takes into account the effectiveness of various evidence-producing validation methods [FAA 2009a].

[3] SAE international was formerly known as the Society of Automotive Engineers.

The effects of acting on this early discovery are reduced error leakage rates to later development phases (e.g., residual defect prediction through the COnstructive QUALity MOdel COQUALMO [Madachy 2010]) and major system cost savings through rework and retest avoidance (e.g., Feiler return-on-investment study[4]). We can leverage these cost models to guide the cost-effective application of appropriate validation methods.

4 Peter Feiler, Jorgen Hansson, Steven Helton. *ROI Analysis of the System Architecture Virtual Integration Initiative*. Software Engineering Institute, Carnegie Mellon University. To be published.

Abstract

Software-reliant systems such as rotorcraft and other aircraft have experienced exponential growth in software size and complexity. The current software engineering practice of "build then test" has made them unaffordable to build and qualify. This report discusses the challenges of qualifying such systems, presenting the findings of several government and industry studies. It identifies several root cause areas and proposes a framework for reliability validation and improvement that integrates several recommended technology solutions: validation of formalized requirements; an architecture-centric, model-based engineering approach that uncovers system-level problems early through analysis; use of static analysis for validating system behavior and other system properties; and managed confidence in qualification through system assurance. This framework also provides the basis for a set of metrics for cost-effective reliability improvement that overcome the challenges of existing software complexity, reliability, and cost metrics.

1 Introduction

Rotorcraft and other military and commercial aircraft rely increasingly on complex and highly integrated hardware and software systems for safe and successful mission operation. Traditionally, avionics systems consisted of a federated set of dedicated analog hardware boxes, each providing different functionality, and exchange of physical signals. Over time avionics systems evolved to using digital implementation of the functions through periodic processing by embedded software and exchange of digital signals through a predictable periodic communication medium such as MIL-STD 1553B. The next step included the (1) migration to an Integrated Modular Avionics (IMA) architecture, in which the embedded software is sometimes modularized into partitions with interactions limited to port-based communication and the (2) deployment of the software on a common distributed computer platform. In this evolution, the role of embedded software has grown from providing the functionality of individual system components to integrating, coordinating, and managing system-level capabilities to meet mission and safety-criticality requirements.[5]

1.1 Reliability Assessment

In keeping with this growing complexity, the qualification and reliability assessment of these systems has become increasingly challenging within budget and schedule [Boydston 2009]. Current practice relies on process standards, best practices, and a safety culture.[6] The phases of a traditional software development process are typically shown as a software development V chart (see Figure 1). The downward portion of the V puts an emphasis on development complemented by design and code reviews; the upward portion focuses on testing complemented by managing build

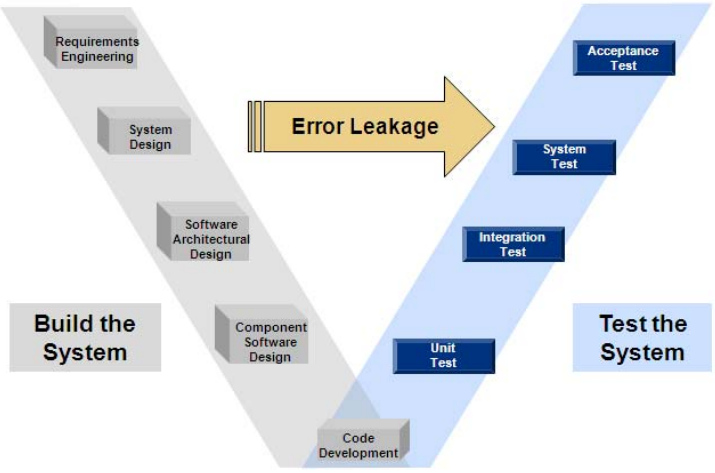

Figure 1: Traditional Phases of Software Development

[5] See Section 3.1 for a definition.

[6] Some of the standards and practices are the Capability Maturity Model Integration® (CMMI®), MIL-STD-882, SAE ARP4754, SAE ARP4761, DO-178B, DO-254, UK 00-56, IEC/ISO 15026, IEC 61508, and ARINC653. (®Capability Maturity Model Integration and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.)

and deployment configurations. The process has evolved from a waterfall model of strict sequencing of the phases to spiral development, in which developers iterate through the phases in order to refine design.

The "build then test" approach illustrated in the V chart is becoming unaffordable for aircraft and rotorcraft development. For example, software costs for the latest generation commercial aircraft are reaching $10 billion [Feiler 2009a], with software making up two thirds of the total system cost. Furthermore, the separation of system and software engineering in the traditional process has led to shortcomings in the delivery of system nonfunctional requirements [Boehm 2006]. Embedded software in aircraft is increasingly responsible for the safety and reliability of aircraft system operation [Leveson 2004a, GAO 2008]. This software is also experiencing exponential growth in size and complexity [Feiler 2009a, Dvorak 2009], making it a challenge to qualify and certify.

Reliability engineering, as practiced, has its roots in the use of statistical techniques to assess the hardware reliability of a slowly evolving system design and an operational system affected by wear and aging over time. Software reliability differs from hardware reliability in that it is primarily driven by design defects. Software evolves quite rapidly and corrections are effectively design changes. As a result its failure distribution curve does not adhere to the bathtub curve[7] common for physical systems.

Often the reliability of the software is assumed to be perfect and to behave deterministically (i.e., to produce the same result given the same input) [Goodenough 2010]. Therefore, the focus in software development has been on testing to discover and remove bugs using various test coverage metrics to determine test sufficiency. Failure-probability density function based on code metrics, such as source lines of code (SLOC) and cyclomatic (conditional) complexity have been used as predictors with limited success [Kaner 2004]. Neither is a good measure of system-level interaction complexity and nonfunctional properties such as performance, reliability, or safety. This is due to the fact that time-sensitive software component interactions may encounter race conditions, unexpected latency jitter, and unanticipated resource contention, which occur non-deterministically. We not only need better reliability metrics, but also a change in the way we engineer and qualify software-reliant systems. Steps in that direction include the use of the Architecture Tradeoff Analysis Method® (ATAM®) developed at the Carnegie Mellon® Software Engineering Institute (SEI) [Kazman 2000]. Applying the ATAM helps to identify architectural risks in early design phases. The use of architecture models with well-defined semantics, such as the Society of Automotive Engineers (SAE) Architecture Analysis and Design Language (AADL), helps to drive early detection of errors through system-level analysis.

The U.S. Army Materiel Systems Analysis Activity (AMSAA) *Reliability Growth Guide* defines reliability growth as "the improvement in a reliability parameter over a period of time due to changes in the product design or the manufacturing process [AMSAA 2000]. It occurs by surfacing failure modes and implementing effective corrective actions." The AMSAA provides funding for hardware reliability improvement programs that use modeling, analysis, and simulation to

---

[7] See footnote 1 for an explanation of the bathtub curve.

® Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

identify and reduce design defects before the system is built, while software funding is focused on finding and removing code faults through code inspection and testing. There is a clear need to

- extend reliability improvement programs to software designs
- include software failure modes in the system design
- design for robustness to address unplanned usage and hazard conditions [Goodenough 2010].

In addition, it is necessary to define metrics for quantifying reliability improvement of software-reliant systems and developing justified confidence in the system behavior.

Several studies identify technologies that are key to addressing these challenges in software-reliant systems and the need to integrate these technologies into a system and software co-engineering practice [Dvorak 2009, Conquet 2008, Redman 2010, Boehm 2006]. The identified technologies are listed below:

- model-based engineering driven by architecture models with well-defined semantics
- improved specification of mission and safety-criticality requirements with focus on system interaction with its operational context and between subsystems
- application of static analysis based on formal methods to complement testing in the end-to-end verification and validation (V&V) of systems
- assurance cases to systematically provide evidence for justified confidence that a system meets its intent and requirements.

Several initiatives have been under way in industrial settings to demonstrate the maturation and integration of these technologies including

- the development of the international SAE AADL standard [SAE 2004-2012] based on research on Architecture Description Languages (ADLs), funded by the Defense Advanced Research Projects Agency (DARPA) with strong industrial participation
- the application of the AADL standard for embedded systems to drive the development and verification of two satellite system families. The European Space Agency (ESA) led this initiative—the Automated proof-based System and Software Engineering for Real-Time applications (ASSERT) initiative [Conquet 2008].
- the use of virtual system and software integration that reduces integration errors and that centers on a single-source-of-truth[8] architectural reference model based on the SAE AADL. Such errors decline through discovery of system-level problems through model-based analysis of mission and safety-criticality properties throughout the development. This work has been undertaken by an international aircraft industry initiative called System Architecture Virtual Integration (SAVI) [Redman 2010, Feiler 2010].
- a system-theoretic approach to safety engineering [Leveson 2005] that builds on early work in formalized requirement specification by Parnas [Parnas 1991]
- the systematic application of model checking to formalized requirement specifications and system and software designs, as well as to code [Tribble 2002, Miller 2010, Gurfinkel 2008]
- the generalization of safety cases, which are part of UK Defense Std 00-56: *Safety Management Requirements for Defense Systems*, into assurance cases [Goodenough 2009].

[8] In a single source of truth model structure, every data element is stored exactly once.

Other initiatives include

- the European Support for Predictable Integration of mission Critical Embedded Systems (SPICES). This initiative is integrating model-based engineering, using AADL with the Common Object Request Broker Architecture (CORBA) Component Model (CCM) and auto generation of SystemC code [SPICES 2006]
- Toolkit in OPen-source for Critical Applications and SystEms Development (TOPCASED), an Eclipse-based open source environment and embedded system development platform that supports model-based engineering through multiple notations via the model bus concept [Heitz 2008]
- Correctness, Modeling, and Performance of Aerospace SystemS (COMPASS), focusing on a system and software co-engineering approach through a coherent set of specification and analysis techniques to evaluate correctness, safety, dependability, and performability in aerospace systems [COMPASS 2011]
- the DARPA META program aiming to achieve dramatic improvement of the systems engineering, integration, and testing process through architecture-centric, model-based design abstractions that lead to quantifiable verification and optimization of system design [DARPA 2010].

To accommodate these technology advances, process and practice standards are being revised to foster better system and software co-engineering, including the

- recent alignment of process standards for the systems life cycle (ISO/IEC 15288) and for the software life cycle (ISO/IEC 12207)[ISO/IEC 2008a, 2008b]
- revision of recommended practice for architectural description of software-intensive systems (IEEE 1471) [IEEE 2000]
- revision of software considerations in airborne systems and equipment certification (DO-178 revision C) incorporating tool qualification, model-based design and verification, use of formal methods, and application of object-oriented technology [RTCA 1992]
- embracing of Model-Based System Engineering (MBSE) by the International Council on Systems Engineering (INCOSE) through a set of grand challenges [INCOSE 2010]
- development by the Object Management Group (OMG) of Unified Modeling Language (UML) profiles such as Systems Modeling Language (SysML) for system engineering [SysML.org 2010].

1.2 Definition of Key Terms

Before we proceed with the report, we define some terms in the context of this report.

We use the term *software-reliant systems* (SRSs) to identify systems whose mission and safety-criticality requirements are met by an integrated set of embedded software and by their interaction with their operators and other systems in their operational environment. Such systems are also referred to as

- *software-intensive systems* (SISs), indicating the need for system and software co-engineering [Boehm 2006]

- *distributed real-time embedded* (DRE) *systems* to indicate that they represent an integrated set of embedded software
- *cyber-physical systems* (CPSs) to indicate that the embedded software interacts with, manages, and controls a physical system [Lee 2008].

System reliability is defined as the ability of a system to perform and maintain its required functions under nominal and anomalous conditions for a specified period of time in a given environment. This definition is adapted from a definition by the National Computer Security Center [NCSC 1988]. System reliability is typically expressed by a failure-probability density function over time.

Airworthiness qualification is defined as the demonstration of an aircraft or aircraft subsystem or component, including modifications, to function safely, meeting performance specifications when used and maintained within prescribed limits [U.S. Army 2007].

System assurance is defined as justified confidence that the system functions as intended and is free of exploitable vulnerabilities, either intentionally or unintentionally designed or inserted as part of the system at any time during the life cycle [NDIA 2008].

1.3 Purpose and Structure of This Report

The U.S. Army Aviation and Missile Research Development and Engineering Center (AMRDEC) Aviation Engineering Directorate (AED) funded the SEI to develop a reliability validation and improvement framework. The purpose is to address the challenges of qualifying increasingly software-reliant safety-critical systems by overcoming limitations of current reliability engineering approaches. Achieving these goals requires leveraging best emerging engineering technologies and practices to complement the process focus of current practice, finding acceptance in industry, and leading an effort to a new set of reliability improvement metrics.

In this report, we

- summarize the findings of the background research in terms of key challenges in the qualification of safety-critical, software-reliant systems
- discuss an engineering framework for reliability validation and improvement that integrates several engineering technologies
- outline a new set of metrics that focus on cost-effective reliability improvement.

We close the report by describing opportunities to leverage ongoing industry and standards efforts and potential follow-on activities specific to the U.S. Army that aim to accelerate adoption of these proposed improvements in engineering and qualification practice.

2 Challenges of Software-Reliant Safety-Critical Systems

In this section, we take a closer look at the challenges arising from increased reliance on embedded software and potential root causes. We do so by examining experiential data and by identifying high-risk areas that can benefit from application of more effective engineering and qualification technology.

2.1 Exponential Growth in Size and Interaction Complexity

For the international aerospace industry, the cost of system and software development and integration has become a major concern. Aerospace software systems have experienced exponential growth in size and complexity—and, also unfortunately, in errors, rework, and cost. Development of safe aircraft is reaching the limit of affordability. Figure 2 shows that the size of on-board software for commercial aircraft (measured in SLOC) doubled every four years since the mid-1990s and reached 27 million SLOC by 2010. Using the COnstructive COst MOdel (COCOMO) II and assuming 70% reuse of software, an estimated cost to develop such software is as much as $10 billion, a sum that would make up more than 65% of the total aircraft development cost.

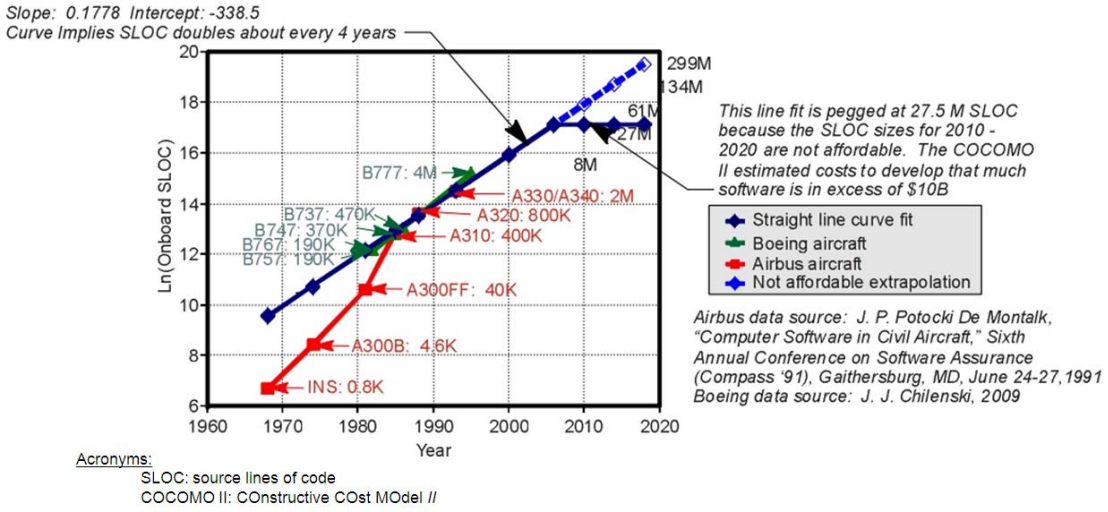

Figure 2: Estimated On-Board SLOC Growth

Figures for military aircraft and rotorcraft are experiencing similar growth. This growth in size is due to reliance on software for (1) providing flight-critical capability, such as fly-by-wire; (2) mission capability through provision of up-to-date situational awareness and command and control; and (3) fault and hazard management to maintain safe and reliable system operation. The resulting embedded software systems have increased interaction complexity between embedded software subsystems and increased potential for conflicting demands of shared computer platform resources. At the same time, digitalization and implementation as an IMA architecture has resulted in weight reduction.

2.2 Error Leakage Rates in Software-Reliant Systems

A number of studies have been performed on where errors are introduced in the development life cycle, when they are discovered, and the cost of the resulting rework. We limit ourselves here to work by the National Institute of Standards and Technology (NIST,) Galin, Boehm, and Dabney [NIST 2002, Galin 2004, Boehm 1981, Dabney 2003]. The NIST data primarily pertains to information technology applications, while the other studies draw on safety-critical systems.

Figure 3 shows a summary of three data points across development phases: percentage of error introduced in a particular phase; percentage of errors discovered in a particular phase; and a rework cost factor normalized with respect to the cost of repair in the requirements phase. The rework cost figures include the cost of retest.

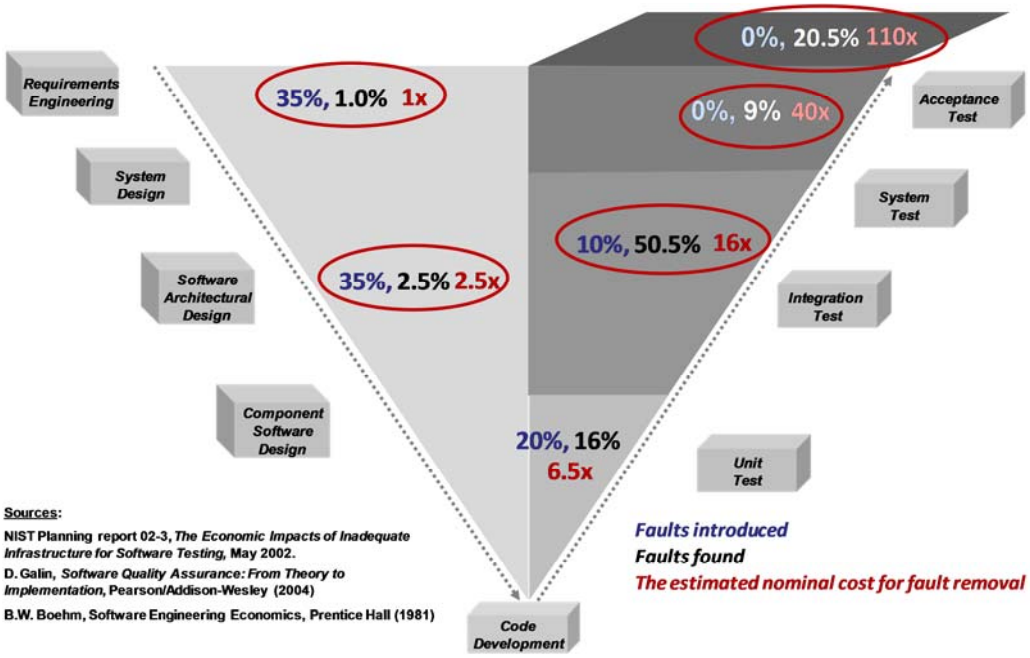

Figure 3: Error Leakage Rates Across Development Phases

The percentages of errors introduced and discovered were quite consistent across the three studies. The figure shows that 70% of all errors are introduced during requirements engineering, including system design (35%), and software architectural design (35%). In comparison, only 20% of the errors are discovered by the end of code development and unit test, while 80% of the errors are discovered at or after integration testing. The figure shows that 20% of the errors are introduced during code development and unit testing and 16% of the errors are discovered in that phase.

Overall, the data shows that we need to do a better job of getting the requirements specified and of managing the interaction complexity between system components not only in terms of system functionality but also in terms of nonfunctional system properties. Note that the studies provide more detailed data in terms of leakage rates between phases than we present here.

Figure 3 shows rework cost factors based on the Galin and Boehm studies [Galin 2004, Boehm 1981]. Table 1 provides more details regarding error rework cost factors from all four of the stud-

ies. The numbers in Table 1 support the estimated nominal cost for fault removal shown in Figure 3. Given those cost factor numbers, the rework cost for requirements errors alone makes up 78% of the total rework cost. In other words, there is high leverage in cost reduction through a focus on early discovery of requirements- and system-design-related errors. In the next section, we take a closer look at requirements-related error data to gain insight into how to improve the situation.[9]

Table 1: Error Rework Cost Factors Relative to Phase of Origin

Phase	Relative Defect Removal Cost of Each Phase of Origin														
	Requirements			Design			Coding			Unit test			Integration		
	[NIST 2002]	[Galin, Boehm 1981]	[Dabney 2003]	[NIST 2002]	[Galin 2004, Boehm 1981]	[Dabney 2003]	[NIST 2002]	[Galin 2004, Boehm 1981]	[Dabney 2003]	[NIST 2002]	[Galin 2004, Boehm 1981]	[Dabney 2003]	[NIST 2002]	[Galin 2004, Boehm 1981]	[Dabney 2003]
Requirements	1	1	1												
Design	1	2.5	5	1	1	1									
Unit coding	5	6.5	10	5	2.5	2	1	*	1		1				
Testing	10	*	50	10	*	10	10	*	5		*	1			
Integration	10	16	130	10	6.4	26	10	*	13	1	2.5	3	*	1	1
System/ Acceptance test	15	40	*	15	16	*	20	*		10	6.2	*	*	2.5	*
Operation	30	110	368	30	44	64	30	*	37	20	17	7	*	6.9	3

2.3 Requirements Errors

Hayes used a requirement fault taxonomy from a U.S. Nuclear Regulatory Commission guideline (NUREG/CR-6316) to examine some National Aeronautics and Space Administration (NASA) data on system errors [Hayes 2003, Groundwater 1995]. The data shows that the top six requirement-related error categories are

1. omitted/missing requirements (33%)
2. incorrect requirements (24%)
3. incomplete requirements (21%)
4. ambiguous requirements (6.3%)
5. overspecified requirements (6.1%)
6. inconsistent requirements (4.7%).

A requirements engineering study for the Federal Aviation Administration (FAA) published in 2009 [FAA 2009a] included an industry survey of requirements engineering practices that is in compliance with Radio Technical Commission for Aeronautics standard DO-178B [RTCA 1992]. The survey includes data on the notations and tools used in capturing requirements.

Figure 4 shows the notations used by different organizations. English text and structured Shall . . . statements together with tables and diagrams are the dominant notations. Note that tables include

[9] An asterisk in the table means that the study cited in that column did not have data on this category (i.e., did not distinguish it).

representation of state information, such as use of truth tables. The next set of notations shows the use of executable models such as MATLAB/Simulink and the use of data flow diagrams (i.e., notations that can be analyzed by tools).

Figure 4 also shows reported tool usage for requirements capture. The tools are dominated by word processing tools such as Microsoft Word and by Dynamic Object Oriented Requirements System (DOORS), which provides good support for requirements traceability. Databases are used as an alternate way of maintaining requirements traceability. Spreadsheets are effective in maintaining tabular representations, especially if they include some computation. Simulink as a tool supports representation of executable Simulink models.

Enter an "x" in every row/column cell that applies	System Requirements	Data Interconnect (ICD)	High-Level Software Requirements	Low-Level Software Requirements	Hardware Requirements
English Text or Shall Statements	39	27	36	32	29
Tables and Diagrams	31	30	30	19	18
UML Use Cases	1		2	4	
UML Sequence Diagrams			3	6	
UML State Diagrams			1	7	
Executable Models (e.g. Simulink, SCADE Suite, etc.)	7	1	8	8	1
Data Flow Diagrams (e.g. Yourdon)	4		6	9	
Other (Specify)-Proprietary Database, DOORS objects	1	4	2	2	1
Other (Specify)XML		1			
Operational models or prototypes	1	1			1
UML			1	1	

Enter an "x" in every row/column cell that applies	System Requirements	Data Interconnect (ICD)	High-Level Software Requirements	Low-Level Software Requirements	Hardware Requirements
Database (e.g., Microsoft Access)	3	4	3	3	
DOORS	23	13	22	18	12
Rational ROSE®			1	3	
RDD-100®					
Requisite Pro®	5	3	5	4	4
Rhapsody	1				
SCADE Suite	2		3	1	
Simulink	5	1	5	3	1
Slate	1		1	1	
Spreadsheet (e.g., Microsoft Excel)	5	4	5	4	3
Statemate					
Word Processor (e.g., Microsoft Word)	19	20	18	17	16
VAPS™		1	3	3	
Designer's Workbench™			1	1	

Figure 4: Notations and Tools Used in DO-178B-Compliant Requirements Capture [RTCA 1992]

A study by Groundwater and colleagues investigates the effectiveness of different techniques for finding errors in a single-mode transition diagram and in two interacting-mode transition diagrams [Groundwater 1995]. The results are shown in Figure 5. Since expected mode behavior in the form of mode transition is commonly part of requirement specification, it is clear that errors in such requirement specifications can easily propagate into the design and implementation phases. Therefore, it is desirable to validate such specifications during requirements capture.

Errors Found by Analysis of a Single Mode Transition Diagram

Dectected By	Likelihood of Being Found by Traditional Methods				
	Trivial	Llikely	Possible	Unlikely	Total
Inspection			1	2	3
Modeling		5	1		6
Simulation					
Model Checking	2	1	6		9
Total	2	6	8	2	18

Errors Found by Analysis of Interacting Mode Transition

Dectected By	Likelihood of Being Found by Traditional Methods				
	Trivial	Likely	Possible	Unlikely	Total
Inspection					
Modeling					
Simulation					
Model Checking			7	1	8
Total			7	1	8

Figure 5: Effectiveness of Different Error Detection Techniques

2.4 Mismatched Assumptions in System Interaction

System engineers are concerned about the interface between the system and its operator, as well as the interaction of the system with its operational environment. In many cases, the system consists of a system under control and a control system that monitors, controls, and manages its operation. In the process of specifying the requirements for the system, engineers make assumptions about how operators interact with the system and about the operational environment. Similarly, they make assumptions about the physical system under control when they specify the requirements for the control system. These assumptions may not always be valid and may be violated over time due to changes in the operational context or in the system itself. The following examples illustrate the point:

- It is common for physical systems to make assumptions about certain conditions of the operational environment, such as the temperature range, in which the system should be operated.
- Operators are expected to have situational awareness of the operational environment with sometimes limited or misleading information or guidance. The result can be an accident, as was the case in the ComAir crash when one of the taxiways was under construction [Nelson 2008].
- An example of mismatched assumptions made about the interaction between the operator and the system is an incident in which a subway train left the platform without the operator present at the operator console. One of the doors on the first car had difficulty closing. The operator stepped out of the operator cabin to close the door and the semi-automated system—sensing that all doors were closed—departed with the operator standing on the platform.

Similarly, assumptions exist when a control system is designed for a system under control (see Figure 6). For example, engineers make assumptions about the lift generated by aircraft wings in determining the maximum load and in identifying the operational envelope. The interaction complexity among several system parameters can lead to violation of assumptions about system parameters and result in incidents or accidents. This circumstance caused Air France flight 447 to crash en route from Brazil to France [Spiegel 2010]. Flight 447 was loaded to within 240kg of maximum capacity, and its estimated fuel consumption was based on a majority of the flight occurring at 39,000 feet. At that altitude, the operational speed for maintaining the required lift is quite narrow, which increases the risk of operating outside a safe flight envelope in non-nominal

situations such as severe turbulences due to storms. This example illustrates the challenge of understanding all system hazards and specifying appropriate safety and reliability requirements.

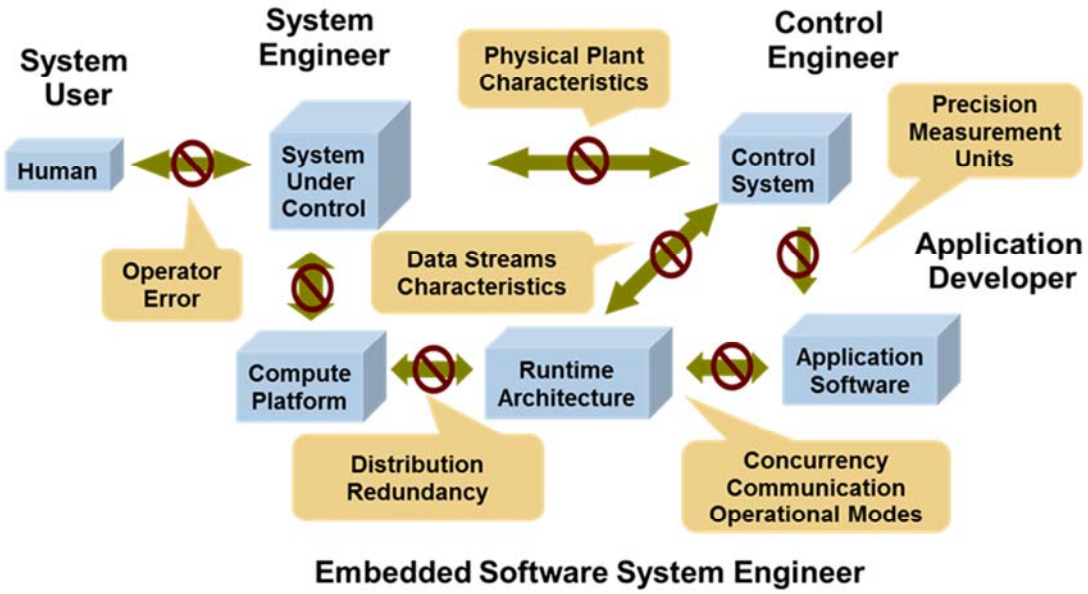

Figure 6: Interaction Complexity and Mismatched Assumptions with Embedded Software

The upper half of Figure 6 illustrates some of this interaction complexity and the potential for mismatched assumptions. As these control and under-control systems have become software reliant, new areas of interaction complexity and potential for mismatched assumptions are introduced as shown on the lower half of Figure 6.

As system functionality is implemented in software, variables in the environment are translated into input and output variables on which the embedded software operates. System engineers may assume the data in the variables to be expressed in a particular measurement unit, which may not have been communicated to the software engineer when system requirements were translated into software requirements. Similarly, the expected range of values and the degree of precision in which they are represented is affected by the base type chosen for the variable. For example, one of the contributing factors to the Ariane 5 accident was the use of a 16-bit integer variable, which could handle the range of values for Ariane 4, but resulted in negative values due to wraparound [Nuseibeh 1997].

Application software is integrated into a **runtime architecture** that supports multiple operational modes, with different modes involving different subsets of active tasks and communication channels. In the way tasks interact, race conditions and a nondeterministic sequence of actions can result, due to the application software making assumptions about the runtime environment. For example, the application software may assume that two tasks may not require explicit synchronization because execution of both tasks on the same processor using a non-preemptive scheduling protocol ensures mutual exclusion. These assumptions may be violated in migration to the use of a multi-core processor or other different runtime systems and computer hardware. Analysis of formalized system models allows us to (1) discover these problems early in the technology refresh

cycle; and (2) increase our confidence, having addressed intricate time-sensitive logic errors that are difficult to test for.

The **computer platform** is typically a distributed networked set of processors with redundancy to provide reliable system operation. This means that replicated instances of the embedded software execute on different processor instances and communicate over different network instances. A change in the deployment configuration of the embedded software may lead to replicated software components' allocation to the same physical processor and to the elimination of physical redundancy. Similarly, migration of embedded software to a partitioned runtime architecture such as ARINC653[10] can result in reduced reliability, if the mapping of embedded software to partitions and the binding of partitions to physical hardware are not performed consistently with the redundancy requirements for the system. Finally, virtualization can lead to unplanned resource contention and performance that is slower than expected [Nam 2009].

Embedded applications may process time-sensitive data and process data in a time-sensitive manner. For example, a control system makes assumptions about the latency of a data stream from a sensor to an actuator. Different control algorithms have different thresholds of sensitivity to sampling jitter, which, if exceeded, can result in unstable control behavior [Feiler 2008]. Differences in the task execution and communication timing of different runtime architectures and their particular hardware deployments can affect latency, as well as sampling and latency jitter. For example, latent delivery of data such as helicopter main rotor speed (Nr) in an autorotation can lead to a catastrophic result.

Similarly, it is common practice to implement event processing by periodically sampling a data variable whose change in value signals that an event has occurred and that reverts to its original value after a given duration. The application logic assumes that all events are communicated to the recipient. However, when the variation of the sampling exceeds a certain threshold, the recipient fails to observe an event. Such an unanticipated loss of events can result in inconsistent system states and deadlock in system interaction protocols.

In a study of embedded software systems with system-level problems that escape traditional fault tolerance mechanisms, the SEI has identified four root cause areas that require attention [Feiler 2009b]:

1. **processing of data streams in a time-sensitive manner**. Data streams are often processed in multiple steps. Different components involved in processing this data stream make assumptions about

- the data of a data stream: for example, the application data type (e.g., temperature), its base type representation (e.g., 16-bit unsigned integer), acceptable range of values, base value that is represented as zero (e.g., -50), and measurement unit (e.g., degree Celsius)
- the timing of the data stream: age of the data (i.e., time since it was read by a sensor), data latency (i.e., handling time of new data), and latency jitter (i.e., variation in latency)
- the data stream characteristics: for example, acceptable transmission rates, acceptable rates of missing stream elements, out of sequence data, dropped data, corrupted data, data encryption/decryption, and acceptable limits in value changes between elements of the data stream

[10] ARINC653 is a specification for system partitioning and scheduling that is often required in safety- and mission-critical systems.

- synchronization of data in voting or self-checking pair system, time stamping, and time distribution throughout a system

2. **interaction between state-based systems with replicated, mirrored, and coordinated state machines**. Examples are replicated discrete state applications, multiple distributed instances of redundancy management logic, and handshaking protocols or coordinated operational modes. The state transition logic embedded in the state machine may make assumptions about
    a. the interaction of replicated and mirrored state machines by working with the same inputs exclusively, by comparing states periodically, or by observing each other's state behavior in order to detect anomalous behavior. The state logic may not accommodate failures in the application logic, in the underlying hardware, or in timing differences due to an asynchronous computer platform.
    b. the communication of state versus state change (e.g., exchange of track information and track updates). Communication of state change information assumes guaranteed and often-ordered delivery of information by the communication protocols and hardware.
    c. the communication of events by sampling state variables. The particular implementation, while maintaining a periodic task set, may not guarantee observation of every event or queuing of events if arrival burst exceeds the processing rate due to the mismatch in paradigms of guaranteed event processing and data sampling.
3. **performance impact of resource management**. Such impact is especially important when sharing computer resources in IMA architectures and can lead to a
    a. mismatch of resource demand and capacity, where the demand may exceed the capacity or capacity of one resource may exceed capacity of connected resource. For example, a high-bandwidth gigabit ethernet network can flood low-performance processors resulting in denial of service and lower than expected processor speed.
    b. lack of guaranteed resource capacity assumed to be available to the embedded application due to undocumented resource sharing and unmanaged resource usage. For example, direct memory access (DMA) transfers that continue independent of the application software initiating them consume bus and memory bandwidth assumed to be available to other application software.
    c. mismatch in assumptions by the application in the execution and communication timing and ordering, and in the scheduling of the processor and network resources by the underlying runtime system
4. **virtualization of resources**. The virtualization of resources such as processors and networks brings flexibility to a system design and provides resource budget enforcement. However, it can lead to
    a. loss of reliability due to differences in logical redundancy and physical redundancy—if logical resources are mapped to physical resources in conflict with application and safety-criticality assumptions
    b. limitations in resource isolation guarantees in terms of both guaranteed resource capacity available to a virtual resource (e.g., virtual channels competing for resources) and information leakage between applications due to resource sharing

c. time inconsistency due to the application software's operating in virtual time. For example, multitasking and execution of embedded applications in partitions do not guarantee input sampling at known time intervals when such input sampling is performed as part of the application code. Similarly, time stamping of time-sensitive data can lead to inconsistency in a multi-clock distributed platform.

d. mixed-criticality systems in which applications with periodic and event-driven resource demands and different security levels, safety levels, and redundancy requirements must use shared resources consistently despite conflicting demands

2.5 Software Hazards and Safety-Criticality Requirements

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

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

Experience has shown that such safety analysis must take into account interactions with operators and the operational environment, as well as the development environment as sources of contributing and systemic hazards [Leveson 2004a, 2005]. Controlling these sources of hazard involves

- translating the results of such safety hazard analysis into safety requirements on the system
- validating these requirements for completeness and consistency, and for ensuring that they are satisfied at design time or managed by fault tolerance mechanisms in the system when violated.

Such translation and validation has led to the formalization of requirements, often expressed as discrete state behavior and boundary conditions on physical system and environmental state [Parnas 1991, Leveson 2000, Tribble 2002] and their V&V through formal methods [Groundwater 1995]. It has also led to an understanding that system safety and reliability are emergent system properties. System reliability can be achieved with unreliable components, and reliable system components do not guarantee system reliability or system safety [Leveson 2009].

Understanding how to quantify software's contribution to system reliability and safety has been a challenge. Initially it may seem that software reliability cannot be addressed in terms of MTBF because software errors either exist or do not exist (i.e., a software function will always produce the same result when executed under exactly the same conditions). However, such conditions include not only the function's explicit inputs, but also its environment, which may reflect its execution history and other activities that are happening concurrently with the function's execution. For

example, software interacting with other systems or with humans may be sensitive to which functions and actions have been taken prior to the function's invocation. Likewise, its behavior might be dependent on timing, resource contention, or other functions that are executing concurrently with it. In such cases, only a particular execution order and environmental state may cause an error. Even though a given execution order and environmental state will produce the same erroneous result every time, the likelihood that the error-inducing fault activations will occur depends on the operational use circumstances of the system.

Errors in software designs and implementations present both reliability and safety hazards and must be treated accordingly; that is, they must be eliminated, reduced in likelihood, or mitigated. We need to understand the hazards introduced by possible errors in software and their impact on system reliability and safety. This requires an understanding of the role of software in system reliability and safety. Software is not just a source of failure; it is also responsible for managing fault tolerance. Embedded software may contribute to the desired mission capability, to reliability (by implementing fault management of physical system components, of the computer platform, and of the mission software), and to safety (by monitoring for violation of safety requirements).

Clearly, reducing software errors is a way of improving software reliability (i.e., it is a way of reducing the likelihood that software will fail under certain conditions). But there are two important classes of software errors—architectural errors (introduced in the design phase) and implementation errors (introduced in the implementation phase). The distinction between the types of errors pertains not only to the phase at which they are introduced: the distinction lies in the characteristic of the errors. In particular, what we call "architectural" errors have to do with the explicit (and implicit) interactions between system components. (Implicit interactions can occur through contention for shared resources, timing properties, dependence on shared state, etc.) In practice, the architectural errors (sometimes called *design* errors) are the ones most often implicated in actual accidents [Leveson 2004b]. Methods and processes intended to improve system safety, reliability, and security must focus on detecting (and eliminating) architectural errors. Because architectural errors can never be completely eliminated in complex systems, such methods must also ensure that the failure effects of errors are adequately mitigated—in addition to validating the source code of the implementation.

2.6 Operator Errors and Work-Arounds

Historical data [Couch 2010] and numerous investigations of rotorcraft accidents identify the operator/pilot as the root cause in 80% of the cases. This finding is often motivated by the need to look for blame [Leveson 1995]. However, in order to improve the safety record of such systems, we must consider other contributing factors such as design errors or complex operational procedures, and we must include the operator in the system analysis.

When problems are discovered, a solution may be identified but not installed in fielded systems immediately. In the case of software problems, the corrections are design changes, which may have unintentional side effects. Furthermore, safety-critical systems require recertification, which with the current practice, is quite expensive and impractical for incremental corrections. Instead, work-arounds are added to operational procedures and operators may spend a majority of their time performing them. In other words, we have passed the responsibility to compensate for system problems that could be corrected to the system operator.

This indicates a clear need for

- including operator behavior specifications
- identifying inconsistencies and complexities in the operator's interaction with the system, including situations where the operator does not follow procedures (intentionally or unintentionally)
- improving the reliability of software-reliant systems by reducing error leakage
- improving the cost effectiveness of system qualification.

2.7 Errors in Fault Management Systems

Safety-critical systems contain mission software and fault management software. Fault management software may make up 50% or more of the total system, and errors in the fault management logic make up a considerable percentage of all system errors. The system portion responsible for reliable operation is, itself, unreliable. One reason for this is a limited understanding of software faults and hazards, due to assumptions made about the operational environment, system components, and system interactions. A second reason is the interaction complexity in systems, particularly of the embedded software. A third reason is the challenge of testing the fault management portion of a system. Fault management software is only executed when the system fails, and fault management errors are only triggered when the system is already dealing with an erroneous condition. Fault injection has been used to exercise fault management, but has struggled to address concurrency and timing-related faults, as well as testing under all expected operational contexts.

Since fault management is a critical component of the system for achieving reliable and safe operation, it is important to improve its quality. We can achieve this by formally specifying and analyzing reliability and safety requirements to address identified hazards and assumptions, decomposing these requirements along the system architecture, and validating the system architecture and its implementation (including fault management) against these requirements.

2.8 Reliability Improvement and Degradation Over Time

Current practice in the reliability improvement of software is focused primarily on finding and removing bugs through review and testing. Testing has focused on exercising the code with various inputs to ensure that all code statements execute as expected and produce the expected results. Many of the test coverage approaches reflect the assumption that software behaves deterministically (i.e., for given inputs, the software executes the same statements and produces the same results). Black-box testing focuses on mapping sets of input data to expected output data. White-box testing focuses on exercising the program logic reflected in the source code statements. Since there are many potential interactions between source code statements, programming language[11] abstractions have been introduced to manage this complexity through concepts such as

- data abstraction and object orientation
- strong typing
- modularity with well-defined interfaces
- restrictions such as static memory allocation

[11] Ada is an excellent example of a programming language for reliable software.

- the absence of application-level manipulation of pointers (as found in high-integrity subset profiles of programming languages, such as the Ada Ravenscar profile [Ada WG 2001]).

Despite these capabilities, the challenge is to find white-box, black-box and system test coverage approaches that bound the amount of necessary testing.

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

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

A complicating factor is that embedded software executes as an interacting set of concurrent tasks that operate on time-sensitive data and events. Such software may encounter aspects that appear to occur randomly and are difficult to test systematically, such as for race conditions, unexpected latency jitter, and unanticipated resource contention. Given the exponential growth in software size and interaction complexity, the concept of exhaustive testing has turned into testing until the budget or schedule has been exhausted [Boydston 2009].

Leveson observes that systems will tend to migrate toward states of higher risk with respect to safe operation [Leveson 2009]. There are 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.

One reason for the trend toward higher risk is that the operational environment impacts the safety of systems. For example, the system may be deployed in new unintended operational environments that may introduce new hazards and may violate assumptions made about the operational environment when it was designed—resulting in unexpected behavior. Similarly, changes to operational procedures and guidelines, whether as the result of operational budget reductions or as a work-around to compensate for known and correctable system faults (see Section 2.6) may also contribute to increased risk.

A second reason for this trend is that modifications to software are design changes, whether they are the addition of new functionality or corrections to existing code. Compared to hardware, software experiences rapid design evolution. In particular, the addition of new mission capability and operational features is a common occurrence, since software can be easily changed. Similarly, technology upgrades to the computer system impact the embedded application software. Examples range from the introduction of multi-core processors or a migration from deterministic network protocols in a federated architecture, to a publish-subscribe paradigm on top of a high-speed ethernet with nondeterministic network protocols. Design changes can result in unintended feature

interactions and the violation of assumptions due to paradigm shifts. A study of 15 operating system releases from 10 vendors shows that failure rates over multiple major releases stay high and may even increase [Koopman 1999]. The result is a failure density curve across multiple releases, whose shape is illustrated categorically in Figure 7.

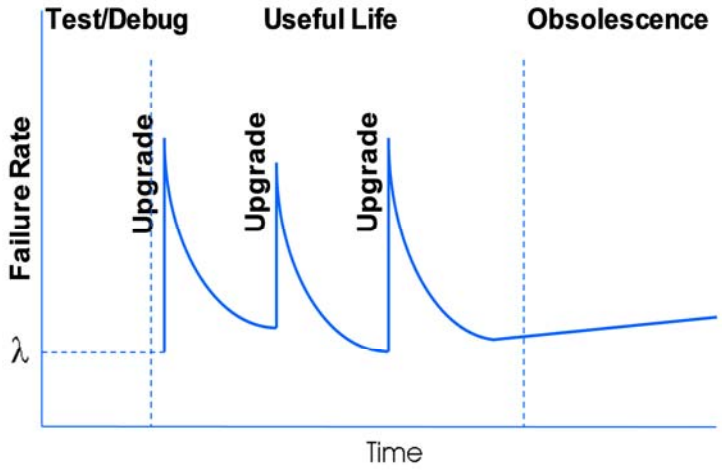

Figure 7: Failure Density Curve Across Multiple Software Releases

A third reason for the trend toward higher risk is that change occurs in the processes, practices, methods, and tools used in the development of embedded software. For example, although Ada has shown to be an excellent choice for the development of highly reliable software, today's marketplace demands the use of C, C++ and Java. The reason is simple: Ada programming skills are scarce, while C, C++ and Java programming skills are plentiful. This has led to efforts in teaching developers safe use of such languages, as in Seacord's work with C-based languages [Seacord 2008]. Similarly, object-oriented design methods now expressed in Unified Modeling Language (UML) were not originally developed with safety-critical embedded software systems in mind, and retrofitting such notations with process standards, such as Motor Industry Software Reliability Association (MISRA) C and C++, to address real-time, reliability, and safety concerns is an ongoing, slow process fraught with pitfalls.

In summary, there is a need to do the following:

- monitor leading indicators of increased risk in evolving software-reliant systems
- investigate potential new problem areas and hazards arising from major capability and technology upgrades
- revise the processes, practices, methods, and tools used to address these risk areas.

2.9 Limited Confidence in Modeling and Analysis Results

Model-based engineering is considered key to improving system engineering and embedded software system engineering. Modeling, analysis, and simulation have been practiced by engineers for a number of years. For example, design engineers have created computer hardware models in Very High-speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) and validated them through model checking [VHDL NG 1997]. Control engineers have used modeling languages such as MATLAB/Simulink to represent the physical characteristics of the system to be controlled and the behavior of the control system. Characteristics of physical system components,

such as thermal properties, fluid dynamics, and mechanics, have been modeled and simulated at various levels of fidelity.

Even for software systems analysis, models and simulation have proven useful in predicting various operational aspects. However, aircraft industry experience has shown that analysis models maintained independently by different teams result in a *multiple truth* problem [Feiler 2009a] (see Figure 8). Such models are created at different times during the development based on an evolving design document and are rarely kept up-to-date with the evolving system design. The inconsistency between the analysis models, with respect to the system architecture and the system implementation, renders the analysis results of little value in the qualification. A need exists for an architecture-centric reference model approach as a common source for system analysis and system generation.

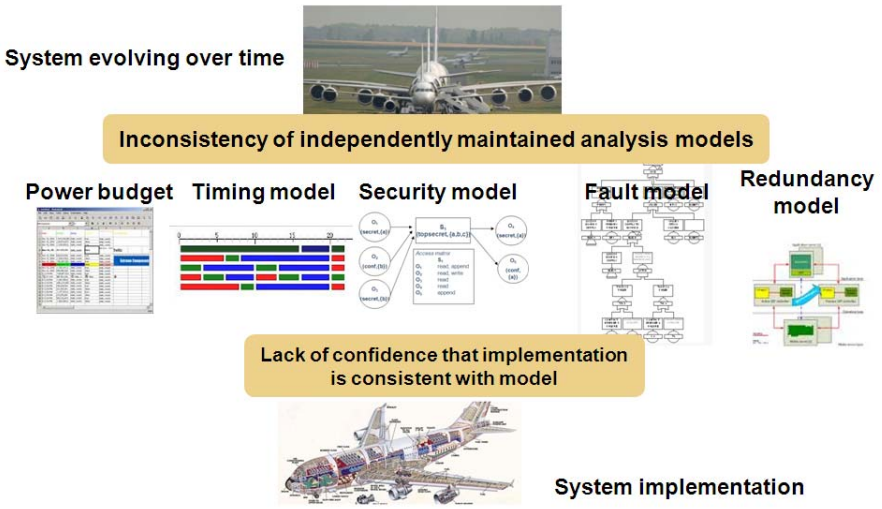

Figure 8: Pitfalls in Modeling and Analysis of Systems

2.10 Summary

The current best development practice—relying on process standards, best practices, and safety culture—is unable to accommodate the exponential increase in the size and interaction complexity of embedded software in today's increasingly software-reliant systems. Traditional reliability engineering has its roots in hardware and assumes slowly evolving system designs with reliability metrics focusing on physical wear. During system design, reliability improvement is achieved through modeling and analysis to identify failure modes. In contrast, software failure modes are primarily design errors, and software is often assumed to be perfectible and to show deterministic behavior. With current best practices as shown in Figure 3, 70% of errors are introduced during requirements and system and software design, while 80% of errors are not discovered until integration and acceptance testing. There is a clear need to reduce the leakage rates of requirements and design errors into later development phases.

Moving beyond or augmenting the textual specification of requirements is essential to validating specification analytically. Similarly, we need architectural models with well-defined semantics that support analysis of nonfunctional mission and safety-critical requirements. Only through such models can we understand, early in the life cycle, how such system properties are impacted by

- architectural decisions
- identifying potentially mismatched assumptions in system interactions.

In system safety analysis, we must take into account hazards due to software malfunction. Fault management has increasingly become the responsibility of the embedded software, requiring increased scrutiny in order to achieve reliability and safety goals. The complexity and the non-deterministic nature of software interaction require the use of formal static analysis methods to increase our confidence in system operation beyond testing. However, analysis results add little confidence to the testing evidence for system qualification unless consistency across analysis models is maintained. Operational work-around, instead of correction, to address software design problems is not a sustainable option and results in reliability degradation over time.

3 A Framework for Reliability Validation and Improvement

In this section, we introduce a framework for reliability validation and improvement of software-reliant systems. The framework integrates four engineering technologies into a practice that improves both the development and qualification of such systems by addressing the challenges outlined in the previous section. Figure 9 illustrates the interplay among four technologies:

1. formalization of mission and safety-criticality requirements at the system and software level
2. architecture-centric, model-based engineering
3. static analysis of mission and safety-criticality-related system properties
4. system and software assurance.

As Figure 9 illustrates, the technologies interact in the proposed framework, as follows:

- Formalization of requirements establishes a level of confidence by assuring consistency of the specifications, as well as their decomposition into subsystem requirements. The requirements are decomposed in the context of an architecture specification.
- The architecture design is expressed in a notation with well-defined semantics for the architectural structure, interaction topology, and dynamics of the embedded software, the computer system, and the physical mission system, refined into component models with detailed designs, and translated into an implementation. This set of evolving models is the basis for virtual integration, that is, the integration of the system through its models. Virtual integration allows for incremental verification and validation (V&V) of mission-related and safety-criticality-related system properties through static analysis and simulation. The annotated architecture model in the model repository is the source of automatically derived analysis models and auto-generated implementations where possible.
- The application of static analysis, such as formal methods, to requirements, architecture specifications, detailed designs, and implementations leads to an end-to-end V&V approach.
- Assurance cases provide a systematic way of establishing confidence in the qualification of a system and its software. They do so through
  - recording and tracking the evidence and arguments, as well as context and assumptions, that the claims of meeting system requirements are satisfied by the system design and implementation, and
  - making the argument that the evidence is sufficient to provide justified confidence in the qualification.

  The assurance case methodology addresses both evidence regarding the system design and implementation and evidence regarding the application of V&V methods.

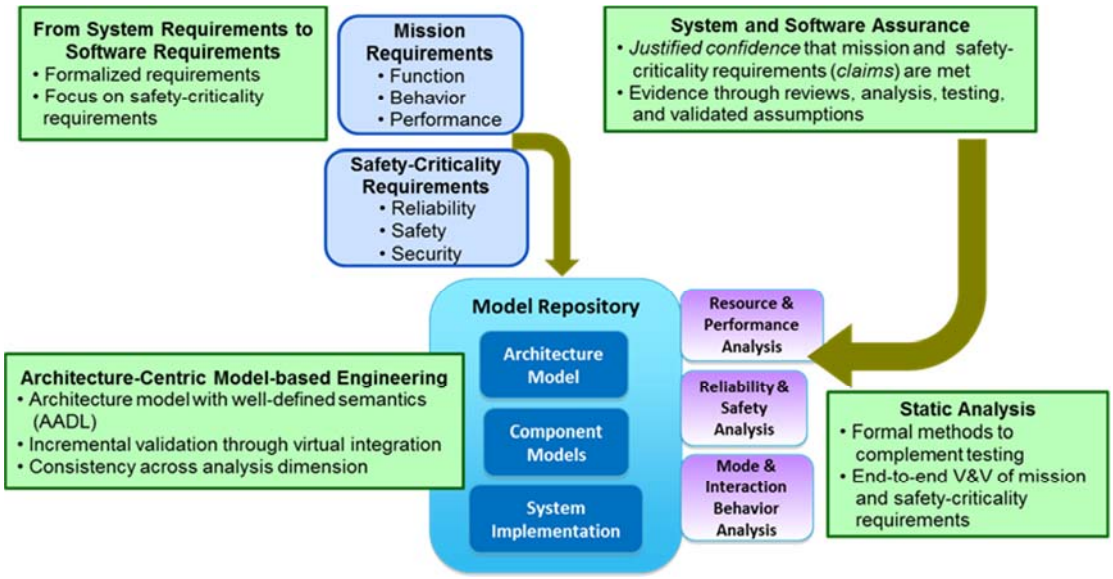

Figure 9: Reliability Validation and Improvement Framework

This framework changes the traditional software development model. The revised development model for software-reliant systems is shown in Figure 10. This revised model consists of two Vs reflecting the development process (*build the system*) and the qualification process (*build the assurance case*) for the system. Both are affected by the use of architecture modeling, analysis, and generation technology. The *build the system* development process covers the life cycle ranging from formalized requirement specification and architecture design, detailed design, and code development, through integration, target, and deployment build. The *build the assurance case* qualification process comprises the traditional unit test, integration test, system test, and acceptance test phases. In addition it covers early-life-cycle phases that bring increased justified confidence in the system, such as requirements validation, system/software architecture V&V, and design V&V through static analysis and simulation.

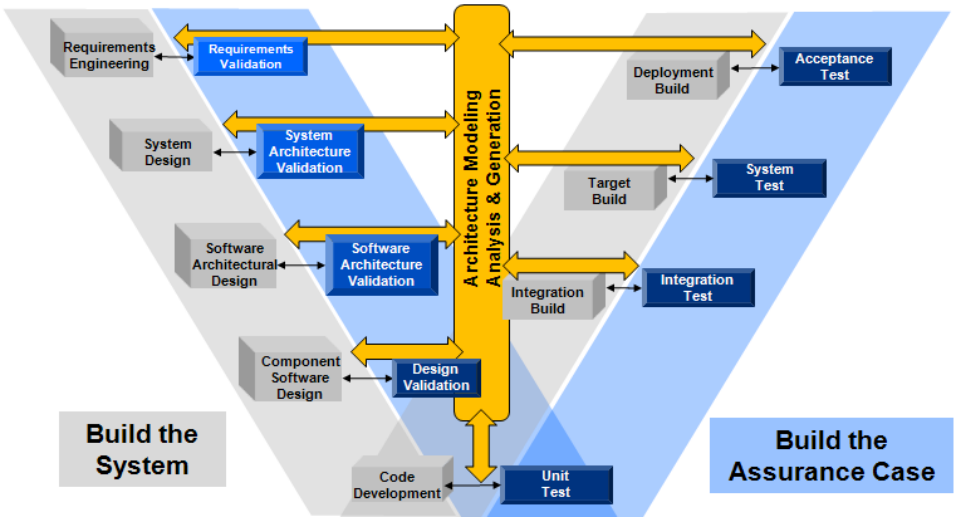

Figure 10: Revised System and Software Development Model

We affect the development of software-reliant systems by discovering errors, in particular system-level errors, earlier in the development life cycle than is done in current practice. This reduces the leakage of errors to later phases and the need for rework and retest—a major cost driver in today's development. In the process, we ensure that all development artifacts—from requirements, to architecture and design models, to implementations and build and deployment configurations—are managed in a consistent manner throughout the development life cycle.

We also affect the V&V with the objective of improving the qualification of software-reliant systems by building the assurance case throughout the life cycle to increase our confidence in the qualified system. In the process, we ensure that all qualification evidence, ranging from validated requirements to analyzed and verified models and implementations, is managed in a consistent manner and evolves in the context of previously validated artifacts.

Finally, we can achieve cost-effective reliability improvement by focusing on high-risk system concerns, such as system interaction complexity and safety-criticality requirements as well as high-payoff areas, namely, system-level problems that currently leak into system integration test and later phases. We achieve this improvement by using virtual integration of architecture models with well-defined semantics and performing end-to-end validation of system properties. The result is a reduction in error leakage to later phases and a major reduction in rework/retest cost.

We proceed by discussing each of the four technologies in terms of their state of the art, contribution to reliability improvement, and interactions with the other technologies in the framework. Then we will outline our approach for the proposed reliability improvement metrics.

3.1 Formalized System and Software Requirements Specification

Requirements are typically divided into business requirements, process requirements, and product requirements. We are focusing on product requirements. Requirements are also divided into functional requirements (what the system is to do) and nonfunctional requirements on the operation (performance, safety, security, availability, etc.) and on the design (modifiability, maintainability, etc.), as well as constraints on the solution (e.g., use of specific technology). A common view for software engineering has been that only functional requirements can be implemented by software and that nonfunctional requirements can be addressed only in the context of the system in which the software is deployed. This separation between system engineering and software engineering has led to system integration problems [Boehm 2006], in particular for software-reliant systems.

There is a clear need for co-engineering of system and software that spans from requirements to architecture design, detailed design, and implementation and that uses formal validation [Boehm 2006, Bozzano 2010]. We need to capture the *shalls* of a system, which tend to focus on achieving the mission under normal conditions. These are the *mission requirements.* We also must capture the *shall nots*, which describe how the system is expected to perform when things go wrong. These are the *safety-criticality requirements*.

Mission requirements address functionality, behavior, and performance under normal conditions. Safety-criticality requirements address safety, reliability, and security, which often involve performance under stress or failure conditions (see Figure 11).

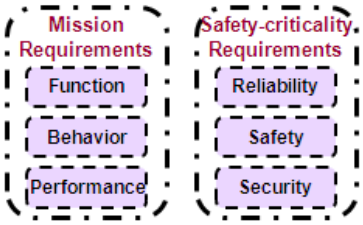

Figure 11: Mission and Safety-Criticality Requirements

In Section 2.3, we identified a clear need for improving requirements capture and validation. A recent industry survey [FAA 2009b] indicates that in DO-178B-compliant practices, requirements are captured in structured text (*shall* statements) with traceability to the design and code as required by practice standards. The challenge is how to formalize the specification of requirements without overwhelming the stakeholders in a system with the formalisms. We proceed by summarizing the state of best practice in formalized requirement specification, linking the specification of requirements to the interactions of the system with its environment, and then discussing a hazard-focused framework for safety-criticality requirements.

3.1.1 A Practical Approach to Formalized Requirements Specification

A method to capture requirements known as Software Cost Reduction (SCR) uses a *Four Variable* model that relies on monitored and controlled variables on the system side and input and output variables on the software side to relate system requirements and software requirements [Parnas 1991]. Miller has proposed an extension to the model that uses tables to represent system state and event/action relations to specify desired behavior and recommends documentation of environmental assumptions [Miller 2001]. This tabular form facilitates coverage and consistency checking [Heitmeyer 1995].

The Requirements State Machine Language (RSML) method refines the tabular representation and adds diagrams to improve the representation of state behavior [Leveson 1994]. Intent Specifications provide an approach to human-centered requirement specification [Leveson 2000]. A commercial toolset supporting the Intent Specification approach, the Specification Toolkit, and Requirements Methodology (SpecTRM) [Lee 2002] includes a behavioral specification language, SpecTRM-RL, which is similar to RSML.

Goal-oriented Requirements Engineering (GORE) [Dardenne 1993, Letier 2002, van Lamsweerde 2004b] goes one step further by including goals and constraints in the requirement specification formalism to complement the event/action model in order to better capture nonfunctional properties [van Lamsweerde 2000, 2004a]. A toolset called Formal Analysis Using Specification Tools (FAUST) supports the capture and analysis of GORE specifications [Rifaut 2003].

Graphical design methods that are often already familiar to engineers include representations for state machines for modeling behavior, such as UML State Charts and Simulink State Flow. Engineers often combine these with user scenarios (e.g., expressed graphically in the use case technique of UML), to express user needs by detailing scenario-driven threads through system functions with the objective of helping derive a system's behavioral requirements.

Based on a study of best practice [FAA 2009b], the FAA developed a handbook that provides practical guidance in formalized requirements capture [FAA 2009a].

A reason for increasing formality in requirement specification is to allow for the validation of requirements with respect to consistency constraint satisfaction. We achieve this by using tools that check for consistency of the specification [Heitmeyer 1995, Lee 2002] and by transforming the specification into a formalism that allows for analysis by formal methods such as model checkers and *provers* for industrial applications [Miller 2010].

3.1.2 Interaction Between the System and Its Environment

We need to specify not only how a system responds to input, but also how it interacts with its environment in other ways (as shown in Figure 12). The Association Française d'Ingénierie Système (AFIS) has defined a process called CPRET that reflects this view; CPRET is "a set of behaviors by execution of functions to transform input into output utilizing state, respecting constraints/controls, requiring resources, to meet a defined mission in a given environment" [AFIS 2010]. Typically requirement specification of systems focuses on the state and behavior of the system and the input/output conditions. This definition provides a more comprehensive coverage of system requirements by including external control imposed on the system and resources required by the system. In other words, a system has four types of interaction points with its environment: input, output, imposed constraints/control, and resource requirements.

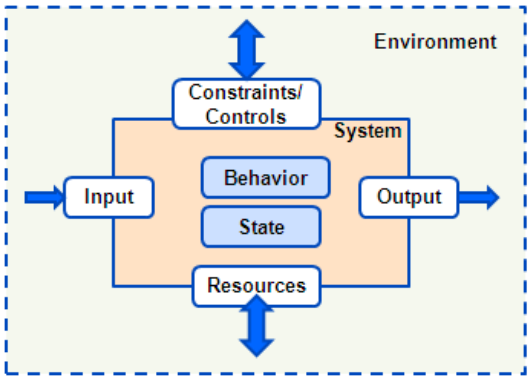

Figure 12: The System and Its Environment

When taking a systems view, we see that the environment itself is a collection of systems. Any system can be a physical/mechanical system, a computer system, a software system, one or more human roles, or a combination thereof. The system of interest interacts with one or more systems in the environment, the combination forming a composite system. The system of interest may itself be composed of interacting systems to act as a whole.

Different types of system interactions are illustrated in Figure 13. The interactions may be in terms of (1) cooperating systems, (2) systems that act as resources to the system of interest, (3) systems that control or constrain the operation of the system of interest, or (4) systems in the environment that may not directly interact but are affected by the operation of a system. The latter, although they do not directly interact with the system of interest, may still represent hazards that affect the ability of the system to achieve its mission by acting as obstacles or by competing for the same resources.

It is desirable to capture expectations and assumptions about these four interaction points in requirements. We can typically express them in requirements by taking into account their temporal

aspects. For example, we may have requirements on the rate and the latency of data in a data stream, on the order of events or commands in an event of command sequence, or on the usage pattern of resources. Some of these aspects have been incorporated into requirement specification methods and into the FAA handbook mentioned in Section 3.1.1.

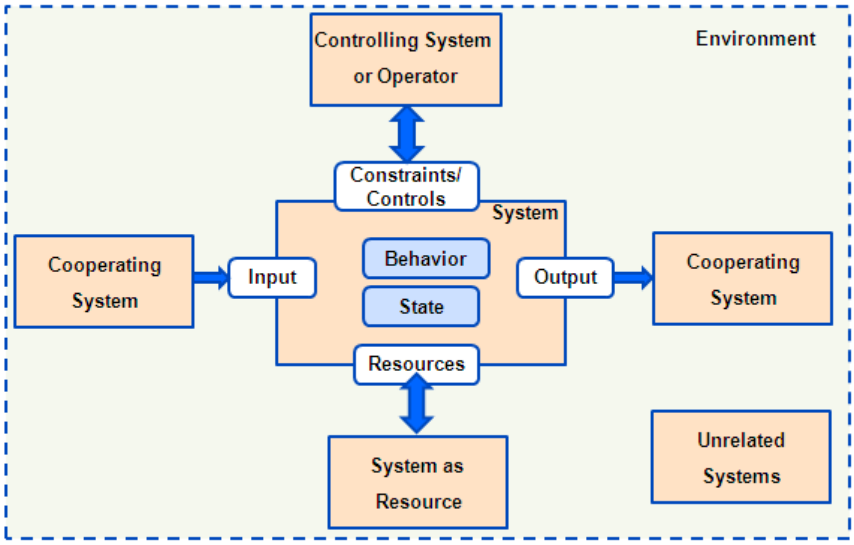

Figure 13: The Environment as a Collection of Systems

The purpose of expressing requirements is to specify the behavior expected of a system (shown in the left-hand image of Figure 14). Users of a system have a set of expected behaviors in mind. The intent of requirements is to define what is expected, but not all of them may be captured and discovered in the course of system development. Unexpected behavior is erroneous behavior that may be tolerated or mitigated. The behavior allowed by a requirement specification, however, may not encompass all of the expected behavior, and it may even include unexpected behavior (as shown in the middle image of Figure 14). A system design or implementation exhibits actual behavior. This behavior may be within the specified behavior (i.e., meet the requirements) or may be outside the specified behavior (as shown in the right-hand image of Figure 14).

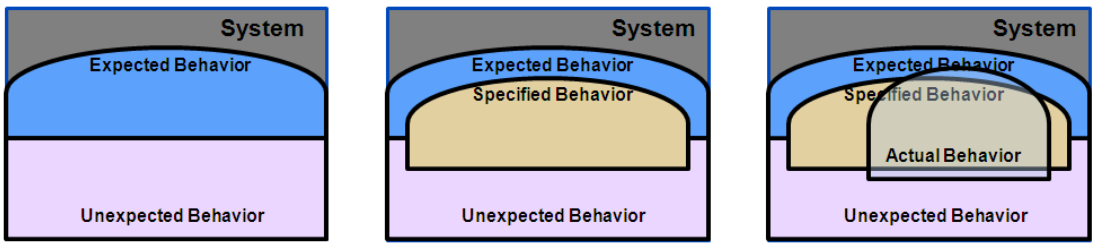

Figure 14: Expected, Specified, and Actual System Behavior

Requirement *verification* ensures that the actual behavior exhibited by the design or implementation satisfies the requirement specification. Note that a design or implementation may meet its requirements, but still exhibit unexpected behavior. Therefore, requirement *validation* and verification are equally important.

Actual behavior that is unexpected behavior presents hazards in that it deviates from the user's expectations. Similarly, actual behavior beyond specified behavior presents hazards because interacting systems have been designed against the specification. These hazards, if not addressed, manifest themselves as error propagations that can result in unexpected behavior.

The purpose of safety-criticality requirements is to specify how to deal with the hazards of unexpected behavior. This is the topic of the next section.

3.1.3 Specifying Safety-Criticality Requirements

In this section, we examine safety engineering approaches and then propose a generalized hazard analysis framework that can be supported by an architecture-centric model-based engineering approach based on the SAE AADL [SAE 2004-2012].

Safety-criticality requirements address three system properties.

1. reliability: the ability of a system to continue operating despite component failure or unexpected behavior by system components or the environment
2. safety: the ability to prevent unplanned and unacceptable loss as a consequence of hazards (failure, damage, error, accident, injury, death)
3. security: the ability to prevent error propagation and the potential harm that could result from the loss, inaccuracy, alteration, unavailability, or misuse of the data and resources that a system uses, controls, and protects.

Satisfying any one of those system properties presents challenges. But all three, together with real-time performance, must be satisfied at the same time, an achievement that requires compatibility across the properties and consistency across the approaches supporting their analysis [Rushby 1994]. However, current practice tends to treat each of these safety-criticality dimensions separately.

Safety engineering practice standards, such as the FAA *Systems Safety Handbook* [FAA 2000] and the *Safety Assessment Process on Civil Airborne Systems and Equipment* [SAE 1996] recommend hazard identification through FHA that involves identification of failure modes and their manifestation in terms of externally observable behavior This step is typically followed by a system safety analysis (SSA) and common cause analysis (CCA), for analyzing the effect of hazards and its potential negative impact. As the system design progresses, fault tree analysis (FTA) and failure mode and effect analysis (FMEA) are used to aid in

- understanding the impact of faults and hazards within the system
- making architectural decisions to prevent or manage faults and hazards in order to satisfy the reliability or safety requirements.

These hazards are then translated into safety requirements on the system or its components. This translation involves specifying additional constraints on the state and behavior of the system and its interactions with the environment. These constraints must be satisfied in order to prevent the hazard from propagating or to determine whether it is an acceptable risk to propagate the hazard. The progression from hazard analysis to requirement specification has been demonstrated by Leveson and by Miller. Leveson's team integrated the hazard analysis method called "STAMP to

Prevent Accidents" (STPA), which draws on (STAMP)[12] [Leveson 2009], with intent specification via the Specification Tools and Requirements Management (SpectTRM) toolset and SpecTRM-RL modeling language [Herring 2007, Owens 2007]. Miller and associates integrated FHA and FTA with a state-based safety requirement specification that can drive V&V through model checking and proof engines [Tribble 2002, Miller 2005a, 2005b].

3.1.4 An Error Propagation Framework for Safety-Criticality Requirements

Figure 15 illustrates an error propagation framework that provides a unified view of safety, reliability, and security faults and hazards. This framework is reflected in the Error Model Annex of the AADL standard, providing a single architecture-centric model source for driving the analysis and validation of all three safety-criticality requirements (see Section 3.2.1). It consists of the ability to specify

- sources of errors, such as faults, expressed as error events
- the error behavior of systems and system components in terms of error states and transitions to represent error-free and failure behavior
- the ability of other systems or system components to observe the failures of a system or system component in the form of error propagations.

Error propagations represent failures of a component that potentially impact other system components or the environment through interactions, such as communication of bad data, no data, and early/late data. Such failures represent hazards that if not handled properly can result in potential damage. The interaction topology and hierarchical structure of the system architecture model provides the information on how these error behavior state machines interact through error propagation.

The concepts of expected and unexpected behavior illustrated in Figure 14 relate to the concepts of error events, error states, and error propagations as follows. Unexpected behavior can be due to a fault in the system or system component, which is expressed as error event. Unexpected behavior can also be due to error propagation from interacting systems or the environment. A fault in the system can be a requirement fault such as an incomplete or omitted requirement, a design fault such as an error in the fault management logic, or an implementation fault such as a coding error in handling measurement units. The fact that a system or component has unexpected behavior is represented by the component entering an error state representing failure. An error state represents a hazard that, if not addressed locally, results in error propagation, which impacts interacting system components or the environment.

Figure 15 illustrates the distinction between two kinds of error propagation. One kind is propagation of errors over modeled interaction channels such as port-based communication. For example, an actuator may send the wrong voltage to a motor over an electrical wire, or a software controller may send a control signal to an actuator too late. This kind of error propagation is shown as *unexpected* interaction over specified interaction channels. The second kind of error propagation occurs between components without explicitly modeled interaction channels. For example, the increasing heat of a motor in use may raise the temperature of a nearby processor, causing it to run slower; or a piece of software may exceed an array bound overwriting data of other software. This

[12] STAMP, Systems Theory Accident Model and Processes, is an accident model based on systems theory.

kind of error propagation is shown as *unintended* interactions. Both these forms of error propagations represent potential safety and security hazards.

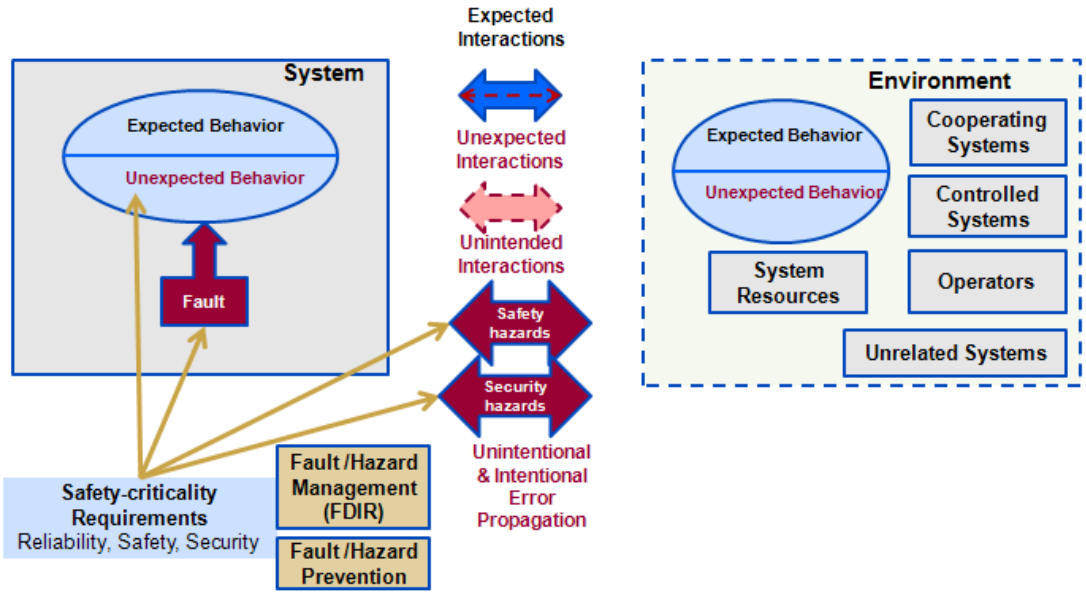

Figure 15: Capturing Safety-Criticality Requirements

A safety hazard is an error propagation from the system to the environment that can cause damage to a system in the environment. A safety hazard also exists if the system shows expected behavior, but an interacting system in the environment shows unexpected behavior (e.g., an operator sticks his finger into a fan). It can also be due to a change in the environment that results in behavior different from that assumed by the system (e.g., a software controller assumes metric units for sensor data readings while the new sensor provides them in imperial units).

Figure 12 in Section 3.1.2 shows four classes of specified interaction points: input, output, control, and resource. We have incoming hazards (expressed as error propagations) through inputs and outgoing hazards through outputs. In the case of interaction with a resource, incoming hazards are related to available resource capacity, while outgoing hazards are related to resource demand by the system. In the case of the control interaction point, the hazards are related to outgoing observations (e.g., sensor output) and incoming control actions (e.g., actuator input). These hazards, if not mitigated locally, become error propagations to be addressed by the recipients. These error propagations can have the following characteristics:

- Commission: providing output (data, events, signals, hydraulic pressure, etc.), providing control status, or requesting resources when not expected propagates outgoing error; receiving input, receiving control input, or being allocated resources when not expected propagates incoming error.
- Omission: not providing output or not receiving input when expected (and equivalent for control and resource) propagates error.
- Bad value: inaccuracy, value out of range, imprecision, and mismatched unit are examples of bad value.

- Early/late: for time-sensitive interactions the input, output, control or resource may be provided too early or too late.
- Too long/too short: for time-sensitive interactions with duration, input, output, control, or resource may be available for too short or too long a time.
- For interactions that are streams or sequences, these characteristics may apply: wrong rate, variation in time interval, variation in value difference, missing element, wrong order.

We can attach these error propagation types to the appropriate interaction point in the system specification together with

- the condition that must be violated for the propagation to occur and
- an indication of whether the error propagation must be prevented by the outgoing interaction point or is expected to be tolerated or handled by the recipient.

In other words, we are recording fault prevention and mitigation requirements that must be fulfilled by the fault management architecture of the system or the tolerance of hazards that is expected of the environment.

Unintended error propagations occur because the system and its environment are not guaranteed to be isolated from each other (e.g., they may share resources). Examples of resource sharing are processor, memory, and network for software; physical location, electrical power, and fuel are examples for physical hardware.

It is desirable to limit such unintended error propagations through enforceable isolation mechanisms, such as

- radiation hardening of hardware
- limits on electrical power consumption through fuses
- enforcement of value limits through filters
- runtime enforcement of address spaces or execution time budgets for software.

Using isolation mechanisms can greatly reduce the types of error propagations we have to deal with. For example, we can place software subsystems in separate operating system processes with virtual memory that enforces address space boundaries at runtime. This causes all faults within the software subsystem[13] to manifest themselves as error propagations of one of the above types through the known interaction points. Such isolation may not always be feasible unless we have an understanding of the architecture of the systems in the environment that interact with the system of interest.

Security hazards are typically viewed as intentional propagation of errors into a system. They take advantage of a system's not detecting error propagation or of faults within the system that can cause the system to enter unexpected behavior despite expected behavior by the environment. Unexpected behavior of the system can also result in error propagation that can cause a security problem in the environment. In other words, security has to be managed at the architectural level since it is driven by the interaction between systems.

[13] Some examples of these faults are division by zero, index out of bounds, or computational errors.

Reliability requirements focus on the ability of the system to continue to operate despite failure of system components or external hazards that cause system components or the system as a whole to fail. Component failures are addressed through fault management strategies such as redundancy within the system or by a composite system that includes the system of interest. Hazard-induced failures are also addressed through fault detection, isolation, and recovery (FDIR) through the enclosing composite architecture. Architectural solutions to fault and hazard management are discussed in the next section.

3.2 Architecture-Centric, Model-Based Engineering

Modeling, simulation, and analysis are essential parts of any engineering discipline. It is important that models have well-defined semantics in order for their simulation and analysis to produce credible results. Modeling notations for describing a system or software present the challenge of representing a cohesive and consistent whole when expressing different aspects of the system architecture and detailed design in different diagrams [UML OMG 2009]. Similarly, analysis models that are maintained independently by different teams present the challenge of producing credible results by maintaining consistency with system architecture, other analysis models, and the system implementation (see Section 2.9).

Therefore, it is essential that the model-based engineering approach is architecture-centric and uses as its foundation a standardized underlying meta model with well-defined semantics that can drive the analysis and generation of a system. We proceed by discussing such an architecture-centric model-based engineering approach based on an international industry standard. Then we elaborate on the relationship between requirements and architecture, focus on safety-critical system architectures, and discuss the practicality of an industrial approach for architecture-centric analysis and construction of software-reliant systems.

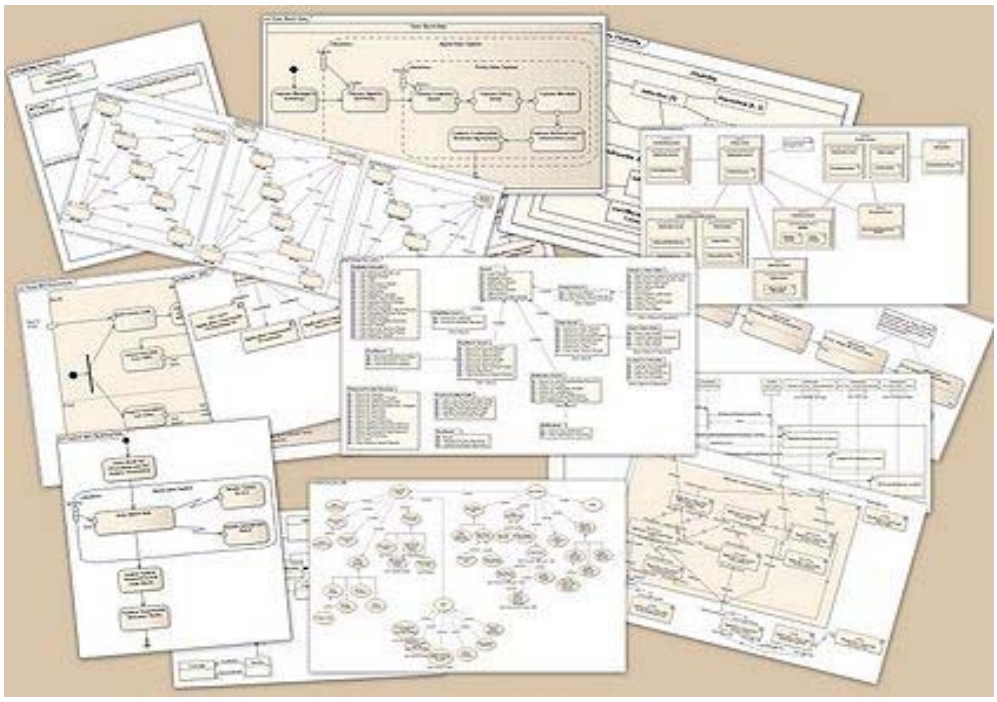

Figure 16: Collage of UML Diagrams [Wikipedia 2011]

3.2.1 An Industry Standard-Based Approach to Architecture-Centric, Model-Based Engineering

The SAE Architecture Analysis & Design Language (AADL) was developed specifically for modeling and analyzing the architecture of embedded real-time systems (i.e., safety-critical software-reliant systems in support of model-based engineering) [SAE 2004-2012, Feiler 2012]. AADL focuses on modeling the system architecture in terms of the software architecture, computer system architecture, and the physical system, as well as the interactions among the three, as illustrated in Figure 17. It supports component-based modeling through a textual and graphical notation, as well as a standardized meta model with well-defined semantics for the language concepts that provides a standardized eXtensible Markup Language (XML) Interchange (XMI) format for exchange of AADL models and for interfacing AADL models with analysis tools.

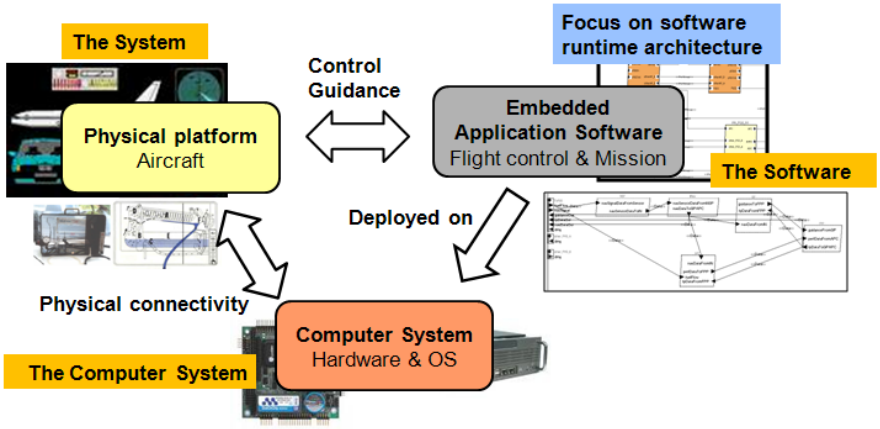

Figure 17: Software-Reliant System Interactions Addressed by AADL

AADL defines component concepts specific to representing the software architecture (process, thread, subprogram, data), the computer system (processor, memory, bus, virtual processor, virtual bus), the physical system (device), as well as their aggregation (subsystem or system). It includes concepts of representing component interactions both at the logical level (connections for sampled data ports, and queued event and message ports, shared data access, and service request/call-return) and the physical level (connectivity via bus access), end-to-end flow specification including flow abstraction in component interfaces, operational modes to characterize dynamic changes to the architecture such as reconfigurations, and specification of deployment binding of software to hardware.

AADL supports modeling of large-scale systems and families of systems in several ways. It supports component-based modeling with separation of component interface specification (component type) and multiple variants of implementation blueprints (component implementation) for each interface specification. AADL supports incomplete component specifications that can be refined, including abstract components and parameterized component type and implementation templates. Finally AADL provides a package concept similar to those found in programming languages to support the organization of the component specifications into a hierarchy of packages. AADL has a set of predefined properties. Its core language is extensible through a notation for introducing user-defined properties and an annex sublanguage mechanism to introduce additional concepts that can be associated with an AADL model [SAE 2004-2012].

AADL is a notation with strong typing, whose value to building reliable software systems has previously been demonstrated by Ada[14] and VHDL. The AADL compiler ensures model completeness and consistency: threads can only be contained in thread groups and processes; buses cannot have ports; the data type of ports match; and sampling ports have only one incoming connection per mode. The AADL standard defines the timing semantics of thread execution and communication, including deterministic sampling by periodic threads, as well as mode transitions. This level of specification leaves little room for misinterpretation of the intended execution behavior and allows for generation of analytical models such as timing models for scheduling analysis.

The extensibility of AADL through property sets and annex sublanguages, whose meta model is semantically consistent with the core AADL, allows AADL to support analysis along multiple quality attribute dimensions from the same model source. This is illustrated in Figure 18 and has been demonstrated in the context of an open source AADL tool environment (OSATE), which is based on Eclipse.[15]

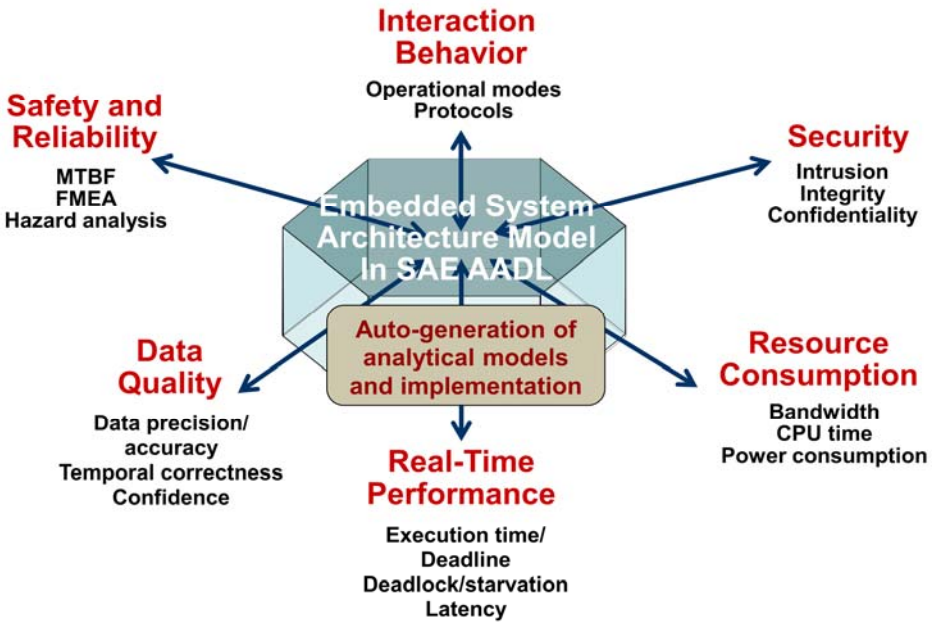

Figure 18: Multidimensional Analysis, Simulation, and Generation from AADL Models

More than 40 research groups and advanced development teams have integrated their formal analysis and generation frameworks into AADL as evidenced by reports in over 250 publications.[16] In this section, we provide a sampling of that work; in Section 3.2.4, we discuss industrial initiatives that have integrated and expanded a range of analysis and generation technologies into AADL to advance architecture-centric, model-based engineering.

---

[14] For more information, see the ongoing Ada Europe Conference series examining reliable software technologies (http://www.ada-europe.org/).

[15] For more information on OSATE, visit http://www.aadl.info/aadl/currentsite/tool/osate.html. For information on Eclipse, visit http://www.eclipse.org/.

[16] The public AADL wiki maintains an annotated list of publications related to AADL (https://wiki.sei.cmu.edu/aadl/index.php/AADL_Related_Publications).

Below are examples of research and advanced development efforts where formal analysis and generation frameworks are integrated into AADL:

- **Resource allocation and scheduling analysis** is supported by mapping AADL models into timed Petri nets, process algebras (VERSA) [Sokolsky 2009], timed automata (Cheddar) [Singhoff 2009], rate monotonic analysis (RMA), and constraint-based resource allocation by binpacking [DeNiz 2008].
- **Flow latency analysis** is supported through component input/output flow specifications and end-to-end flows using latency properties and interpreting the execution and communication timing semantics of partitions and threads, as well as the hardware they are deployed on [Feiler 2008].
- **Security analysis** with respect to confidentiality is supported by mapping the Bell LaPadula security analysis model into AADL through a set of security properties and its analysis supported through an OSATE plug-in. Later this has been extended to support security analysis in the context of Multiple Independent Levels of Security (MILS) architecture [Hansson 2009, Delange 2010a].
- **Resource consumption analysis** is supported for computer resources such as processor cycles, memory, and network bandwidth, as well as physical resources such as electrical power through resource capacity and budget properties [Feiler 2009a].
- **Data quality analysis** is supported through additional properties of the data content, its representation in variable base types, and in mapping of data into different protocols [Feiler 2009a].
- An **Error Model Annex standard** [SAE 2006] was added to the AADL standard suite based on fault concepts introduced by Laprie [Laprie 1995]. The annex supports the annotation of AADL models with fault sources, error behavior of components and systems, error propagation between system components, and mappings between the error behavior in the error model and the fault management architecture in the core AADL model. The annotated model has become the source for various forms of reliability and safety analysis ranging from reliability predictions such as Mean Time To Failure (MTTF) to FHA, fault impact such as FMEA, and FTA [Rugina 2008, Joshi 2007, Bozzano 2009].
- A **Behavior Annex standard** [SAE 2011] has been added to the AADL standard that supports the annotation of behavioral specifications beyond the architectural dynamics expressed by AADL modes. The Behavior Annex focuses on specification of component interaction semantics and individual component semantics. Annotated AADL models have thus become useful as a source for temporal reasoning [Berthomieu 2010, Bozzano 2010].
- The **ARINC653 Annex standard** [SAE 2011] combined with security and safety properties has been used to validate safety and security properties in partitioned architectures [Delange 2009], and implementations of validated ARINC653 architectures have been generated, including ARINC653-standard-compliant, XML-based configuration files [Delange 2010a].
- AADL has been integrated with **detailed design specifications**
  - expressed in Esterel Safety Critical Application Development Environment (SCADE), using the SCADE-Simulink Gateway, and
  - combined with **data models** expressed in Abstract Syntax Notation version 1 (ASN.1)

to create system models, analyze them, and generate implementations [Perrotin 2010, Raghav 2010].

In summary, the value of an architecture modeling notation with well-defined semantics that is extensible as the source of multiple analysis dimensions, simulation, and code generation has been recognized by the research community, evidenced in the analysis frameworks integrated with AADL, and by industry, evidenced by the range of pilot projects (some of which are discussed in Section 3.2.4).

3.2.2 Requirements and Architectures

As discussed in Section 3.1, there is a need to associate requirements with a system architecture for two reasons:

1. The system interacts with a set of systems in its deployment environment, representing a composite system.
2. The system requirements must be satisfied by a system design.

The system architecture, as a result, imposes requirements on the subsystems. Behavioral specifications in terms of state machines have been decomposed into hierarchical state machines and analytically validated [Heimdahl 1996]. More recently, formalized requirement specifications have been associated with architecture models (e.g., requirements expressed with GORE were associated with AADL models to drive several analyses [Delehaye 2009]). The same requirements formalism has been explored as a basis for system safety as an emergent property in composite systems to validate requirements coverage and identify possible gaps by validating them against simulations of the system [Black 2009].

Many requirements can be mapped into constraints on discrete system states, typically expressed in terms of state machines, as well as continuous value states common in physical systems, typically expressed as continuous time functions such as differential equations. It is desirable to associate both types of constraints with components in a system architecture, as done through state charts and parametrics in SysML [SysML.org 2010]. The same is achievable with AADL, and proposals have been made to the AADL standards committee to provide a standardized constraint language annex. Furthermore, it is desirable to support the development of safety requirements through the ability to represent hazards in the context of a system model (see Section 3.1.4), which is supported by the AADL Error Model Annex standard.

As discussed in Section 3.1.2, requirements can be associated with a system (component) itself if it concerns the system state or behavior, or with one of its interaction points in terms of input/output, resource use, or external control. Once thus related, these requirements are associated with an architecture model on which we can perform checks on two types of consistency:

1. consistency between requirements of subcomponents and the requirements of the containing system
2. consistency between requirements that interacting system components place on each other

In the next section, we discuss how architecture patterns can be used to address safety-criticality requirements.

3.2.3 Safety-Critical, Software-Reliant System Architectures

Safety-critical, software-reliant systems have two elements to their architecture: (1) the nominal mission (application) architecture with its fault and hazard potential and (2) the fault management architecture that addresses reliability, safety, and security concerns by mitigating error propagations.

In Sections 3.1.2 and 3.1.4, we outlined an approach to model the faults, hazards, and safety requirements of a system in terms of its external interface and observable states and behavior. Here we are examining

- the interactions between system components in terms of faults and hazards
- how faults and hazards within the composite system affect its realization as a set of interacting components.

The simplest form of interaction is a flow pattern (pipeline) of system components that process a data, event, or message stream in multiple steps. In this case, the faults and hazards associated with the outgoing interaction point of one component (expressed as outgoing error propagations in the AADL model) must be consistent with the incoming error propagations specified by the recipient. They must be in agreement as to which hazards and faults are propagated and which ones are expected to be prevented (masked) by the sender. In addition, the protocol used in the interaction (e.g., a locking protocol for shared data or a communication protocol for port connections) may have its own fault and hazard potential and contribute to the potential error propagations to the recipient.

When the components in the pipeline use resources, as is the case for software, we have another set of error propagation paths based on the assignment of the resources to the system components. In the case of software components, this set of paths is expressed by the deployment binding of the software to the hardware, as Figure 19 illustrates. For example, a processor failure may result in an omission hazard that is propagated to the software component, which, in turn, results in an outgoing omission hazard. Similarly, a failure in the network hardware propagates as an omission hazard to the connection, which, in turn, results in an omission hazard of the communicated information. The AADL Error Model Annex standard defines the possible, explicitly modeled, error propagation paths in an AADL model and provides a mechanism to express the effect of an incoming hazard (error propagation) on the error behavior of the component itself (expressed as a transition in error state machine) and on the outgoing error propagation (expressed as an outgoing propagation guard).

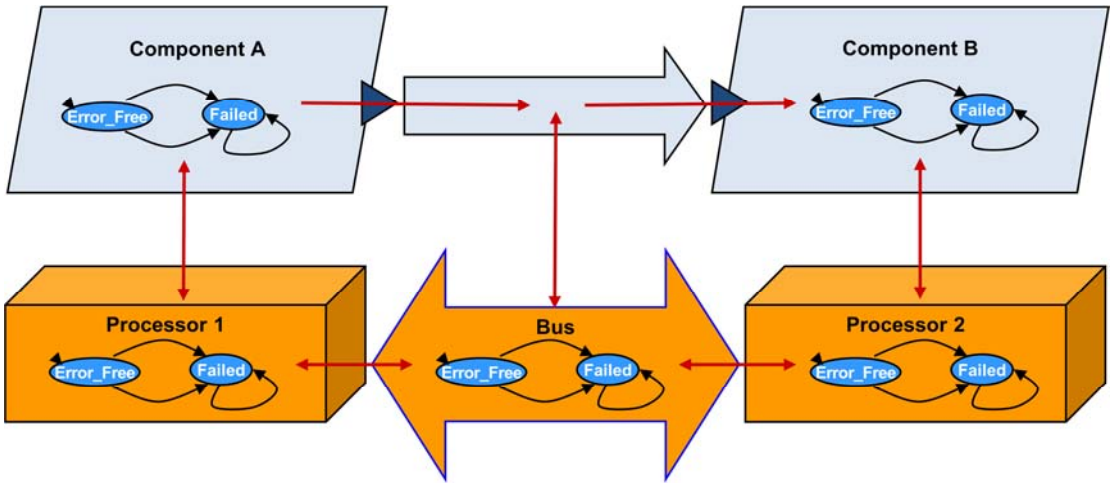

Figure 19: Error Propagation Across Software and Hardware Components

We can identify three other application architecture interaction patterns: (1) two-way, peer-to-peer cooperation, (2) feedback control interaction, and (3) multi-layered service interactions. The co-operation pattern is shown in Figure 20. In this case, the two parties interact in a two-way exchange of data, events, or messages (e.g., in the form of an application-level handshaking or synchronization protocol). Each system has a model of the expected behavior and current state of the other system and acts accordingly. An example of a new potential hazard area is that of a deadlock hazard when the model of the other system's state does not correspond to the actual system state.

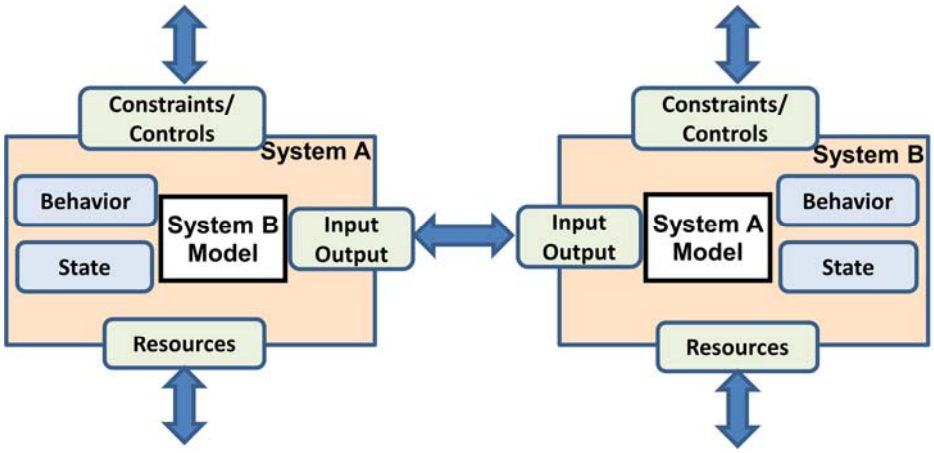

Figure 20: Peer-to-Peer Cooperation Pattern

Leveson uses a feedback control pattern exclusively to analyze the safety of systems [Leveson 2009]. This pattern is shown in Figure 21 with a controller and a controlled system component as well as sensor and actuator components that connect the two. The figure also shows various hazards that can potentially be propagated between the components and faults (labeled with circled numbers) that manifest themselves in potential hazard propagations.

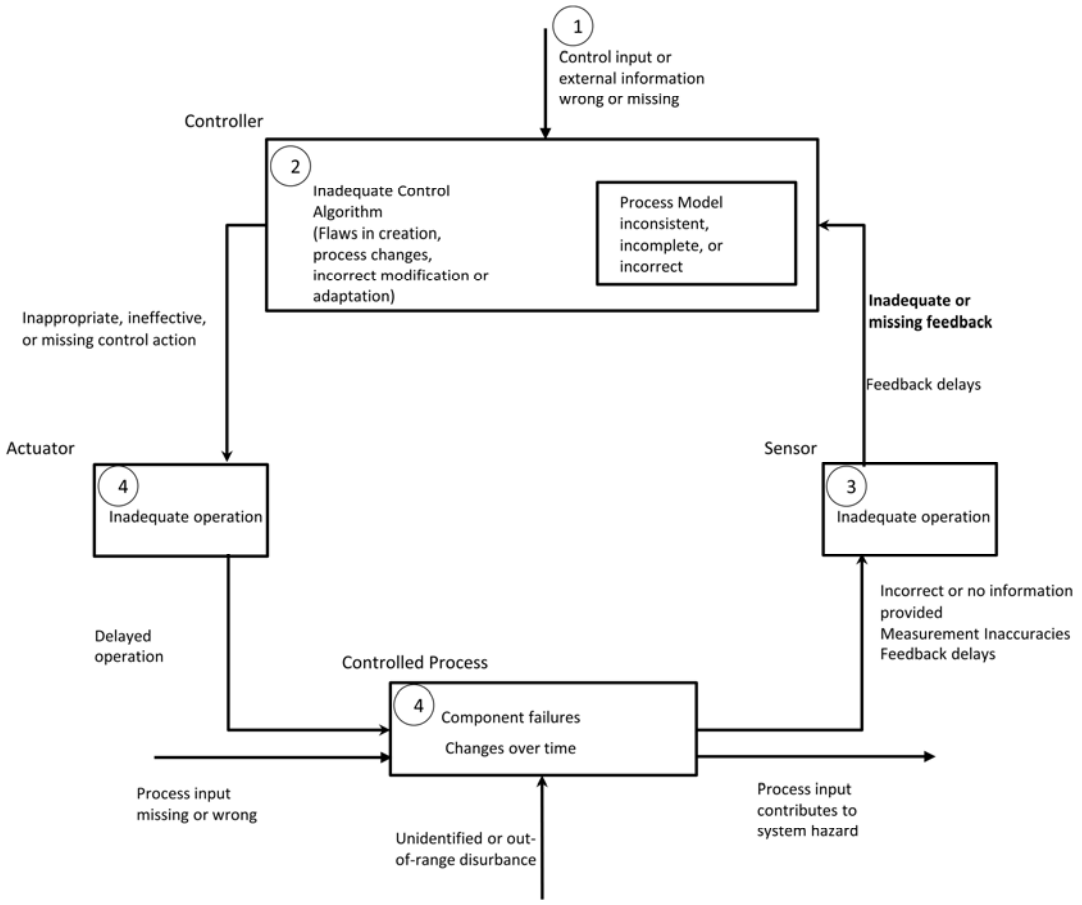

Figure 21: Feedback Control Pattern [Leveson 2009]

Figure 22 depicts a system instance with a combination of these patterns where there is a flow on the left, peer-to-peer cooperation on the right, internal feedback control of a resource on the bottom left, an external feedback control on the top center, and multiple service layers in terms of resource usage. We will use this figure to discuss the relationship between the faults and hazard propagations within a system and their manifestation as potential hazard propagations at the system level. Faults within system components manifest themselves as propagated hazards, and only those components that interact with the system's interaction points affect outgoing hazards and are affected by incoming hazards.

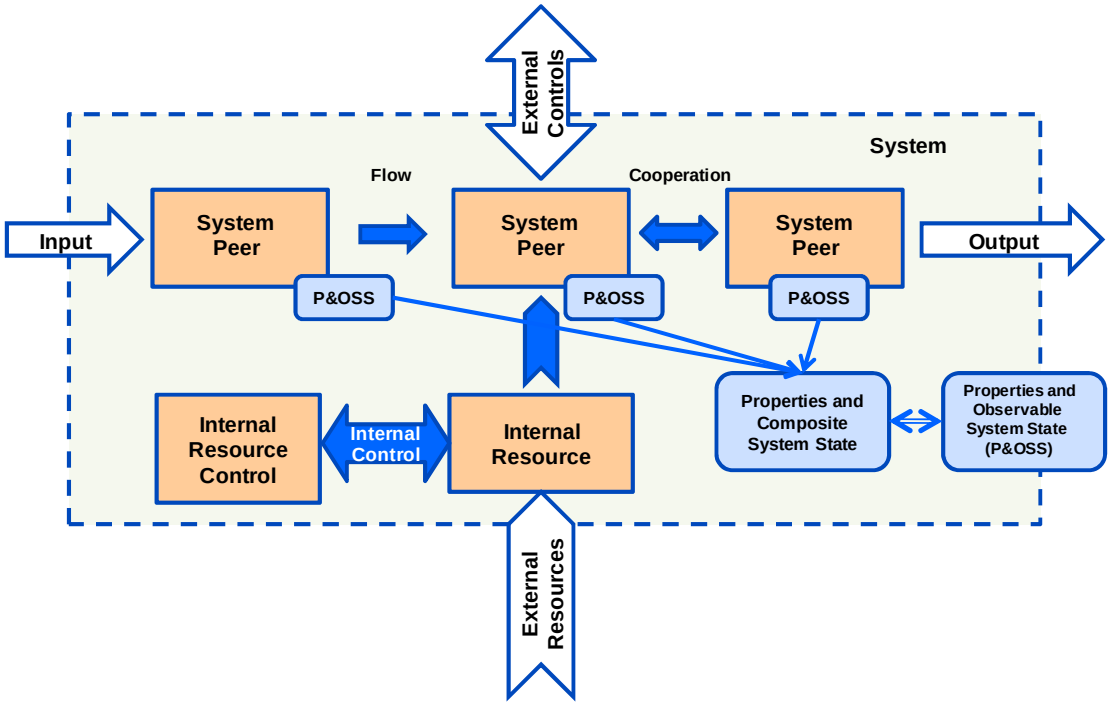

Figure 22: Multiple Interaction Patterns and Composite System

As discussed in Section 3.1.4, a system or system component can act as an isolation boundary for propagation of certain faults and hazards. In this case, we can exclude them from consideration as resulting in unintended error propagation. In the case of physical or computer systems, this is typically achieved through physical separation. For example, the system might be a box with a known set of connection sockets; heat transfer could be an unintended interaction if airflow through a vent is not explicitly represented. In the case of software, partitioning in terms of space and time is a key concept for providing isolation and reducing the interaction complexity of fault and hazard propagation (see the work of Rushby [Rushby 1999]). Partitioning allows us to map a wide range of software design and coding faults (defects) into their manifestation as a much smaller set of propagated hazards through explicitly specified interaction points. For example, logical design errors in the decision logic or algorithm, as well as coding errors such as array index out of bounds, can be mapped into externally observed error propagations in the form of no output, bad output, early/late output, and so on.

One pattern of fault/hazard isolation uses partitioning. Other fault management patterns are redundancy patterns, and monitoring and recovery patterns. Examples of redundancy patterns are replication patterns (dual, triple, quad) with identical copies or dissimilar instances. Often identical copies of software are replicated across redundant hardware to accommodate for hardware redundancy. Sometimes software redundancy is addressed through independent software designs (N-Version programming). Other redundancy patterns take the form of a recovery pattern allowing for dissimilarity, such as the Simplex architecture [Sha 2009]. This pattern supports tolerance to software fault in software upgrade scenarios, such as upgrading the controller software in a software-fault-resilient manner. Monitoring and recovery patterns represent the interactions between the health monitoring/fault management system and the application system, with observation interactions and control interactions, as well as fault management decision logic.

These fault management patterns have hazards in their own right (see Section 2.7). For example, partitions introduce timing-related hazards due to the virtualization of time (input is not sampled at the periodic frame start time, but rather at the start time of the partition slot within the frame) and changes in latency (see Section 2.4). Similarly, a replicated identical software pattern to support hardware redundancy has the hazard of a collocated deployment binding. This means that, although the software is replicated, both copies of the software may be deployed on the same hardware and thus fail to provide redundant service if the hardware fails. Finally, the monitoring and recovery pattern has a number of hazards related to the fault management mode logic [Miller 2005a].

Finally, we need to assess the existing fault management patterns for their ability to provide robustness [DeVale 2002] (i.e., resilience to unknown and unintended fault hazards [Sha 2009]).

3.2.4 Piloting Architecture-Centric, Model-Based Engineering

The SAE AADL standard has 30 voting member organizations that have participated in and contributed to the development of the AADL standard suite. Many of them use the technology of the aircraft, space, and automotive industry. They have also become early adopters of the standard as a technology. Some of the major industry initiatives are shown in Figure 23.

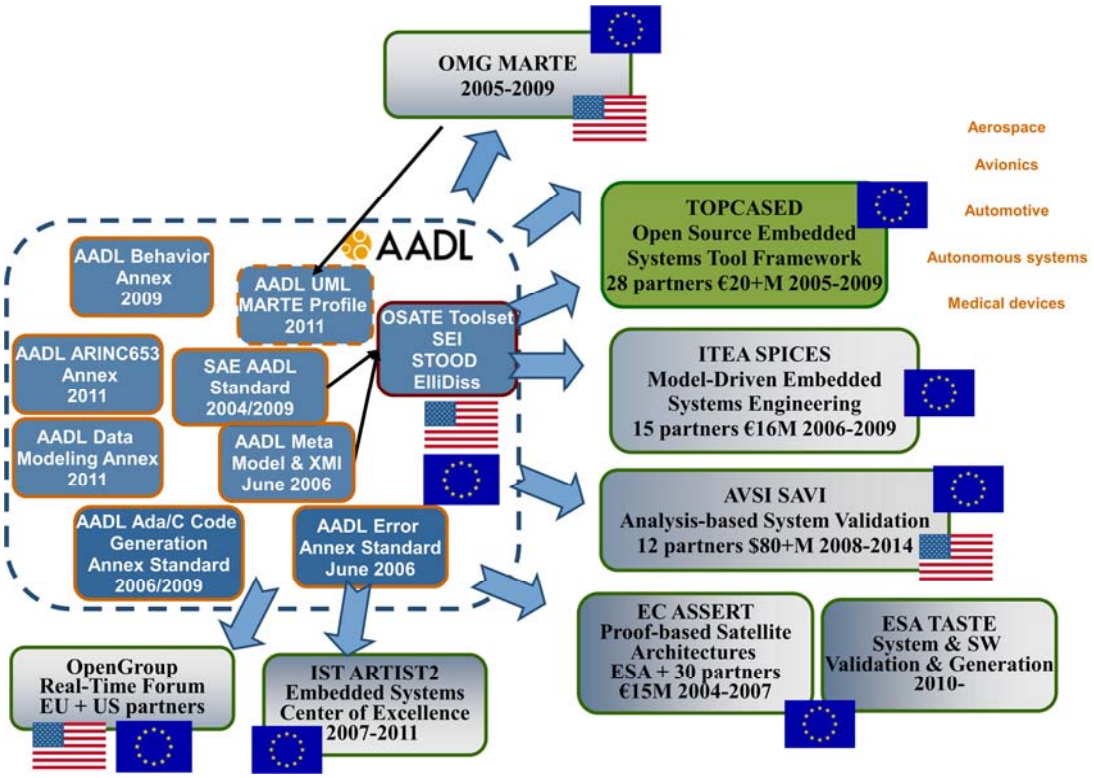

Figure 23: Industry Initiatives Using AADL

The first initiative was led by the European Space Agency (ESA). This initiative, called Automated proof-based System and Software Engineering for Real-Time applications (ASSERT) [Conquet 2008] focused on representing two families of satellite architecture in AADL, validating them, and generating implementations from the validated architecture. In a follow-on initiative,

the ESA is providing an architecture-centric, model-based development process supported by a tool chain, The ASSERT Set of Tools for Engineering (TASTE) [Perrotin 2010].

A second initiative was led by Airbus Industries to focus on the creation of an open source toolkit for critical systems called TOPCASED [Heitz 2008]. This toolkit is based on Eclipse and provides a meta-model-based environment for supporting model-based engineering of safety-critical systems. It uses a model-bus concept to support model transformation between different model representations in the model repository and during interface with analysis and generation tools. AADL is one of the supported modeling notations, and OSATE has been integrated with TOPCASED.

Another European initiative is Support for Predictable Integration of mission Critical Embedded Systems (SPICES) [SPICES 2006]. SPICES integrates the use of AADL with formalized requirement specification, the Common Object Request Broker Architecture (CORBA) Component Model (CCM), and SystemC into an engineering framework for formal analysis and generation of implementations. Some examples of this work are the GORE-based requirements integration with AADL [Delehaye 2009], an adaptation of an electrical power consumption analysis toolbox with AADL [Senn 2008], and a prototype for behavioral verification of an AADL model with Behavior Annex annotations using the TIme petri Net Analyzer (TINA) toolkit [Berthomieu 2010]. The SPICES methodology and tools have been piloted on avionics, space, and telecommunication applications.

Modeling and Analysis of Real-Time Embedded systems (MARTE) is an effort by the Object Management Group (OMG) to provide a UML profile for embedded system modeling that draws on the AADL meta model and its semantics and includes a profile subset to represent AADL models [OMG MARTE 2009].

ARTIST2[17] is a primarily European network of excellence on embedded systems design that provides a forum for researchers and industry practitioners to exchange ideas on the advancement of embedded system design technology. It has cosponsored the international UML and AADL workshop series and organized a number of conferences and other workshops at which AADL-related work and other model-based technologies have been presented and discussed, including a workshop on Integrated Modular Avionics (IMA).

The Open Group[18] is an industry-focused organization that promotes the adoption of new technologies. Its Real-Time Forum has examined AADL as a promising technology internationally.

The System Architecture Virtual Integration (SAVI) [Redman 2010] is an international aircraft industry-wide, multi-year, multi-phase initiative (see Figure 24 and Figure 25) to mature and put into practice an architecture-centric, model-based engineering approach based on industry-standard model representation and interchange formats. Members include aircraft manufacturers (Boeing, Airbus, Lockheed Martin), suppliers (BAE Systems, Rockwell Collins, GE Aviation), government/certification agencies (FAA, NASA, DoD), and the SEI.

The SAVI approach is to drive the development and validation/verification process through a reference model in a standardized meta model, semantics, and interchange format in order to achieve

[17] For more information, see http:// www.artist-embedded.org.

[18] For more information, see http://www.opengroup.org.

earlier discovery of system-level problems in the embedded software systems. After evaluating several architecture modeling technologies, initiative members chose AADL as a key technology. A single architectural reference model expressed in AADL represents the source of the single truth for multiple dimensions of analysis, simulation, and generation.

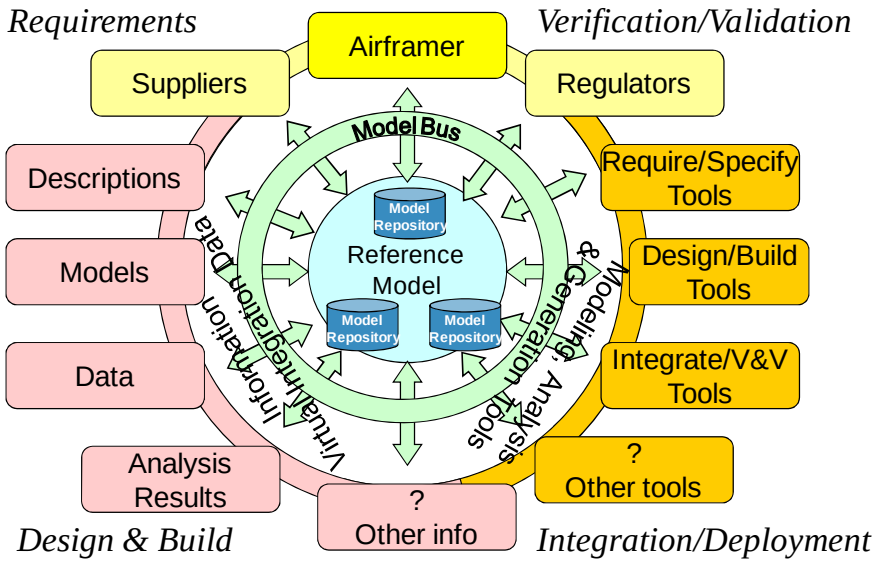

Figure 24: SAVI Approach

SAVI takes a multi-notation approach to the content of the model repository that is based on a semantically consistent meta model of its content—the reference model— thus minimizing model overlap. At the architecture level, this is achieved by expanding the meta model for AADL, which supports SysML® component modeling, to accommodate architectural aspects of computer hardware expressed in VHDL and of mechanical system models expressed in notations such as Modelica.® At the detailed design level, this modeling is complemented by detailed design notations, such as physical system and control dynamics expressed in Simulink,® or application code expressed in Ada, C/C++, or Java.

In Phase 1, a proof-of-concept (POC) demonstration was conducted to get buy-in from management of the member companies to fund the next phases of the initiative. The SAVI POC demonstrated that the SAVI concepts of a model repository and model bus support an architecture-centric, single-source representation. Auto-generation of analytical models interfacing with multiple analysis tools from annotated architecture models preserves single-truth analysis results. This was demonstrated through

- multi-tier modeling and analysis across system levels
- coverage of system engineering and embedded software system analysis
- propagation of changes across multiple analysis dimensions
- distributed team development via a repository to support airframe manufacturer and supplier interaction.

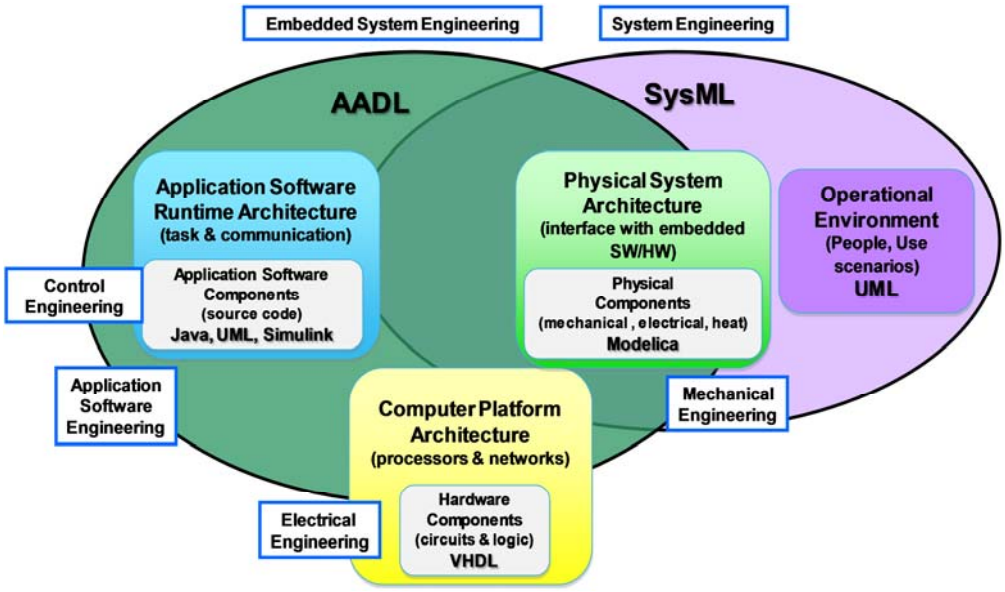

Figure 25: A Multi-Notation Approach to the SAVI Model Repository Content

Figure 26 shows some of the elements of this demonstration. At Tier 1, the focus was on the aircraft system and included analysis of weight and power. At Tier 2, the integrated modular avionics (IMA) portion of the aircraft architecture was expanded to represent the computer platform

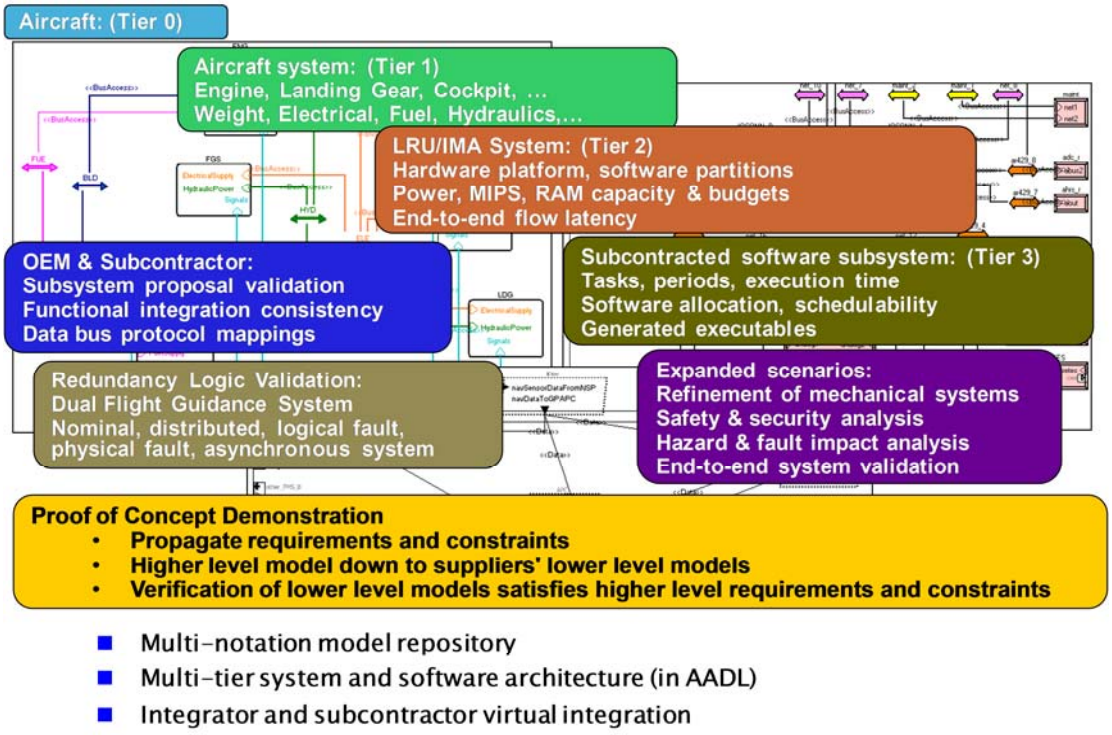

Figure 26: SAVI Proof-of-Concept Demonstration

and major embedded software subsystems in a partitioned architecture. At this time, the previous analyses were repeated on the refined model. Additional analyses included a first-level computer resource analysis and end-to-end latency analysis of two critical flows.

Next, the subcontracting negotiation phase between a system integrator and suppliers was demonstrated by providing AADL models as part of a request for proposals (RFP) and as part of submitted proposals. At that time, the system integrator virtually integrated the supplier models to ensure consistency with the system architecture and between the subsystems; this was demonstrated by the checking of functional integration and of data bus protocol mappings, in addition to revisiting earlier analyses in order to ensure that the analysis results still met the requirements. Two of the subcontractor subsystems were then elaborated into a task architecture and, in one case, populated with detailed design, Ada code, and a generated runtime system from AADL to analyze best allocation of computer resources including schedulability and actual testing of code. Finally, the detailed subsystem models were delivered to the integrator and integrated into the system architecture for system-level analysis, which revisits all previous analyses based on the refined models and data from actual code replacing the initial resource estimates such as execution time.

In 2010, the demonstration was expanded by

- refining the physical systems with mechanical models to better demonstrate the ability to perform virtual integration and analysis across system engineering and software engineering boundaries (also demonstrated in a paper by Bozzano and associates [Bozzano 2010]). In particular, the demonstration showed how a single AADL model could represent the interaction between a finite element model of the aircraft wing structure and a mechatronics actuator model.
- including a focus on safety and reliability requirements, supporting hazard, fault impact, and reliability analysis. In particular, the demonstration illustrated the ability to perform Functional Hazard Assessment (FHA), Failure Mode and Effect Analysis (FMEA), and Mean Time To Failure (MTTF) analysis from one set of Error Model Annex annotations to the AADL model of the aircraft, with a focus on applying these analyses to the embedded flight guidance system.
- applying static analysis such as model checking to validate the mode logic and other system and software behavior specifications early and throughout the development life cycle with the goal of demonstrating end-to-end system validation from requirements to implementation. In particular, the redundancy logic of the flight guidance system was validated under nominal conditions, as well as under the occurrence of several types of failures.

Architecture-centric modeling and analysis with AADL has also been applied to a reference architecture for autonomous space vehicles [Feiler 2009c] and to the evaluation of potential migration issues for an avionics system moving from a federated architecture to an IMA architecture [Feiler 2004]. Other examples include the use of model-based analysis with AADL in the context of an architecture evaluation using the ATAM and the development and piloting of a virtual upgrade validation method that incorporates techniques to address the root cause areas identified in Section 2.4 [DeNiz 2012].

3.3 Static Analysis

Static analysis is any technique that, prior to deployment and execution, mathematically proves system properties from the system description. As mentioned in Section 3.1, it's necessary to apply static analysis to benefit from formalizing requirements, since it enables verification of their completeness and consistency. Furthermore, unlike other validation methods such as testing and simulation, static analysis techniques apply throughout the development cycle: for validation of requirements, verification of design models against requirements, verification of implementation against the design model, and verification of code against common problems such as deadlocks, buffer overflows, and so on. Static analysis leads to early error detection and improves the development process by providing an end-to-end validation framework [Miller 2010, Bozzano 2010].

Many system properties are amenable to static analysis, such as performance, resource allocation, and ability to diagnose. However, for safety-critical systems, the most prominent form of static analysis is analysis of behavioral properties using model checking and abstract interpretation. Static analysis can establish that the system (or its model) satisfies its functional requirements, never enters its unsafe region, never produces a runtime error, and never deadlocks. Case studies show that model checking is more effective than testing for detecting subtle defects in the design [Miller 2010].

Scalability is the main bottleneck to a widespread use of formal methods-based static analysis in development of safety-critical systems. This is not surprising: establishing the safety of a system is a difficult problem. It is unlikely that static analysis techniques will ever scale fully to complex systems and displace other validation techniques such as testing and simulation. However, experience shows that scalability can be achieved by a combination of abstraction (i.e., extracting finite models of the system) [Clarke 1994], designing for verification [Groundwater 1995, Miller 2010], compositional verification [Clarke 1989, Grumberg 1994], and by developing domain-specific analysis tool chains [Groundwater 1995, Miller 2010].

We proceed with highlighting some of the state-of-the-art approaches to static analysis of discrete system behavior and static analysis of other system properties, and give examples of end-to-end validation systems based on these techniques.

3.3.1 Static Analysis of Discrete System Behavior

We divide discrete systems into two types: finite state (systems with finite control and data domains) and infinite state (systems with finite control but infinite data domain). Finite-state systems can be expressed as models (or programs) using only Boolean and enumerated data types. Infinite-state systems are models (or programs) that require unbounded (or continuous) state values. We describe static analyses techniques for both types of systems in turn.

Model-checking is the most prominent static analysis technique for finite-state systems [Clarke 1999]. A model checker takes as an input a finite description of the system and a property (or a requirement) formalized in temporal logic and decides whether the system satisfies the property. If the system does not satisfy the property, the model checker returns a counterexample—that is, an execution that violates the property. The ability to produce counterexamples makes model checkers effective debugging tools for both the system and the formalization of the requirement.

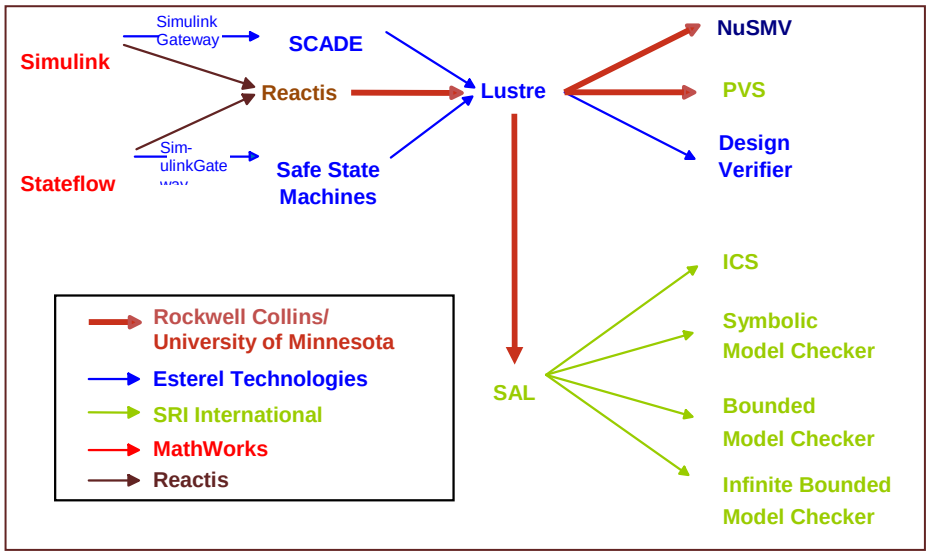

Figure 27: Rockwell Collins Translation Framework for Static Analysis

To be suitable for model checking, a model must describe the behavior of a system using only Boolean and enumerated types. Many industrial systems have components that satisfy this criterion or can be altered to satisfy it [Miller 2010]. In practice, the model is often generated from an architectural or other description of the system. For example, Rockwell Collins and the University of Minnesota have developed a translation framework to automatically construct models for model checking from Simulink and Stateflow modeling languages (see Figure 27). A similar approach, but based on AADL models, is taken by Noll and associates in the COMPASS project [Bozzano 2010, COMPASS 2011].

Properties must be formalized in temporal logic.

- Linear temporal logic (LTL) is used to specify a behavior of a system's individual execution. It extends propositional logic (that can express a single state of a system) with these temporal modalities:
  - **G**lobally, true in every state of the execution
  - **F**uture, true in some future state of the execution
  - **U**ntil, true from the current state until some condition (or event) is seen

  For example, a requirement "every request is eventually acknowledged" is formalized in LTL as

  ```
  G (req implies F ack).
  ```

  req and **ack** are Boolean state variables that indicate the generation of a request and receipt of an acknowledgement, respectively. The LTL expression states, "it is always the case that a request is followed by an acknowledgment sometime in the future."
- Computation tree logic (CTL) is used to specify a requirement with respect to all system behaviors at once. It extends the temporal modalities of LTL with universal (**A**, for all paths) and existential (**E**, exists a path) path quantifiers. For example, the above requirement "every request is eventually acknowledged" is formalized in CTL as

  ```
  EG(req implies AF ack).
  ```

The CTL expression states that "there *exists* a path on which a request is always acknowledged."

It is often inconvenient to formalize the requirements directly in temporal logic. Several alternatives are available. Whenever requirements are formalized using one of the formal frameworks described in Section 3.1.1, they can be converted into a suitable temporal logic automatically. Property patterns provide templates for many common requirement specifications [Dwyer 1999]. Safety properties can also be described by directly annotating the model with assertions or by embedding monitors.

A variety of industry-grade model checkers are available that have different strengths and weaknesses.

- NuSMV [Cimatti 2002] is a symbolic model checker[19] that is a *new (Nu)* implementation and extension of Symbolic Model Verification (SMV), the first model checker based on binary decision diagrams (BDDs). It was developed as a joint project between the Formal Methods group at the Istituto Trentino di Culture – Istituto per la Ricerca Scientifica e Tecnologica (ITC-IRST), the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at University of Genova, and the Mechanized Reasoning Group at University of Trento. NuSMV has been designed to be an open architecture for model checking that can be reliably used for the verification of industrial designs, as a core for custom verification tools, and as a testbed for formal verification techniques. It can also be applied to other research areas. NuSMV has been extensively used in verification of safety-critical systems. It is the main reasoning engine in the work of Miller and associates and the COMPASS project [Miller 2010, Bozzano 2010, COMPASS 2011]. It has been successfully used to verify systems with 10^{200} states [Miller 2010].
- Simple Promela Interpreter (SPIN) is an explicit-state model checker for verification of distributed systems [Holzmann 2003]. Its development began at Bell Labs in 1980 in the original UNIX group of the Computing Sciences Research Center. Spin provides a very rich modeling language called Promela. The supported features of the language include dynamic creation of concurrent processes and communication via synchronous and asynchronous messages. The SPIN model checker has been used extensively in verification of telecommunication protocols.
- The C Bounded Model Checker (CBMC) is aimed at static analysis of embedded software [Clarke 2004]. It supports models described in ANSI C and System C languages. It allows for verification for buffer overflows, pointer safety, and user-specified assertions. It can also be used to check ANSI C implementations for consistency with other languages such as Verilog. Unlike NuSMV and SPIN, the CBMC is incomplete since it examines only bounded executions of the system. However, it can be directly applied to the system's source code.

Static analysis of discrete infinite-state systems is much harder than analysis of finite-state systems. It is well known from the works of Church, Gödel, and Turing in the 1930s that verification of such systems is *undecidable*. Thus, techniques for static analysis of infinite systems are inherently incomplete. That is, they typically detect all possible errors but occasionally either indicate

[19] For more information, go to http://nusmv.fbk.eu/NuSMV/index.html.

an error that cannot really happen (a false alarm) or never terminate (in this case, the tool is typically stopped when resources are exhausted but before the analysis has completed).

ASTREÉ is a static program analyzer aimed at proving the absence of RunTime Errors (RTEs) in C programs [Cousot 2005]. It can analyze structured C programs that have complex memory usage but no dynamic allocation. It is targeting embedded programs in earth transportation, nuclear energy, medical instrumentation, aeronautic, and aerospace applications. It has been used to completely and automatically prove the absence of any RTE, in the primary flight control software of the Airbus A340 fly-by-wire system and to analyze the electric flight control codes for the A380 series.

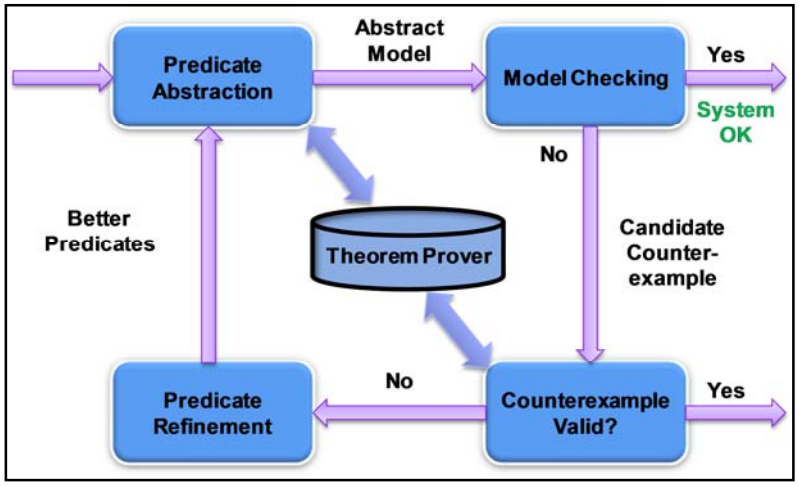

Figure 28: CounterExample-Guided Abstraction Refinement Framework

CounterExample-Guided Abstraction Refinement (CEGAR) is a technique pioneered at Carnegie Mellon University that extends finite-state model checkers to analyze infinite-state systems [Clarke 2003]. The technique uses an automated theorem prover (or an Satisfiability Modulo Theory (SMT)-solver) to automatically extract a finite-state abstraction of an infinite-state system (see Figure 28). The behaviors of the abstraction are a superset of the concrete behaviors. Thus, if the abstraction is shown to be safe by a model checker, the concrete system is safe as well. If the abstraction is unsafe, the counterexample generated by the model checker is used to either construct an unsafe execution of the concrete system or to automatically refine the abstraction. Many academic tools are available in this active area of research.

3.3.2 Static Analysis of Other System Properties

Static analysis is not limited to analyzing discrete system behavior. There are static analysis techniques for scheduling, resource allocation, and real-time, probabilistic, and hybrid control properties. Here, we highlight tools for verification of real-time and probabilistic properties.

In traditional model-checking, temporal properties are defined with respect to the temporal order of events. However, the actual passage of time between events is ignored. For example, it is possible to check whether a request is acknowledged but not whether a request is acknowledged within a given time bound, say 10ms. When real time is important, the system must be modeled with real-valued clocks.

Static analysis of such real-time systems can be done with the UPPAAL tool that supports modeling systems as a nondeterministic composition of timed automata [UPPAAL 2009]. It can then simulate and model-check properties of such systems. It uses symbolic techniques to reduce the state space exploration problem. It has been used in a variety of industrial case studies, including verification of an automobile gearbox controller.

For some properties, it is important to consider an inherent probabilistic nature of real systems. For example, it might be necessary to know the expected time for a request to be acknowledged or the expected power consumption. These questions can be answered by a *probabilistic model checker*, such as PRISM [Kwiatkowska 2010] or the Markov Reward Model Checker (MRMC) [Katoen 2009]. In this case, the system is modeled as a Markov model, and properties are expressed in temporal logic extended with probabilistic modalities.

3.3.3 End-to-End Validation

Static analysis enables a complete end-to-end validation framework. Formalized requirements can be validated for completeness (all system behaviors are considered), consistency (no requirement conflicts with another), and explored for possibility (which potential behaviors are compatible with requirements and which are not). This process can find flaws in the requirements before a detailed design is constructed. This, in turn, avoids the costs of fixing the requirements later during the design cycle and leads to faster qualification. Formalized detailed designs can be validated against the requirements. This process can find flaws in the designs (or, potentially, in the requirements) before implementation is completed. This avoids the cost of fixing the design and requirements during the implementation phase of the development cycle. Finally, the implementation can be validated against the detailed design. This facilitates more efficient discovery of implementation errors than testing and simulation alone [Miller 2010]. The overall framework promotes modularity, which is paramount for scalability of static analysis techniques.

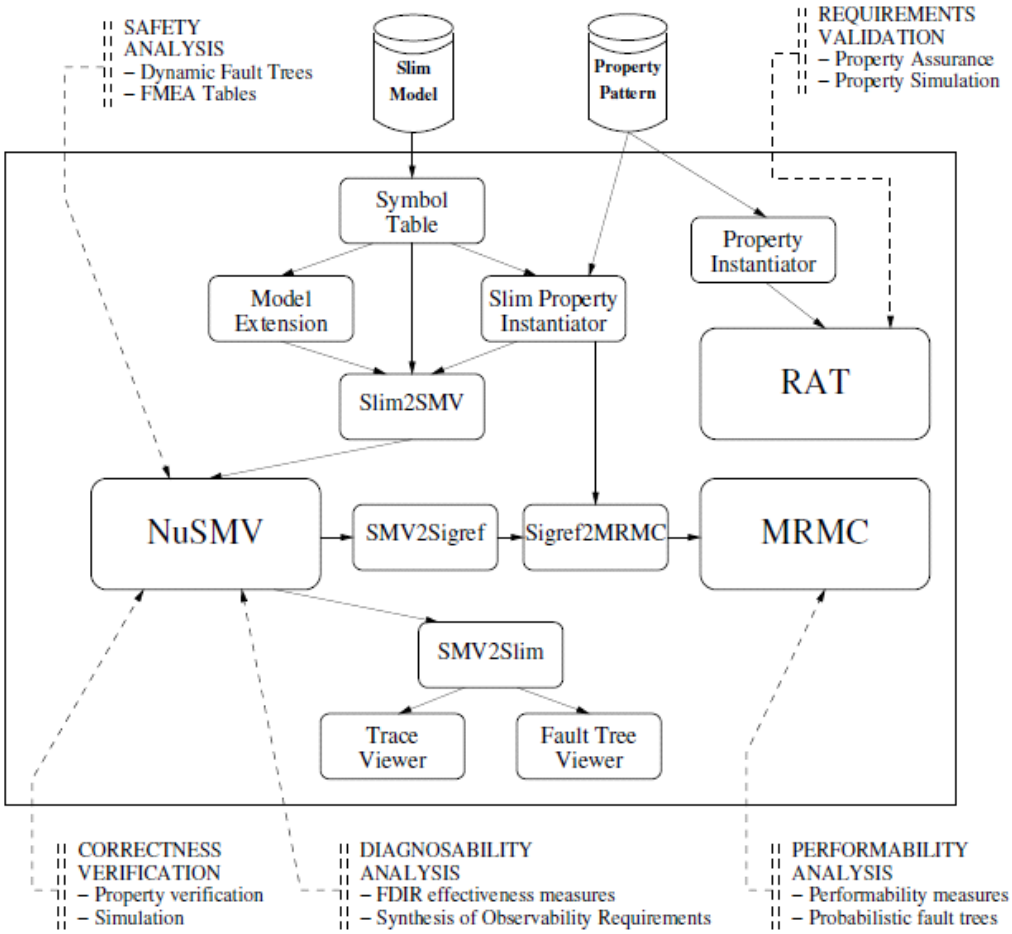

Figure 29: The Architecture of the COMPASS Toolset

The COMPASS project funded by the European Space Agency (ESA) is a good example of a toolset supporting end-to-end validation through static analysis (see Figure 29) [COMPASS 2011]. The framework is building on AADL and the AADL Error Model Annex. It provides tool support for the following analyses:

- formalizing requirements as specification patterns and parameterized templates. This facilitates capturing safety, correctness, performance, and reliability requirements.
- requirement validation through the Requirements Analysis Tool (RAT) [FBK 2009]. The requirements are converted automatically to a suitable temporal logic. The included analyses are consistency (checking that requirements do not contradict one another) and assertion checking (checking whether requirements satisfy a given assertion).
- functional validation through model checking using the NuSMV2 tool [Cimatti 2002]. This includes checking whether the design satisfies its requirements under nominal conditions and checking for the absence of deadlocks.
- safety analysis with traditional techniques including FTA and FMEA
- performance evaluation to compute system performance under degraded operations based on probabilistic inference using Markov Reward Model Checker (MRMC)

- Fault Detection, Isolation, and Recovery (FDIR) analyses to verify that the system can properly detect, isolate, and recover from a given fault.

3.4 Confidence in Qualification Through System Assurance

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

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

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

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

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

The goal-structured assurance case is similar. Affirming evidence is associated with a property of interest (e.g., safety), attesting that it fulfills its claim. For instance, test results might be collected into a report. Without an argument as to why the test results support the claim of safety, an interested party could have difficulty seeing its relevance or sufficiency. With only a detailed argument that depends on test results to show that a system was safe, but does not provide those re-

sults, again it would be hard to establish the system's safety. So a goal-structured assurance case as shown in Figure 30 specifies a claim (or goal) regarding a property of interest, evidence that supports that claim, and a detailed argument explaining how the evidence supports the claim.

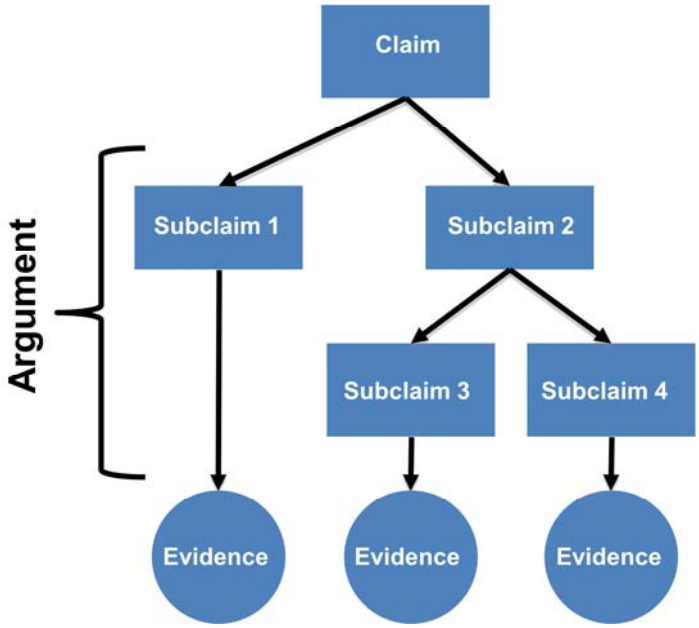

Figure 30: A Goal-Structured Assurance Case

The goal-structured assurance case [Kelly 2004] has been used extensively outside of the United States for a number of years to assure the safety[20] of nuclear reactors, railroad signaling systems, avionics systems, and so forth. Assurance cases are now being developed for other attributes (e.g., security of a software supply chain [Ellison 2008]) and other activities (e.g., acquisition [Blanchette 2009]).

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

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

- It can help determine which claims are most critical and, hence, what evidence and assurance-related activities are most needed to support such claims.
- It can help guide design decisions that will simplify the case and, thus, make it easier to develop a convincing argument that important properties have been met.
- It serves as documentation as to why certain design decisions have been made, making it easier to revisit these decisions should the need arise, helping to communicate engineering expertise, and allowing for more efficient reuse in subsequent systems.

[20] When used to show safety, an assurance case is called a *safety case*.

- It can help management make a more accurate determination of whether the development is on track to produce a system that meets its requirements.

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

3.4.1 Requirements and Claims

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

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

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

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

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

We can represent such a safety requirement as a claim in an assurance case together with the evidence and other arguments needed to show that the requirement is satisfied (see Figure 31). One set of evidence is the aforementioned testing results. However, we can increase confidence in the

validity of the claim by also showing that pitfalls to valid testing have been adequately mitigated. Going a step further, our confidence in the validity of the claim would also be increased by an argument asserting the accuracy of the algorithm used to estimate remaining battery life. The combination of testing results and algorithm analysis makes the case stronger than if just test results alone were presented. To support the algorithm-accuracy claim, a developer might reference design studies and literature, as well as an analysis of the actual design.

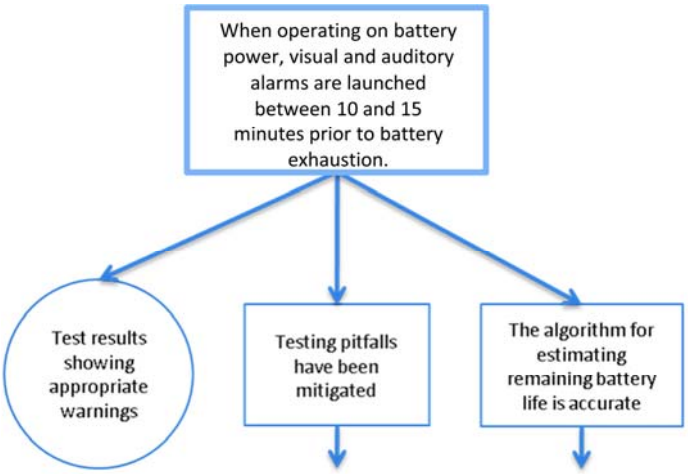

Figure 31: Confirming That a Safety Requirement Has Been Satisfied

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

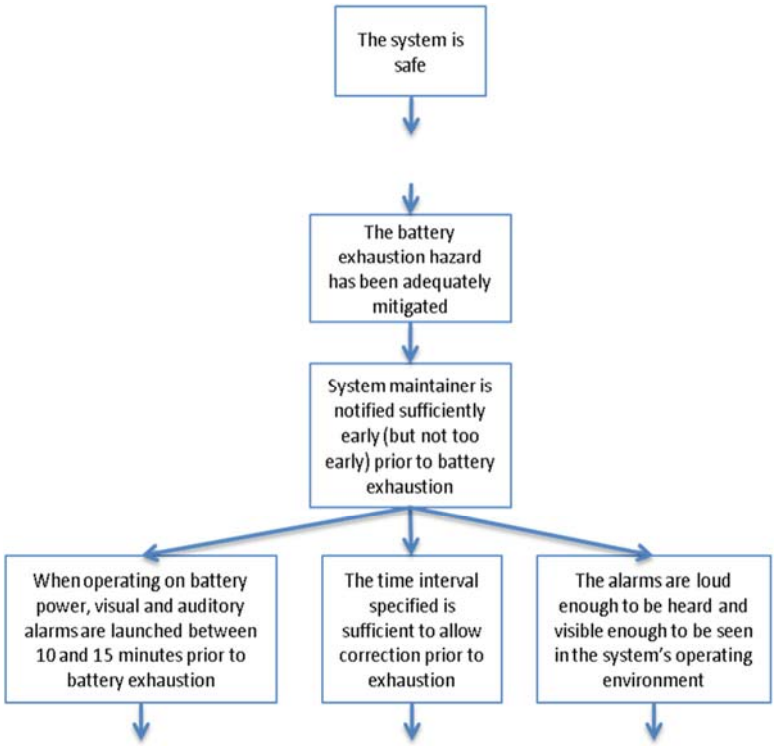

Figure 32: Context for Raising an Alarm About Impending Battery Exhaustion

Figure 32 presents

- claims related to a battery-exhaustion warning system and
- the context for such claims

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

The proposed mitigation for battery exhaustion is to notify a system maintainer in a timely manner that the battery is about to become exhausted. This is shown in the case by making the claim of notifying the maintainers "sufficiently soon" but not "too soon." We are now in a position to state the safety requirement about when an alarm needs to be raised. In addition, we can now readily deal with other hazards not addressed directly by the safety requirement; namely, we can consider whether the warning time is sufficient to allow human response and also whether the alarm is sufficiently noticeable that the human will be unlikely to ignore it.[21]

[21] The case supporting the "Alarm noticeability" claim could be fairly complex, since the total variety of alarms and indicators needs to be considered, as well as the fact that some alarms are more important than others. One could ask: Is the system safer if the auditory alarm is louder or more urgent when the remaining battery life is less than five minutes? Less than three? And what happens when there are competing alarms? Which one gets the highest alarm signal? Is the overall alarming strategy for the system consistent with user interface standards for alarm signaling? Will the alarm for an important condition be loud enough to be heard over competing

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

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

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

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

3.4.2 Assurance Over the Life Cycle

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

As an example of the above, consider a system that has a requirement to restart within one minute of a system failure. Early in the life cycle, designers are faced with making decisions on how to

---

alarms or the sounds of other equipment? All these issues can be raised and dealt with in the expansion of the "Alarm noticeability" claim.

best meet this requirement. Obvious choices include hot standby, warm restart, and cold restart. Each has its costs and benefits, and tradeoffs must be made.

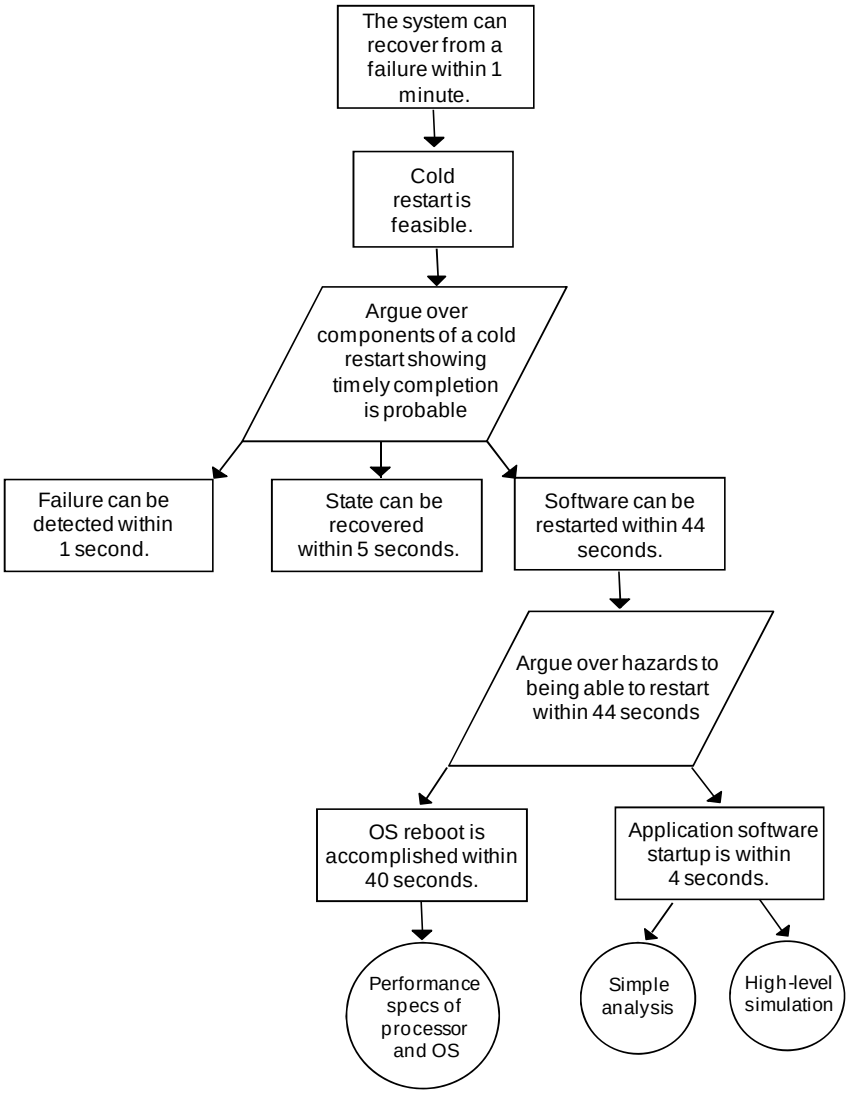

Figure 33: An Assurance Case Early in Design

Figure 33 shows a partial assurance case for such a design. Only the case for cold restart has been expanded because that alternative has proven, at this early stage, to likely be able to accomplish the goal of restarting within one minute. If there were any question about this goal being met, the other alternatives would have also been expanded to enable a more informed decision. Figure 34 shows an assurance case for the same system later in the life cycle. It is both simpler (the rejected alternatives have been removed from the case) and more complex (the analysis of the cold restart alternative is supported by additional evidence) than the case in Figure 33. As the system is developed further, the case is augmented with actual test results as evidence, as well as more details—simulations, AADL models, and so on.

We expect to see much more detailed evidence as the development of the system continues along the V that we discussed in Section 1.1. Thus it is important to develop and maintain the assurance case in parallel with the system being assured. This has a multitude of benefits, including

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

4 A Metric Framework for Cost-Effective Reliability Validation and Improvement

The objective of this metric framework is to drive the cost-effectiveness of a validation and reliability improvement strategy for system qualification. We address this objective from two perspectives:

1. by focusing on high-risk areas that introduce faults into a system, we can see those areas in the system that require more attention to reduce the introduction of faults
2. by accounting for the effectiveness of validation methods at different times in the life cycle, we can understand the effectiveness of methods to discover and remove faults throughout the development life cycle.

This allows us to reduce development and qualification cost by avoiding rework and retest through early discovery of faults. It also allows us to focus the validation resources on those parts of the system and validation methods that most cost-effectively minimize residual system faults with acceptable risk. We proceed by considering

- architectural metrics that reflect the potential of system-level faults
- qualification-evidence metrics based on assurance cases that reflect sufficient evidence and acceptable risk for the absence of faults in the qualification of a system
- metrics that reflect cost and cost savings in system development and validation.

4.1 Architecture-Centric Coverage Metrics

Traditional reliability engineering has focused on fault density and reliability growth as key metrics. These are statistical process metrics that reflect the presence of faults by tracking discovery of faults during review and testing, as well as failures during operation. This has worked well for slowly evolving physical systems where the emphasis is on failure of physical components rather than fault in the design. Such metrics have had limited success with software systems because software faults are design faults. Furthermore, as software is frequently changed and evolved, there is additional design fault potential.

Review and testing has been the primary approach for addressing faults. Sequential code faults are activated by certain execution paths operating on given data, which can be addressed by test coverage. For systems with concurrent processing and sharing of resources, the combinations of possible interactions grow exponentially, and faults such as race conditions are difficult to test for. Especially for embedded software systems, the operational environment affects the behavior of the system. A change in operational context may cause the system to behave in a way that activates a previously latent fault.

We propose three metrics that aid in addressing faults introduced during the system design and do so early in the life cycle: (1) one focusing on requirements coverage, (2) one focusing on safety hazard coverage, and (3) one focusing on architecture-level system interaction coverage.

4.1.1 A Requirements Coverage Metric

Requirement specifications are a key artifact in the development process, since systems are validated against their requirements. As we have seen in Sections 2.2 and 2.3, requirement errors are major contributors to system-level problems that are currently discovered late in the development process. Requirement specifications are often incomplete/ambiguous and incorrect/inconsistent. High-level requirements, which may be difficult to validate in themselves, are refined into concrete requirements for which qualification evidence can be provided in form of analysis, simulation, or testing. Requirements for a system must be specified with respect to its external interactions. Furthermore, requirements at the system level must be decomposed into requirements placed on system components. This decomposition must be done across the system architecture to include requirement specifications beyond functional requirements on the embedded software system.

We define a requirements coverage metric with several contributors. We base this definition on the assumption that requirements are associated with elements of a system architecture—in other words, that requirements can be traced to specific system components.

The first contributor reflects the coverage of all interaction points of a system or system component with its context. We can measure coverage of the requirement specification with respect to its input, output, resource demands, and control, as well as its state and behavior (see Figure 12 in Section 3.1.2). For input/output interactions, the requirement specification must address domain data type, expected value range, measurement units, data stream characteristics (such as rate, latency, ordering, and value changes), and implied resource demand. For control interactions the requirement specification must address action request and responses (including expected ordering of actions). The system (component) behavior must be characterized in terms of discrete states (set of behavioral states and transitions between states, as well as continuous value state spaces—often expressed as equations). A requirement specification must address resource demands in terms of types of resources; demand amount, such as worst-case execution time; rate of demand, such as the period of a task; and time frame in which the resource must be available, such as deadline. The requirement specification must not only address nominal mission operation, but also safety-criticality aspects, such as the ability to address safety hazards.

A second contributor to the requirements coverage metric is the degree of decomposition of requirements into requirements on system components. This contributor (1) tracks the degree to which requirements at one level of the system hierarchy are addressed by requirements of its components and (2) reflects whether satisfaction of component requirements is a necessary or sufficient condition to satisfy the system requirement.

A third contributor is the consistency of the requirement specification. This includes consistency and correctness between elements of a requirement specification of a system or component, as well as between the requirement specification of system components and that of the containing system. In other words, this contributor reflects the set of constraints that requirement specifications satisfy. Examples of such constraints are (1) the reachability of states in a behavioral state description, (2) the consistency between input/output requirement specification of the system and those of its components, (3) the relationships among the processing rate of a task, the intended sampling rate of input, and the arrival rate of data streams, (4) the relationship between resource

budgets of components and those of the system, and (5) the traceability of hazards in a FHA to failure modes in an FMEA.

4.1.2 A Safety Hazard Coverage Metric

The part of the system addressing safety and reliability makes up a considerable portion of the system functionality, and its robustness to hazards is critical to system operation. Assuring that safety hazards are being addressed is critical; in particular we need a better understanding of the hazard contributions by the embedded software systems. We define a safety hazard coverage measure with several contributors.

The first contributor reflects how complete the hazard specifications are, that is, how well the specification covers a known set of hazards. The hazard coverage count tracks for the system and its components how many hazard specifications are documented for each of the interaction points (input/output, control, resource usage) of the requirement specification and compares it to the known hazard count. A hazard specification indicates whether an error is being propagated out, is expected as an incoming hazard, or is expected to be contained (masked) by the originator (completeness of specification).

For embedded software hazards we leverage fault containment mechanisms, such as the use of dedicated processors, runtime-enforced protected address spaces, and virtual machines/partitions. Hazard and error propagation between software units in different fault containment units can be limited to their interactions in terms of input/output, control, and shared resource usage. Faults, whether they are design or coding errors inherent in the software component or the result of error propagation from other software or from hardware, manifest themselves as interaction errors. As discussed in Section 3.1.4, we have a known set of potential hazards for data streams, control interaction, and resource usage.

The second contributor reflects consistency between hazard specifications of interacting components. The hazard consistency count tracks how many interactions in the form of connections and deployment bindings to platform resources have an inconsistent set of hazard specifications for their endpoints. The hazard specifications of two interaction endpoints are inconsistent if the recipient expects the potential hazards to be masked while the originator intends to propagate them.

The third contributor reflects the impact potential on high-criticality components. This high-criticality impact count tracks the number of error propagation channels (interaction points and deployment mappings) from lower criticality components to higher criticality components, as well as the number of intended and unintended error propagations on each channel. A higher count represents a higher safety risk.

The fourth contributor reflects the robustness of the system and its components to unintended hazard propagation, that is, propagation of errors due to fault activation within a component that were not intended to be propagated according to the hazard specification. For this count, focus is on the ability of error propagation recipients to tolerate propagations that they expected to be masked. The count tracks the number of outgoing masked hazard specifications for which the recipient specification indicates expected masking, rather than expected incoming error propagation.

4.1.3 A System Interaction Coverage Metric

The quality of a system implementation is strongly affected by architectural decisions. In Section 2.4, we identified root cause areas for introducing system-level faults due to interactions among embedded software system components, their interactions with the physical system and environment, and their interactions with the underlying computer platform. Therefore, we complement coverage metrics for software code, such as Decision Coverage (DC) and Modified Condition/Decision Coverage (MC/DC) found in DO-178B [RTCA 1992], with coverage metrics that focus on architecture-level system interactions.

System interactions can introduce architectural design complexity. The objective of the system interaction coverage metric is to capture several contributors to this complexity.

The first contributor tracks different peer-to-peer architecture interaction patterns, such as a pipeline, service request/response, or a control feedback loop (see Section 3.2.3), in a system. Many such interactions are between subsystems that maintain state. Each participant in the pattern has its own state machine with transitions and makes assumptions about the state of the other participants. Work by Miller has shown that specification of interactions between two small state machines with transitions is prone to error [Miller 2005a]. Therefore, the size of the state machines, transition coverage, and reachability of states within each subsystem, as well as their interactions, provide a good measure of the system's interaction complexity.

The second contributor focuses on the operation modes of the system and its subsystems. Operational modes represent operational states in which a system or system component exhibits a particular behavior. Larger embedded systems have operational modes at the system level to reflect mission operation. These are supported by operational modes of various subsystems, which themselves may make use of subsystems with operational modes. This support requires a coordination of operational mode states across the architectural hierarchy. We have a measure of mode state and transition coverage and consistency between the system's and subsystems' operational modes.

The third contributor focuses on the fault management portion of the system architecture, that is, on redundancy patterns and the recovery of faults, as well as the traceability between the identified hazards and their expected mitigation and the fault management mechanisms in the actual system. We have measures of the

- complexity of the redundancy management logic for each encountered redundancy pattern, which may operate in a distributed setting
- complexity in coordinating fault recovery across different subsystems, which is similar to the coordination of operational modes
- traceability coverage between safety and reliability hazards and their management in the safety-critical portion of the system

4.2 Qualification-Evidence Metrics

Standards for safety-critical systems such as DO-178B focus on specifying qualification criteria. Criticality levels are identified for different system components, and qualification criteria are assigned to each—a larger set and more stringent criteria for higher criticality levels. The criticality level is determined by (1) the cause of a software component's anomalous behavior or (2) that

behavior's contribution to the failure of a system function that results in system hazards and failure conditions of varying severity. Five severity levels combined with likelihood of occurrence (expressed qualitatively as likeliness of occurrence or quantitatively as probability of occurrence) act as a system safety management decision matrix [Boydston 2009].

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

These criteria can be mapped into an assurance case framework, and we will use such a framework to drive a qualification-evidence metric. This is conceptually illustrated in Figure 34. Claims represent qualification criteria on the system and its subsystems, that is, requirements that must be satisfied by the system design and implementation as shown by the evidence. The operational context and the assumptions for each claim are documented. The evidence takes the form of a V&V activity, ranging from review and testing to formal analysis. The process of producing the evidence has its own set of claims and evidence, such as validity of the model or test case implementation and correct application of the evidence-generating method. The risk-level annotations reflect the criticality levels of different subsystems and different requirements on those subsystems. Whether the evidence for meeting the qualification criteria is sufficient is reflected in the argument and represents a risk assessment by the qualification authority.

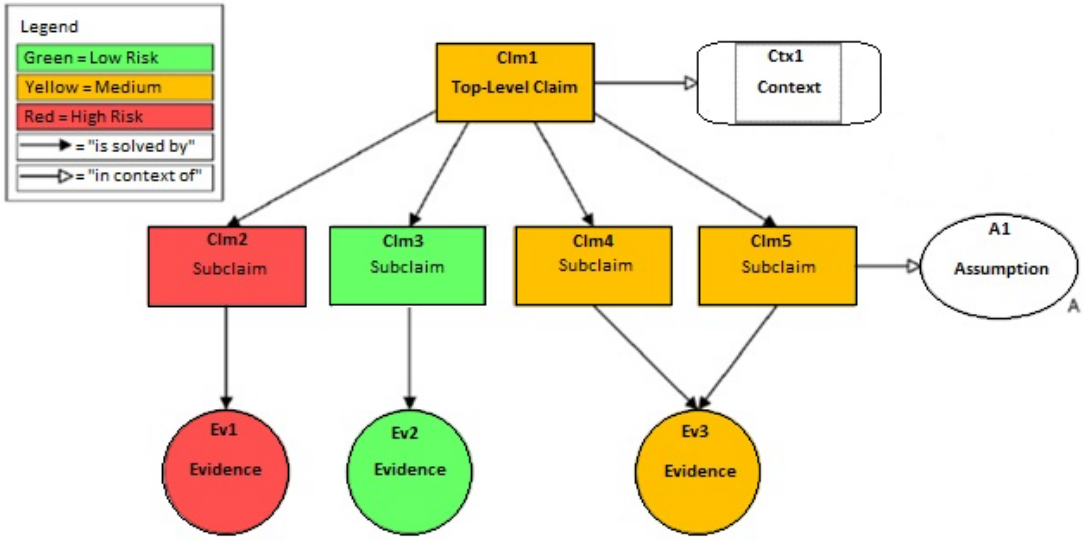

Figure 34: Qualification Evidence Through Assurance Cases

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

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

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

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

4.3 A Cost-Effectiveness Metric for Reliability Improvement

The objective of this metric is to determine cost savings from the application of methods for early error detection. Such methods result in avoidance of certain rework and reduction of retest cycles, thus reducing error leakage and increasing our confidence in the qualification evidence. For this metric, we can draw on two pieces of work: the AVSI SAVI return-on-investment (ROI) study [Ward 2011] and the COnstructive QUALity MOdel (COQUALMO) work [Madachy 2010]. Both these efforts draw on a taxonomy and on results of a NASA study by Hayes [Hayes 2003].

The SAVI ROI study approaches the problem of estimating cost savings due to rework and retest avoidance by comparing error introduction and removal percentages in the current development practice (shown in the aggregate on page Figure 3 on page 7)) against the architecture-centric model-based virtual integration approach of SAVI. When we take the normalized rework and retest cost factors shown in Table 1 on page 8, and apply them to the error percentages introduced in early phases (requirements and design) and detected in late phases (integration, system and acceptance testing), we see that rework/retest cost due to requirement and design errors dominates the total rework/retest cost.

Table 2 shows the cost to remove a defect of a given type relative to the total cost of defect removal. For example, requirements defects account for 79% of rework cost and 62% of rework cost occur during integration.

Table 2: Relative Defect Removal Cost (as Percent)

	Phase in Which Defect is Removed					
Defect Type	**Requirements**	**Design**	**Code**	**Test**	**Integration**	***Sum***
Requirements	0.03%	0.21%	1.87%	28.11%	48.73%	*78.97%*
Design		0.04%	0.37%	5.79%	9.75%	*15.96%*
Code			0.19%	1.28%	3.10%	*4.57%*
Test				0.17%	0.26%	*0.43%*
Integration					0.09%	*0.09%*
Sum	*0.03%*	*0.26%*	*2.44%*	*35.36%*	*61.92%*	*100.00%*

By estimating the ability of a SAVI approach to discover errors in early phases, possibly in-phase with the introduction, we can determine the percentage of rework/retest cost that can be avoided. Representatives from eight AVSI SAVI member companies provided their estimates of possible in-phase detection for different categories of requirements errors based on their in-house pilot experiences with SAVI technologies. We used the resulting figure, 66%, and as a conservative alternate, 33%, in the ROI calculation. We then calculated the rework/retest avoidance cost savings according to the following formula:

*Cost avoidance = Estimated total development cost * % Rework cost * % Requirements error rework cost * % Requirements error removal efficiency*

We calculated the estimated total development cost with the COnstructive COst MOdel (COCOMO) II [Boehm 2000] using software SLOC size estimates from commercial aircraft companies, assuming a distributed integrator/supplier development environment with commonly encountered code reuse percentage. The system development cost was extrapolated from the software cost based on industry figures that software in aircraft systems makes up two-thirds of the total system cost. Again based on industry experience, a rework/retest cost percentage (*% Rework cost*) of 30% and 50% were chosen. The *% Requirements error rework cost* figure was taken from Table 2, and for the *% Requirements error removal efficiency* rate the above mentioned estimates were used. The cost savings shown, even for the most conservative estimate, savings considerably greater than the investment for a single aircraft development. We compared and confirmed these estimates with member company internal estimates.

In addition to the cost savings, the ROI study calculated the arithmetic and logarithmic (ROI) as well as the Net Present Value (NPV) based on an investment in the SAVI technology infrastructure of $80 million.

COQUALMO was developed as an extension to COCOMO for predicting the number of residual defects in a system and the impact of a schedule change to the cost and quality of the system. It is being further extended to help identify strategies for reducing defects by quantifying the impact of different processes and technologies [Madachy 2010]. The COQUALMO extension adds defect introduction rates for requirements, design, and code defects, and defect removal rates for different removal methods (see Figure 35). The defect categories for requirements are correctness, completeness, consistency, and ambiguity/testability. For design/code, the categories are interface, timing, class/object/function, method/logic/algorithm, data values/initialization, and checking. The removal methods (peer review, automated analysis, and execution testing and tools) are categorized from very low to extra high (e.g., under automated analysis from simple compiler

syntax checking to formalized specification and verification through model checking and symbolic execution). The model has been calibrated with industry data and applied in various settings. A continuous simulation model version has been used to evaluate the effectiveness of different removal methods at different times in the life cycle.

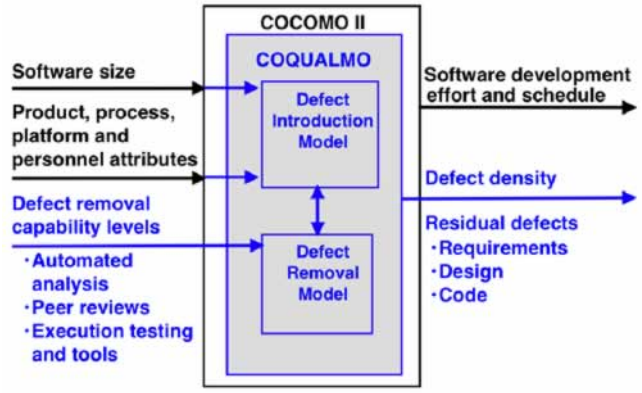

Figure 35: COQUALMO Extension to COCOMO

We propose to investigate the adaptation and possible integration of the two models above to the reliability validation and improvement framework. In this context we intend to elaborate the categories of defect removal capability to cover the full range of evidence-producing methods in an assurance-based qualification process. In particular, we can consider the effectiveness of independent system integration labs and virtual system integration labs in reducing defects. A virtual system integration lab uses a SAVI-like architecture-centric virtual integration approach to evaluate and validate system-level requirements as early as possible in the development life cycle. We also intend to evaluate the defect categories to see how well they capture the challenges discussed in this report (e.g., the issue of multiple truths through model inconsistency).

Boehm discusses the use of different risk minimization techniques with the COQUALMO, using defect removal as the primary risk reduction measure [Madachy 2010]. Our proposed supporting metrics refine such risk minimization by targeting specific system areas, taking advantage of architectural knowledge. Furthermore, we take into account both defect removal and management of faults and hazards, an important element of safety-critical systems. We can consider the possibility of extending the COQUALMO to take these extensions into account.

5 Roadmap Forward

The aircraft and space industry in the U.S. and Europe has recognized the shortcomings of "build then test" for their safety-critical, software-reliant systems and has pursued a SAVI solution using AADL [SAE 2004-2012] in an international initiative (AVSI). The objective of this initiative is to mature and put into place a practice infrastructure to support SAVI in terms of standards-based methods and tools as discussed in Section 3.2.4. This maturation is accomplished through multiple phases, each increasing the technical readiness level (TRL) as illustrated in Figure 36. The first phase, shown as the proof-of-concept loop, was completed in 2009. It included a proof-of-concept demonstration (see Section 3.2.4) and an ROI study (see Section 4.3) [Ward 2011].

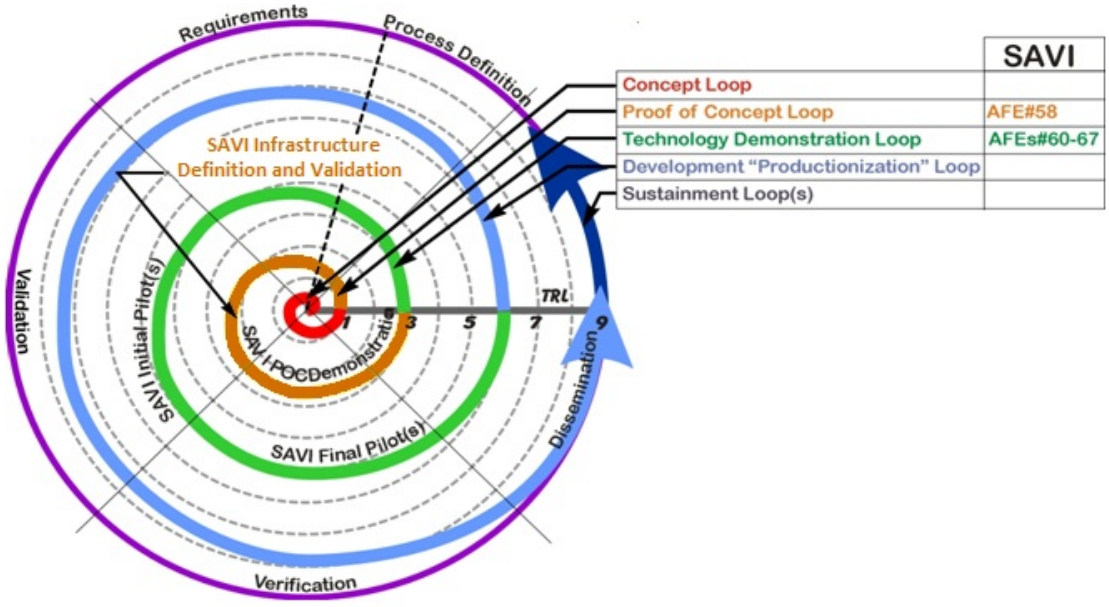

Figure 36: Multi-Phase SAVI Maturation Through TRLs

Phase 2 includes an extended POC demonstration emphasizing the

- integration between system engineering and software engineering
- definition of model repository and model bus requirements
- identification of technology gaps in the SAVI framework, and
- engagement of commercial tool vendors.

Additional phases extend into 2016 (shown in Figure 37) [Redman 2010].

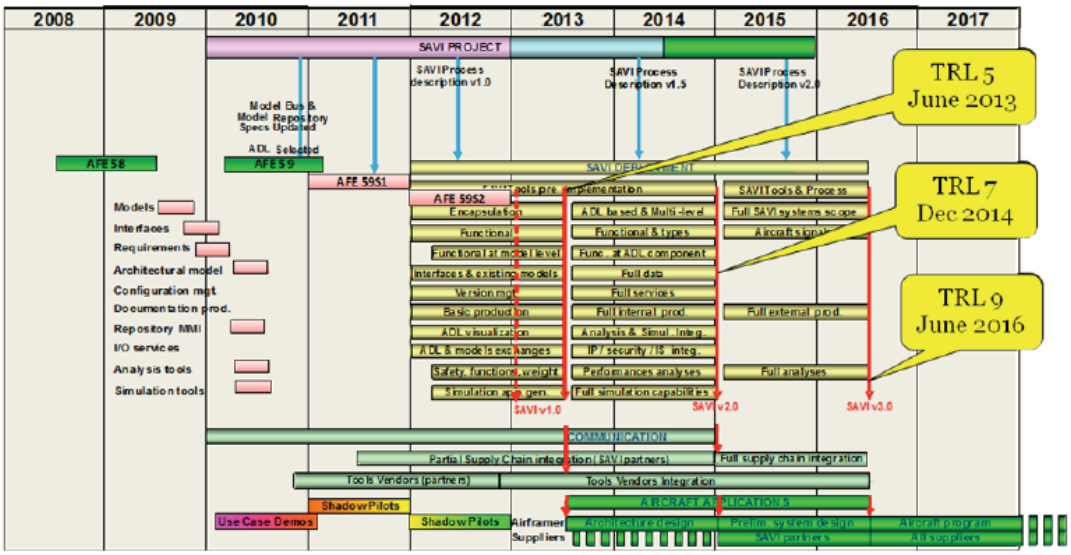

Figure 37: Multi-Year, Multi-Phase SAVI Project Plans

The objectives of the SAVI initiative coincide with the objectives of the reliability validation and improvement framework, with the latter placing more focus on the qualification of software-reliant systems. The Army, already an active member of the SAVI initiative, is positioned to take a leadership role in familiarizing program offices with this technology and in extending and adapting it to the reliability improvement and qualification focus of the DoD. We recommend a set of actions that will advance the practice of engineering safety-critical software-reliant systems—rotorcraft, aircraft, missile systems, automotive systems, autonomous systems—through a reliability improvement program focused on embedded software. We propose actions that will integrate and mature the four technology areas discussed in Section 3 and facilitate the adoption of this reliability improvement and qualification practice.

5.1 Integration and Maturation of Reliability Validation and Improvement Technologies

In its Phase 2, the SAVI initiative has placed a focus on reliability and safety. There is an opportunity to pursue the

- integration of formalized requirements (e.g., as documented in the *Requirements Engineering Management Handbook* [FAA 2009a]) and best model-based safety analysis practices (e.g., the systems approach to safety by Leveson [Leveson 2009]), as an extension of the SAVI framework from two perspectives: (1) mapping the requirements and practices into an architectural model and (2) developing assurance case patterns around safety analysis with an increased focus on embedded-software-related hazards. This integration requires
  - refining the Error Annex of the SAE AADL standard to better accommodate the semantics of software-specific faults and safety-criticality hazards and their mitigation, such as those found in FHA, FMEA, and STPA
  - developing a safety-criticality requirement engineering methodology driven by safety analysis and focused on the system and embedded software architecture as discussed in Section 3.1
  - developing a library of assurance case patterns reflecting the necessary evidence to build a strong assurance case. This methodology will be piloted in the context of the SAVI ini-

tiative to expose it to the aerospace industry and in an Army program to gain experience within the Army.

- expansion of system integration labs (SILs) into virtual system integration labs (VSILs) and determining their value-added over current testing practices in an assurance-based qualification practice. In a VSIL, the system architecture, its component design, and their implementation may be represented by models that are statically analyzed, simulated, or executed on a simulated platform. This change allows Army labs to establish an architecture-centric model-based independent validation and qualification practice, throughout the development life cycle, separate from DoD contractors' adoption of an architecture-centric, model-based development practice. This expansion requires
  - evaluating existing SILs (as well as the proposed VSILs) by identifying problem categories they can discover early now and problem categories that currently leak to later phases but could be addressed earlier by improved capability of a SIL or VSIL
  - piloting the concept of a VSIL in an Army lab on an actual system to validate the ability of VSILs to discover certain problem categories
  - developing an error-leakage-reduction-driven ROI model that predicts cost savings achieved in qualification through rework and retest cost avoidance using VSILs and value-added SILs. As outlined in Section 4.3, this can be achieved by adapting the ROI model developed under SAVI Phase 1 [Ward 2011] and incorporating aspects of the COQUALMO [Madachy 2010], as well as calibrating it with Army-specific data.
- establishment of a cost-effectiveness metric for reliability validation and improvement as outlined in Section 4. This involves
  - studying the cost and effectiveness of different development and validation methods in the development and qualification life cycle
  - calibrating the model with Army-specific data
  - applying a resource allocation strategy that maximizes reduction of error leakage and minimizes risk, by focusing on high-risk areas and taking into account the criticality of the system components
- development of an end-to-end, assurance-based qualification methodology and its piloting in an Army program. This involves
  - expanding assurance case patterns to cover the full development life cycle
  - reflecting in the argumentation aspect of assurance cases, the risk of not providing sufficient or sufficiently qualified evidence. Such an assurance case framework can then become the basis for a metric of sufficient evidence that allows qualifiers to quantitatively assess the residual risk in qualifying software-reliant systems as outlined in Section 4.2.

- proactive initiation by the Army, in coordination with the other services, of investigations into
  - the impact of new technologies and paradigm shifts (such as the use of multi-core technology and the migration to mixed-criticality systems) on the safety criticality of systems and existing analysis methods
  - reducing the potential negative reliability impact by putting a new analysis framework into place

The SEI continues its work value-driven incremental development to provide the architectural and assurance foundations necessary to make incremental development viable in the DoD. The SEI also continues be involved in the SAVI initiative to advance an architecture-centric virtual integration practice. The Army has an opportunity to leverage both the SAVI initiative and the SEI's investment in these technology maturation activities.

5.2 Adoption of Reliability Improvement and Qualification Practice

Adoption of this architecture-centric, model-based reliability validation and improvement practice is a process that can benefit from several activities driven by the Army:

- The Army can make changes to its acquisition language. The SEI has experience in cooperating with the DoD in revising such language to encourage the use of these technologies (and requiring them as appropriate) while they are maturing.
- The Army can benefit from gaining experience with the use of the technologies to better understand their benefits and limitations. This can be achieved through a series of well-coordinated pilot projects that will introduce the technologies incrementally in existing systems and their upgrades, as well as new system developments. Pilots allow the Army to present a strong argument to contractors who might resist the adoption of new technologies. Examples of incrementally introduced technologies include (1) a model-based variant of an architecture evaluation using the ATAM and (2) assurance cases on high-risk aspects of a system or system upgrade. Such pilots can be patterned after the
  - Common Avionics Architecture System (CAAS) evaluation [Feiler 2004]
  - case study of NASA Jet Propulsion Laboratory's Mission Data System reference architecture [Feiler 2009c]
  - comparative modeling case study of six CAAS-based helicopter systems
  - an AADL model supported architecture evaluation of the Apache Block 3 upgrade using the ATAM
  - the application of the Virtual Upgrade Validation (VUV) method [DeNiz 2012] to evaluate the migration of Apache to the Block 3 platform and a ARINC653-compliant platform

  Completion of the pilots could make way for development of a handbook for safety engineering for software-reliant systems, patterned after the *Requirements Engineering Handbook* [FAA 2009a]. Similarly, experience with reliability metrics for the software-reliant aspect of systems would be useful toward developing an addendum to the *Software-in-Systems Reliability Handbook* [DoD 2010]. Finally, a handbook on assurance-based qualification should be developed.

- The Army can ensure that its concurrence with the results of SAVI activities is recognized. The SAVI initiative will promote model representation and model interchange standards in support of the SAVI engineering framework, and can extend these efforts to include assurance-related technology standards. In addition, practice and process standards for safety-critical systems are currently under revision. For example, DO-178C is in the process of being finalized and includes sections on object technology, model-based engineering, and use of formal methods. The SAE S8 committee released a revision of SAE ARP 4754 [SAE 2010] and is currently revising SAE ARP 4761 [SAE 1996].

6 Conclusion

Rotorcraft and other military and commercial aircraft frequently undergo migration from federated systems to IMA architectures and experience exponential growth in on-board software size and complexity. This is due to increasing reliance on complex and highly integrated hardware and software systems for safe and successful mission operation. Current industrial practice of "build then test" has resulted in increasing error leakage to system integration test and later phases—rapidly increasing cost and reducing confidence in purely test-based qualification.

Reliability engineering has its roots in hardware reliability assessment that uses historical data of slowly evolving system designs. Hardware reliability is a function of time, accounting for the wear of mechanical parts. In contrast, software reliability is primarily driven by design defects—resulting in a failure distribution curve that differs from the *bathtub* curve common for physical systems. Furthermore, when assessing the reliability of a system, engineers often assume software to be perfect and to behave deterministically—that is, to produce the same result given the same input or to predict fault occurrence based on the size and branching complexity of source code. Therefore, the focus in software development has been on testing to discover and remove bugs using various test coverage metrics to determine test sufficiency. However, embedded software is time sensitive and implemented as a concurrent set of tasks, leading to nondeterministically occurring race conditions, unexpected latency jitter, and unanticipated resource contention. The source-code-based reliability growth metrics are not a good measure of such system-level interaction complexity.

The high cost of recertifying software-reliant systems, required for acceptance of software changes, has resulted in use of operational work-arounds rather than software fixes to address software-related problems, due to a less stringent approval process for these work-arounds. As a result, operators on some systems spend up to 75% of their time performing work-around activities. In other words, there is a clear need for improvements in recertification.

There is also a need for a reliability improvement program of these software-reliant systems; it must aim to overcome the limitations of current reliability engineering approaches, by integrating best emerging technologies that have shown promise in industrial application. Several studies in the U.S. and Europe have identified four key technologies in addressing these challenges:

1. specification of system and software requirements in terms of both a mission-critical system perspective (function, behavior, performance) and safety-critical system perspective (reliability, safety, security) in the context of a system architecture to allow for completeness and consistency checking

2. architecture-centric, model-based engineering, using an architecture model representation with well-defined semantics, to characterize the system and software architectures in terms of interactions between the physical system, the computer system, and the embedded software system. When annotated with analysis-specific information, the model becomes the primary source for incremental validation with consistency along multiple analysis dimensions through virtual integration.

3. use of static analysis in the form of formal methods to complement testing and simulation as evidence of meeting mission requirements and safety-criticality requirements. Analysis results can validate the completeness and consistency of system requirements, architectural designs, detailed designs, and implementation, and ensure that requirements and design constraints are met early and throughout the life cycle.

4. use of assurance cases throughout the development life cycle of the system and software 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 context and assumptions.

A number of initiatives in the U.S., Europe, and Japan are integrating and maturing these technologies into an improved safety-critical software-reliant system engineering practice. In particular, the SAVI initiative, an international Aerospace industry effort, offers an opportunity of leveraged cooperation as outlined in Section 5, “Roadmap Forward.”

Applied throughout the life cycle, reliability validation and improvement lead to an end-to-end V&V approach. This builds the argument and evidence for sufficient confidence in our system throughout the life cycle, concurrent with the development. The framework keeps engineering efforts focused on high-risk areas of the system architecture and does so in a cost-saving manner through early discovery of system-level problems and resulting rework avoidance [Feiler 2010]. From a qualification perspective, the assurance evidence is collected throughout the development life cycle in the form of formal analysis of the architecture and design, combined with testing the implementation.

The architecture-centric framework provides a basis for a reliability validation and improvement program of software-reliant systems [Goodenough 2010]. Building software-reliant systems through an architecture-centric, model-based analysis of requirements and designs allows for the discovery of system-level errors early in the life cycle.

The framework also provides the basis for a set of metrics that can drive cost-effective reliability validation and improvement. These metrics address shortcomings in statistical fault density and reliability growth metrics when applied to software. They are architecture-centric metrics that focus on a major source of system-level faults: namely, requirements, system hazards, and architectural system interactions. They are complemented by a qualification-evidence metric that is based on assurance case structures, leverages the DO-178B model of qualification criteria of different stringency for different criticality levels, and takes into account the effectiveness of different evidence-producing validation methods.

The effects of acting on this early discovery are reduced error leakage rates to later development phases (e.g., residual defect prediction through the COQUALMO [Madachy 2010]) and major system cost savings through rework and retest avoidance (e.g., as demonstrated by the SAVI ROI study [Ward 2011]). We can leverage these cost models to guide the cost-effective application of appropriate validation methods.

Appendix Selected Readings

For additional reading on the topics presented in this report, see the publications below.

- [Leveson 2004b] The Role of Software in Spacecraft Accidents. *This paper discusses a number of software-related factors that have contributed to spacecraft accidents.*
- [Dvorak 2009] NASA Study on Flight Software Complexity. *This paper reports the results of a study of issues related to the increasing complexity in on-board software.*
- [Boehm 2006] Some Future Trends and Implications for System and Software Engineering Processes. *This paper discusses several trends for improving the engineering of software-intensive systems.*
- [Feiler 2009b] Challenges in Validating Safety-Critical Embedded Systems. *This paper outlines system-level problem areas in safety-critical embedded software systems and identifies four root cause areas that can be addressed through architectural analysis.*
- [Goodenough 2010] Evaluating Software's Impact on System and System of Systems Reliability, SEI March 2010. *A paper summarizing state of reliability engineering for software-reliant systems and the need for software-specific reliability improvement programs.*
- [Jackson 2007] Software for Dependable Systems: Sufficient Evidence? *This National Research Council study identifies assurance through evidence in the form of formal analysis of system architectures as key to improving embedded software in dependable systems.*
- [Feiler 2009c] Model-Based Software Quality Assurance with the Architecture Analysis & Design Language. *A case study on use of AADL to analyze a multi-layered reference architecture, including a planning and plan execution component and its instantiation for a specific system in the autonomous space vehicle domain.*
- [Feiler 2010] System Architecture Virtual Integration: A Case Study, *A summary of an Aerospace industry (AVSI) case study involving multi-tier modeling of an aircraft and multi-dimensional analysis at different levels of fidelity in the context of a development process that involves a system integrator and multiple suppliers.*
- [Feiler 2012] Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. *This book provides an introduction to the use of AADL in architecture-centric model-based engineering.*
- [Leveson 2009] Engineering a Safer World, System Safety for the 21st Century, *This book reflects Leveson's latest insights on safety engineering.*
- [Goodenough 2009] Evaluating Hazard Mitigations with Dependability Cases. *This paper demonstrates the use of assurance cases to validate safety hazard mitigation.*
- [Miller 2010] Software Model Checking Takes Off. *This paper summarizes the state of model checking in industrial applications.*
- [Bozzano 2010] Formal Verification & Validation of AADL Models. *This work illustrates the use of the Error Annex extension to AADL in modeling and validating safety-critical systems from both a system and software perspective.*

References

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

[Ada WG 2001]
Ada Working Group. *Ada Reference Manual*, 2001. http://www.adaic.org/resources/add_content/standards/05rm/html/RM-TOC.html (Section D.13.1.).

[AFIS 2010]
Association Française d'Ingénierie Système. *CPRET* - a process definition developed by AFIS, dedicated to System Engineering and open to all domains. http://en.wikipedia.org/wiki/Process_%28engineering%29#Processes (2010).

[Aldrich 2004]
Aldrich, Bill, Fehnker, Ansgar, Feiler, Peter H., Han, Zhi, Krogh, Bruce H., Lim, Eric & Sivashankar, Shiva. "Managing Verification Activities Using SVM." *Proceedings of Sixth International Conference on Formal Engineering Methods (ICFEM).* November 2004. IEEE, 2004.

[AMSAA 2000]
Army Materiel Systems Analysis Activity. *AMSAA Reliability Growth Guide, TR-652,* Department of Defense, 2000. http://www.scribd.com/doc/22443416/AMSAA-Reliability-Growth-Guide

[Berthomieu 2010]
Berthomieu, B., Bodeveix, J.-P., Dal Zilio, S., Dissaux, P., Filali, M., Gaufillet, P., Heim, S. & Vernadat, F. "Formal Verification of AADL Models with Fiacre and Tina." *Embedded Real-time Software and Systems Conference (ERTS 2010)*. Toulouse (France), May 2010. Available through http://www.erts2010.org/Default.aspx?Id=973&Idd=981

[Black 2009]
Black, J. & Koopman, P. "System Safety as an Emergent Property in Composite Systems." *Proceedings of the International Conference on Dependable Systems and Networks (DSN'09).* Estoril, Portugal, June–July 2009.

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

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

[Boehm 2000]
Boehm, B., Abts, C., Brown A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., & Steece, B. *Software Cost Estimation with COCOMO II*, Prentice-Hall, 2000.

[Boehm 2006]
Boehm, B. "Some Future Trends and Implications for Systems and Software Engineering Processes." *Systems Engineering 9*, 1 (Jan. 2006): 1-19.
http://www.cs.cofc.edu/~bowring/classes/csis%20602/docs/FutureTrendsSEProcesses.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

Bozzano 2009]
Bozzano, Marco, Cimatti, Alessandro, Roveri, Marco, Katoen, Joost-Pieter, Nguyen, Viet Yen, & Noll, Thomas. "Codesign of Dependable Systems: A Component-Based Approach," 121–130. *Proceedings of the Seventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '09)*. Cambridge, MA, July 2009. IEEE Computer Society Press, 2009.

[Bozzano 2010]
Bozzano, M., Cavada, R., Cimatti, A., Katoen, J.-P., Nguyen, V. Y., Noll, T., & Olive, X. "Formal Verification and Validation of AADL Models." *Embedded Real-Time Software and Systems Conference (ERTS^2 2010).* Toulouse, France, May 2010.
http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0098_final.pdf

[Cimatti 2002]
Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M. Sebastiani, R., & Tacchella, A. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking." *Proceedings of International Conference on Computer-Aided Verification (CAV 2002).* Copenhagen, Denmark, July, 2002. *Lecture Notes in Computer Science*, Springer 2002.

[Clarke 1989]
Clarke, E., Long, D., & McMillan, K. "Compositional Model Checking." *Proceedings of Logic in Computer Science (LICS'89)*. Springer, 1989.

[Clarke 1994]
Clarke, E., Grumberg, O., & Long D. "Model Checking and Abstraction." *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 5 (September 1994): 1512-1542.

[Clarke 1999]
Clarke, E., Grumberg, O., & Peled, D. *Model Checking*. MIT Press, 1999.

[Clarke 2003]
Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking." *Journal of the ACM 50*, 5 (2003): 752-794.

[Clarke 2004]
Clarke, E., Kroening, D., & Lerda, F. "A Tool for Checking ANSI-C Programs." *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Barcelona, Spain, 2004. Springer, 2004.

[COMPASS 2011]
Correctness, Modeling and Performance of Aerospace Systems (COMPASS). http://compass.informatik.rwth-aachen.de/ (2011).

[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

[Couch 2010]
Couch, Mark and Lindell, Dennis. *Study on Rotorcraft Safety and Survivability*. AHS International Vertical Flight Society, 2010.
http://vtol.org/B17CF690-F5FE-11E0-89190050568D0042

[Cousot 2005]
Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. "The ASTRÉE Analyser." *ESOP 2005: The European Symposium on Programming*. Endinburgh, Scotland, April 2005. *Lecture Notes in Computer Science*, Springer, 2005.

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

[Dardenne 1993]
Dardenne, A., v. Lamsweerde, A., & Fickas, S. "Goal Directed Requirements Acquisition." M. Sintzoff, C. Ghezzi, and G. Roman, eds, *Science of Computer Programming 20*, 1-2 (April 1993): 3-50. Elsevier Science, 1993.

[DARPA 2010]
DARPA. *META Program* as part of *Adaptive Vehicle Make (AVM)*. http://www.darpa.mil/Our_Work/TTO/Programs/AVM/AVM_Design_Tools_%28META%29.aspx (2010).

[Delange 2009]
DeLange, Julien , Pautet, Laurent, & Feiler, Peter. "Validating Safety and Security Requirements for Partitioned Architectures." *Proceedings of the 14th International Conference on Reliable Software Technologies (RTS 2009*) *Ada Europe*. Brest, France, June 2009. *Lecture Notes in Computer Science 5570*, Springer, 2009.

[Delange 2010a]
Delange, Julien, Pautet, Laurent, & Kordon, Fabrice. "Modeling and Validation of ARINC653 Architectures," *Embedded Real-time Software and Systems Conference (ERTS2010)*, May 2010. http://www.erts2010.org/Default.aspx?Id=973&Idd=982

[Delange 2010b]
Delange, J., Pautet, L., & Kordon, F. "Design, Verification and Implementation of MILS Systems." *21st IEEE International Symposium on Rapid System Prototyping (RSP 2010*). June 2010. IEEE, 2010.

[Delehaye 2009]
Delehaye, Mathieu & Ponsard, Christophe. "Towards a Model-Driven Approach for Mapping Requirements on AADL Architecture." *Proceedings of 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS09).* UML & AADL Workshop, June 2009. IEEE, 2009.

[DeNiz 2008]
DeNiz, Dio & Feiler, Peter. "On Resource Allocation in Architectural Models," *Proceedings of the 11th IEEE International Symposium on Object/Service-Oriented Real-time Distributed Computing*. May 2008. IEEE, 2008

[DeNiz 2012]
DeNiz, D., Feiler, P., Gluch, D., & Wrage, L. *A Virtual Upgrade Validation Method for Software Reliant Systems* (CMU/SEI-2012-TR-005). Software Engineering Institute, Carnegie Mellon University, 2012.

[DeVale 2002]
DeVale, J. & Koopman, P. "Robust Software – No More Excuses." *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02).* Washington D.C., June 2002. IEEE, 2002.

[DoD 2010]
Department of Defense, Reliability Information Analysis Center. *Software-in-Systems Reliability Toolkit.* 2010. http://theriac.org/riacapps/search/?mode=displayresult&id=545

[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.

[Dwyer 1999]
Dwyer, M., Avrunin, G., & Corbett, J. "Patterns in Property Specifications for Finite-state Verification." *Proceedings of the 21st International Conference on Software Engineering (ICSE 99).* Los Angeles, CA, May 1999. ACM, 1999.

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

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

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

[FAA 2009a]
Federal Aviation Administration. *Requirements Engineering Management Handbook DOT/FAA/AR-08/32*. 2008.
http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR-08-32.pdf

[FAA 2009b]
Federal Aviation Administration. *Requirements Engineering Management Findings Report DOT/FAA/AR-08/34.* 2008.
http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR-08-34.pdf.

[FBK 2009]
Foundazione Bruno Kessler. *RAT—Requirements Analysis Tool.* http://rat.fbk.eu (2009).

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

[Feiler 2004]
Feiler, Peter H., Gluch, David P., Hudak, John., & Lewis, Bruce A. "Pattern-Based Analysis of an Embedded Real-time System Architecture." *Proceedings of IFIP World Computer Congress - Workshop on Architecture Description Languages (WADL04*). August, 2004, Toulouse, France. Volume 176/2005, Springer, 2004.

[Feiler 2008]
Feiler, Peter. "Efficient Embedded Runtime Systems through Port Communication Optimization," 294-300. *13th IEEE International Conference on Engineering of Complex Computer Systems*. Belfast, Northern Ireland, March 2008. IEEE Computer Society, 2008. Available through http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/f/Feiler:Peter_H=.html

[Feiler 2009a]
Feiler P. H., Hansson J., de Niz D., & Wrage L. *System Architecture Virtual Integration: An Industrial Case Study* (CMU/SEI-2009-TR-017). Software Engineering Institute, Carnegie Mellon University, 2009. http://www.sei.cmu.edu/library/abstracts/reports/09tr017.cfm

[Feiler 2009b]
Feiler, Peter H. "Challenges in Validating Safety-Critical Embedded Systems," *Proceedings of SAE International AeroTech Congress*, Warrendale, PA, November 2009. https://www.sae.org/technical/papers/2009-01-3284

[Feiler 2009c]
Feiler P., Gluch D., Weiss K., & Woodham K. "Model-Based Software Quality Assurance with the Architecture Analysis & Design Language," Presented at *AIAA Infotech @Aerospace 2009.* Seattle, Washington, April 2009. http://pub-lib.jpl.nasa.gov/docushare/dsweb/ImageStoreViewer/Document-364

[Feiler 2010]
Feiler, P., Wrage, L., & Hansson, J. "System Architecture Virtual Integration: A Case Study." *Embedded Real-time Software and Systems Conference (ERTS 2010).* May 2010. http://www.erts2010.moonaweb.com/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010%202/ERTS2010_0105_final.pdf

[Feiler 2012]
Feiler, P., Gluch, D., *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012. Part of the SEI Series in Software Engineering series. ISBN-10: 0-321-88894-4.

[GAO 2008]
General Accounting Office. *DOD's Goals for Resolving Space Based Infrared System Software Problems Are Ambitious* (GAO-08-1073). 2008. http://www.gao.gov/new.items/d081073.pdf

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

[Goodenough 2009]
Goodenough, J. B. & Barry, M. "Evaluating Hazard Mitigations with Dependability Cases." American Institute of Aeronautics and Astronautics, 2009. http://www.sei.cmu.edu/library/assets/Evaluating%20Hazard%20Mitigations%20with%20Dependability%20Cases.pdf

[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://www.sei.cmu.edu/library/abstracts/whitepapers/swandreliability.cfm

[Gurfinkel 2008]
Gurfinkel A. & Chaki, S. "Combining Predicate and Numeric Abstraction for Software Model Checking." *In Proceedings of the Formal Methods in Computer-Aided Design International Conference (FMCAD 2008)*. Portland, Oregon, November 2008. Curran Associates, 2008.

[Groundwater 1995]
Groundwater, E. H., Miller, L. A., & Mirsky, S. M. *Guidelines for the Verification and Validation of Expert System Software and Conventional Software* (NUREG/CR-6316, SAIC-95/1028). U.S. Nuclear Regulatory Commission, 1995.

[Grumberg 1994]
Grumberg, O. & Long, D. "Model Checking and Modular Verification." *Transactions on Programming Languages and Systems 16,* 3: 843-871, May 1994.

[Hansson 2009]
Hansson, Jörgen, Lewis, Bruce, Hugues, Jérôme, Wrage, Lutz, Feiler, Peter H., & Morley, John. "Model-Based Verification of Security and Non-Functional Behavior using AADL," *IEEE Journal on Security and Privacy PP*, 99: 1-1. IEEE Computer Society, 2009.

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

[Heimdahl 1996]
Heimdahl, Mats P. E. & Leveson, Nancy. "Completeness and Consistency in Hierarchical State-Based Requirements" *IEEE Transactions on Software Engineering 22, 6:* 363-377 (June 1996). IEEE Computer Society, 1996.

[Heitmeyer 1995]
Heitmeyer, C., Labaw, B., & Kiskis, D. "Consistency Checking of SCR-Style Requirements Specification," 56-65. *Proceedings of the Second IEEE International Symposium on Requirements Engineering.* York, England, March 1995. IEEE Computer Society, 1995.

[Heitz 2008]
Heitz, Maurice, Gabel, Sebastien, Honoré, Julien, Dumas, Xavier, Ober, Iulan, & Lesens, David. "Supporting a Multi-formalism Model Driven Development Process with Model Transformation, a TOPCASED Implementation." *Proceedings of 4th International Congress on Embedded Real-Time Systems*. Toulouse, France, January 2008.

[Herring 2007]
Herring, Margaret Stringfellow, Owens, Brandon D., Leveson, Nancy, Ingham, Michel, & Weiss, Kathryn Ann. *A Safety-Driven, Model-Based System Engineering Methodology, Part I.* MIT Technical Report, December 2007.

[Holzmann 2003]
Holzmann, G. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[IEEE 1471]
Institute of Electrical and Electronics Engineers Standards Association. *1471-2000 – IEEE Recommends Practice for Architectural Description for Software Intensive Systems*. 2000. Available through http://standards.ieee.org/findstds/standard/1471-2000.html

[INCOSE 2010]
International Council on System Engineering. *MBSE Workshop*. Phoenix, Arizona, January 2010. http://www.incose.org/newsevents/workshop/details.aspx?id=MBSE

[ISO/IEC 2008a]
International Organization for Standardization/International Electrotechnical Commission. *ISO/IEC 15288:2008 Systems and Software Engineering–System Life Cycle Processes.* 2008. Available through http://www.iso.org/iso/catalogue_detail?csnumber=43564

[ISO/IEC 2008b]
International Organization for Standardization**/**International Electrotechnical Commission. *ISO/IEC 12207: 2008 Systems and Software Engineering–System Life Cycle Processes.* 2008. Available through http://www.iso.org/iso/catalogue_detail?csnumber=43447

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

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

[Joshi 2007]
Joshi, A., Vestal, S., & Binns, P. "Automatic Generation of Static Fault Trees from AADL Models." *Proceedings of the 37th Annual IEEE/IFIP Conference on Dependable Systems and Networks' Workshop on Dependable Systems*. Edinburgh, Scotland, June 2007. IEEE, 2010.

[Kaner 2004]
Kaner, C. "Software Engineering Metrics: What Do They Measure and How Do We Know?" *Proceedings of the 10th International Software Metrics Symposium*. Chicago, Illinois, September 2004. IEEE, 2004.

[Katoen 2009]
Katoen, J. P., Zapreev, I., Moritz Hahn, E., Hermanns, H., & Jansen, D. "The Ins and Outs of the Probabilistic Model Checker MRMC." *Proceedings of the International Conference on Quantitative Evaluation of Systems (QEST)*. Budapest, Hungary, September 2009. IEEE, 2009. Available through http://www.computer.org/portal/web/csdl/doi/10.1109/QEST.2009.11

[Kazman 2000]
Kazman, R., Klein, M., & Clements, P. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004). Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/library/abstracts/reports/00tr004.cfm

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

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

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

[Kwiatkowska 2010]
Kwiatkowska, M., Norman, G., & Parker, D. "Advances and Challenges of Probabilistic Model Checking,"1691-1698. *Proceedings of the 48th Annual Allerton Conference on Communication, Control and Computing*. Monticello, Illinois, September-October 2010. IEEE, 2010.

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

[Laprie 1995]
Laprie, J.-C., Arlat, J., Blanquart, J.-P., Costes, A., Crouzet, Y., Deswarte, Y., Fabre, J.-C., Guillermain, H., Kaniche, M., Kanoun, K., Mazet, C., Powell, D., Rabejac, C., & Thvenod, P. *Dependability Handbook*. Cépaduès, 1995.

[Lee 2002]
Lee, G., Howard, J., & Anderson, P. "Safety-Critical Requirements Specification and Analysis Using SpecTRM." Safeware Engineering, 2002. http://www.safeware-eng.com/system%20and%20software%20safety%20publications/sswg2002.pdf (2003).

[Lee 2008]
Lee, Edward. *Cyber Physical Systems: Design Challenges (TR.UCB/EECS-2008-8)*. Electrical Engineering and Computer Sciences, University of California at Berkeley, 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html

[Letier 2002]
Letier, E. & van Lamsweerde, A. "Deriving Operational Software Specifications from System Goals," 119-128. *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. Charleston, SC, Nov. 2002. ACM, 2002.

[Leveson 1995]
Leveson, Nancy G. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[Leveson 1994]
Leveson, N., Heimdahl, M., Hildreth, H., & Reese, J. "Requirements Specifications for Process-Control Systems." *IEEE Transactions on Software Engineering 20*, 9 (September 1994): 684–707.

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

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

[Leveson 2004b]
Leveson, Nancy G. “The Role of Software in Spacecraft Accidents.” *Journal of Spacecraft and Rockets 41*, 4 (July 2004): 564–575.

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

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

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

[Miller 2001]
Miller, S. & Tribble, A. “Extending the Four-Variable Model to Bridge the System-Software Gap.” Presented at the 20th *Digital Avionics Systems Conference (DASC01)*. Daytona Beach, FL, October 2001.

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

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

[Miller 2010]
Miller, S., Whalen, M., & Cofer, D. “Software Model Checking Takes Off.” *Communications of the ACM 53*, 2 (2010): 58–64.
http://cacm.acm.org/magazines/2010/2/69362-software-model-checking-takes-off/fulltext

[Nam 2009]
Nam, M., Pellizzoni, R., Sha, L., & Bradford, R. M. "ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs," 11–22. *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*. Potsdam, Germany, June 2009. IEEE Computer Society Press, 2009.

[NDIA 2008]
National Defense Industrial Association System Assurance Committee. *Engineering for System Assurance (Version 1.0)*. National Defense Industrial Association, 2008. http://www.acq.osd.mil/se/docs/SA-Guidebook-v1-Oct2008.pdf

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

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

[NCSC 1988]
National Computer Security Center. *Glossary of Computer Security Terms.* DoD Directive 5215.1. NCSC-TG-004-88. 1988. http://packetstormsecurity.org/files/13995/NCSC-TG-004.txt

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

[OMG MARTE 2009]
OMG MARTE. *The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems*. http://www.omgmarte.org (2009).

[Owens 2007]
Owens, Brandon D., Herring, Margaret S., Leveson, Nancy, Ingham, Michel, & Weiss, Kathryn Ann. "Application of a Safety-Driven Design Methodology to an Outer Planet Exploration Mission." Presented at the *IEEE Aerospace Conference*. Big Sky, MN, March 2008.

[Paige 2009]
Paige, Richard F., Rose, Louis M., Ge, Xiaocheng, Kolovos, Dimitrios S., & Brooke, Phillip J. "Automated Safety Analysis for Domain-Specific Languages," 229–242. *Lecture Notes in Computer Science 5421, Models in Software Engineering*. Springer, 2009.

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

[Perrotin 2010]
Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., & Hugues, J. "The TASTE Toolset: Turning Human Designed Heterogeneous Systems into Computer Built Homogeneous Software." Present-

ed at the Embedded Real-Time Software and Systems Conference (ERTS2010). Toulouse, France, May, 2010.

[Raghav 2010]
Raghav, Gopal, Gopalswamy, Swaminathan, Radhakrishnan, Karthikeyan, Hugues, Jérôme, & Delange, Julien. "Model Based Code Generation for Distributed Embedded Systems." Presented at the *Embedded Real-Time Software and Systems Conference (ERTS2010).* Toulouse, France, May, 2010.

[Ravenscar 2011]
Ravenscar Profile. http://en.wikipedia.org/wiki/Ravenscar_profile (2011).

[Redman 2010]
Redman, David, Ward, Donald, Chilenski, John, & Pollari, Greg. "Virtual Integration for Improved System Design," *Proceedings of The First Analytic Virtual Integration of Cyber-Physical Systems Workshop* in conjunction with the *Real-Time Systems Symposium (RTSS 2010).* San Diego, CA, November 2010. http://www.andrew.cmu.edu/user/dionisio/avicps2010-proceedings/virtual-integration-for-improved-system-design.pdf

[Rifaut 2003]
Rifaut, A., Massonet, P., Molderez, J.-F., Ponsard, C., Stadnik, P., van Lamsweerde, A., & Van Hung, T. "FAUST: Formal Analysis of Goal-Oriented Requirements Using Specification Tools," 350. *Proceedings of the Requirements Engineering Conference (RE'03)*. Monterey, CA, September 2003. IEEE Computer Society Press, 2003.

[RTCA 1992]
Radio Technical Commissions for Aeronautics, in collaboration with EUROCAEDO-178B. *Incorporated Systems and Equipment Certification,* 1992.
http://en.wikipedia.org/wiki/DO-178B

[Rugina 2008]
Rugina, Ana-Elena, Kanoun, Karama, Kaaniche, Mohamed, & Feiler, Peter. "Software Dependability Modeling Using an Industry-Standard Architecture Description Language." *Proceedings of the 4th International Congress on Embedded Real-Time Systems*. Toulouse, France, January 2008. arXiv, 2008.

[Rushby 1981]
Rushby, John. "The Design and Verification of Secure Systems," 12–21. *Proceedings of the Eighth Symposium on Operating Systems Principles.* Asilomar, CA, December 1981. *ACM Operating Systems Review 15*, 5 (December 1981). http://portal.acm.org/citation.cfm?id=806586

[Rushby 1994]
Rushby, John. "Critical System Properties: Survey and Taxonomy." *Reliability Engineering and System Safety 43*, 2 (1994): 189–219.

[Rushby 1999]
Rushby, John. *Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance* (DOT/FAA/AR-99/58, CR-1999-209347). NASA, 1999.

[Rushby 2007]
Rushby, John. "Just-in-Time Certification," 15–24. *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*. Auckland, New Zealand, July 2007. IEEE Computer Society Press, 2007.

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

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

[SAE 2006]
Society of Automotive Engineers International. *Architecture Analysis & Design Language (AADL) Annex Volume 1: AADL Meta model & XML Interchange Format Annex, Error Model Annex, Programming Language Annex*. SAE International Standards Document AS5506/1, June 2006. http://standards.sae.org/as5506/1

[SAE 2004-2012]
Society of Automotive Engineers International. *Architecture Analysis & Design Language (AADL)*. SAE International Standards Document AS5506B, 2012. http://standards.sae.org/as5506b/

[SAE 2010]
Society of Automotive Engineers International. *Guidelines for Development of Civil Aircraft and Systems*. 2010. http://standards.sae.org/arp4754a/

[SAE 2011]
Society of Automotive Engineers International. *Architecture Analysis and Design Language (AADL) Annex Volume 2*: Behavior Annex, Data Modeling Annex, ARINC653 Annex. SAE International Standards Document AS5506/2, 2011. http://standards.sae.org/as5506/2

[Seacord 2008]
Seacord, Robert C. *The CERT C Secure Coding Standard*. Addison-Wesley, 2008 (ISBN 978-0-321-56321-7).

[Senn 2008]
Senn, J. Laurent & Diguet, J. P. "Multi Level Power Consumption Modelling in the AADL Design Flow for DSP, GPP, and FPGA." *Proceedings of the First International Workshop on Model Based Architecting and Construction of Embedded Systems*. Toulouse, France, September 2008. Springer, 2008.

[Sha 2009]
Sha, Lui. "Resilient Mixed-Criticality Systems." *CrossTalk: The Journal of Defense Software Engineering 22*, 6 (September/October 2009): 9-14.

[Singhoff 2009]
Singhoff, F., Plantec, A., Dissaux, P., & Legrand, J. "Investigating the Usability of Real-Time Scheduling Theory with the Cheddar Project." *Journal of Real-Time Systems 43*, 3 (November 2009): 259-295.

[Sokolsky 2009]
Sokolsky, Oleg, Lee, Insup, & Clarke, Duncan. "Process-Algebraic Interpretation of AADL Models." *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*. Brest, France, June 8–12, 2009. Springer, 2009.

[SPICES 2006]
The SPICES Consortium. *Support for Predictable Integration of Mission-Critical Embedded Systems*. http://www.spices-itea.org/public/info.php (2006-2009).

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

[SysML.org 2010]
SysML.org. *UML Profile for System Engineering Modeling (SysML).* http://www.sysml.org (2003-2011).

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

[UML OMG 2009]
Unified Modeling Language Object Management Group, *UML Version 2.2.*, 2009. http://www.omg.org/spec/UML/2.2/

[UPPAAL 2009]
UPPAAL: an integrated tool environment for modeling, validation and verification of real-time systems. Uppsala Universitet and Aalborg University. http://www.uppaal.org/ (2009).

[U.S. Army 2007]
U.S. Army Research, Development and Acquisition. *Airworthiness Qualification of Aircraft Systems*. Army Regulation 70-62, 2007. http://www.army.mil/usapa/epubs/pdf/r70_62.pdf

[van Lamsweerde 2000]
van Lamsweerde, A. & Letier, E. "Handling Obstacles in Goal-Oriented Requirements Engineering." *IEEE Transactions in Software Engineering 26*, 10 (October 2000): 978-1005.

[van Lamsweerde 2004a]
van Lamsweerde, A. "Elaborating Security Requirements by Construction of Intentional Anti-Models," 148–157. *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, Scotland, May 2004. IEEE Computer Society Press, 2004.

[van Lamsweerde 2004b]
van Lamsweerde, A. “Goal-Oriented Requirements Enginering [*sic*]: A Roundtrip from Research to Practice,” 4–7. *Proceedings of the 12th IEEE International Requirements Engineering Conference*. Kyoto, Japan, September 2004. IEEE Computer Society Press, 2004.

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

[Weinstock 2004]
Weinstock, Charles B., Goodenough, John B., & Hudak, John J. *Dependability Cases* (CMU/SEI-2004-TN-016). Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/library/abstracts/reports/04tn016.cfm

[Ward 2011]
Ward, D. and Helton, S., "Estimating Return on Investment for SAVI (a Model-Based Virtual Integration Process)," SAE Int. J. Aerosp. 4(2):934-943, 2011, doi:10.4271/2011-01-2576.

[Wikipedia 2011]
Wikipedia. *Unified Modeling Language*, Collage of UML Diagrams. http://en.wikipedia.org/wiki/ Unified_Modeling_Language (2011).

Acronyms

Acronym	Definition
AADL	Architectural Analysis and Description Language
ACM	Association for Computing Machinery
AED	US Army Aviation Engineering Directoration
AF	Air Force
AFB	Air Force Base
AFIS	Association Française d'Ingénierie Système
AHS	American Helicopter Society
AIAA	American Institute of Aeronautics and Astronautics
AMRDEC	Aviation and Missile Research, Development and Engineering Center
AMSAA	US Army Material Systems Analysis Activity
ANSI	American National Standards Institute
AR	Army Regulation
ARP	Aeronautical Recommended Practice
AS	Aeronautical Standard
ASIIST	Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs
ASN	Abstract Syntax Notation
ASSERT	Automated proof-based System and Software Engineering for Real-Time applications
ASTRÉE	Analyseur statique de logiciels temps-réel embarqués (real- time embedded software static analyzer).
ATAM	Architecture Tradeoff Analysis Method®,
AVM	Adaptive Vehicle Make
AVSI	Aerospace Vehicle Systems Institute
BAE	British Aerospace & Engineering
CA	California

Acronym	Definition
CAAS	Common Avionics Architecture System
CAST	Commercial Aviation Safety Team
CAV	Computer Aided Verification
CBMC	C Bounded Model Checker
CCA	Common Cause Analysis
CCM	CORBA Component Model
CD	Compact Disc
CEGAR	Counter Example-Guided Abstraction Refinement
CERT	CERT" and "CERT Coordination Center" are registered service marks of Carnegie Mellon University. CERT is not an acronym.
CMMI	Capability Maturity Model Integration
CMU	Carnegie Mellon University
CO	Colorado
COCOMO	COnstructive COst MOdel
COMPASS	*Correctness, Modeling and Performance of Aerospace Systems*
COQUALMO	COnstructive QUALity MOdel
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off the Shelf
CTL	Computation Tree Logic
DACS	Data and Analysis Center for Software
DARPA	Defense Advanced Research Projects Agency
DC	Decision Coverage
DMA	Direct Memory Access
DNA	Deoxyribonucleic acid
DO	Document
DOORS	Dynamic Object Oriented Requirements System
DOT	Department of Transportation
DRE	Distributed Real-time Embedded

Acronym	Definition
DSP	Digital Signal Processing
DTIC	Defense Technology Information Center
EECS	Electrical Engineering Computer Sciences
ERTS	Embedded Real Time Systems
ESA	European Space Agency
ESOP	European Symposium on Programming
FAA	Federal Aviation Administration
FAUST	Formal Analysis of Goal-Oriented Requirements Using Specification Tools
FDA	Food and Drug Administration
FDIR	Fault Detection, Isolation and Recovery
FL	Florida
FMCAD	*Formal Methods in Computer-Aided Design International Conference*
FMEA	Failure Modes Effects Analysis
FPGA	Field Programmable Gate Array
FTA	Fault Tree Analysis
GAO	Government Accounting Office
GE	General Electric
GORE	Goal-oriented Requirements Engineering
GPP	General Purpose Processor
ICFEM	*International Conference on Formal Engineering Methods*
ICSE	*International Conference on Software*
IEC	International Engineering Consortium
IEEE	Institute of Electrical and Electronics Engineers
IFIP	International Federation of Information Processing
IMA	Integrated Modular Avionics
INCOSE	International Council on Systems Engineering
IRST	Istituto per la Ricerca Scientifica e Tecnologica

Acronym	Definition
ISBN	International Standard Book Number
ISO	International Standards Organization
ISSRE	International Symposium on Software Reliability Engineering
ITC	International Test Conference
LTL	Linear Temporal Logic
MA	Massachusetts
MARTE	*Modeling and Analysis of Real-Time and Embedded Systems*
MBSE	Model Based Systems Engineering
MC	Modified Condition
MIL	Military
MILS	Multiple Independent Levels of Security
MISRA	Motor Industry Software Reliability Association
MIT	Massachusetts Institute of Technology
MN	Minnesota
MPEC	*Models and Processes for the Evaluation of COTS Components*
MRMC	Markov Reward Model Checker
MTBF	Mean Time Between Failure
MTTF	Mean Time To Fix
NASA	National Aeronautics and Space Administration
NCSC	National Computer Security Center
NDIA	National Defense Industrial Association
NG	Newsgroup
NIST	National Institute of Science and Technology
NPV	Net Present Value
NRC	National Research Council
NSN	National Stock Number
NTIS	National Technical Information Service

Acronym	Definition
NUREG	US Nuclear Regulatory Commission
OMB	Office of Management & Budget
OMG	Object Management Group
OSATE	Open Source AADL Tool Environment
PA	Pennsylvania
POC	Proof of Concept
POSIX	Portable Operating System Interface
PRISM	probabilistic model checker
PSSA	Preliminary System Safety Assessment
QEST	*Quantitative Evaluation of Systems*
RAT	Requirements Analysis Tool
RFP	Request for Proposal
RMA	Rate Monotonic Analysis
ROI	Return on Investment
ROM	Read Only Memory
RSML	Requirements State Machine Language
RSP	Rapid System Prototyping
RTCA	Radio Technical Commission for Aeronautics
RTE	Run Time Errors
RTS	Reliable Software Technologies
RTSS	Real-time System Symposium
SA	System Assurance or Situational Awareness
SAE	Society of Automotive Engineers
SAIC	Science Applications International Corporation
SAVI	Systems Architecture Virtual Integration
SC	South Carolina
SCADE	Safety-Critical Application Development Environment

Acronym	Definition
SCR	Software Cost Reduction
SEI	Software Engineering Institute
SIGSOFT	Special Interest Group on Software Engineering
SIL	System Integration Lab
SLOC	Software Lines of Code
SMT	Satisfiability Modulo Theory
SMV	Symbolic Model Verification
SPICES	*Support for Predictable Integration of Mission-Critical Embedded Systems*
SPIN	Simple Promela Interpreter
SSA	System Safety Assessment
STAMP	Systems Theory Accident Model and Processes
STD	Standard
STPA	STAMP to Prevent Accidents
SVM	System Verification Manager
SW	Software
TACAS	*Tools and Algorithms for the Construction and Analysis of*
TASTE	The ASSERT Set of Tools for Engineering
TINA	TIme petri Net Analyzer
TOPCASED	Toolkit in OPen-source for Critical Applications and SystEms Development
TOPLAS	*Transactions on Programming Languages and Systems*
TRL	Technical Readiness Level
UK	United Kingdom
UML	Unified Modeling Language
UNIX	Uniplexed Information and Computing System (was UNICS)
UPPAAL	Uppsala Universitet and Aalborg University Language
VA	Virginia
VERSA	Verification Execution and Rewrite System for ACSR (Algebra of Communicating Shared Resources).

Acronym	Definition
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VSIL	Virtual System Integration Laboratory
VUV	Virtual Upgrade Validation
WG	Working Group
XMI	XML Message Interface
XML	Extensible Markup Language

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2012	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Reliability Validation and Improvement Framework	FA8721-10-C-0008

6. AUTHOR(S)
Peter F. Feiler, John B. Goodenough, Arie Gurfinkel, Charles B. Weinstock, Lutz Wrage

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2012-SR-013

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent ESC/XPK 20 Schilling Circle, Bldg 1305, 3rd floor Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

Software-reliant systems such as rotorcraft and other aircraft have experienced exponential growth in software size and complexity. The current software engineering practice of "build then test" has made them unaffordable to build and qualify. This report discusses the challenges of qualifying such systems, presenting the findings of several government and industry studies. It identifies several root cause areas and proposes a framework for reliability validation and improvement that integrates several recommended technology solutions: validation of formalized requirements; an architecture-centric, model-based engineering approach that uncovers system-level problems early through analysis; use of static analysis for validating system behavior and other system properties; and managed confidence in qualification through system assurance. This framework also provides the basis for a set of metrics for cost-effective reliability improvement that overcome the challenges of existing software complexity, reliability, and cost metrics.

14. SUBJECT TERMS	15. NUMBER OF PAGES
reliability assessment, safety-criticality requirements, coverage metric, error leakage rate	117

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL