DOT/FAA/TC-15/57

Federal Aviation Administration
William J. Hughes Technical Center
Aviation Research Division
Atlantic City International Airport
New Jersey 08405

Software Assurance Approaches, Considerations, and Limitations: Final Report

October 2016

Final Report

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/TC-15/57		

4. Title and Subtitle	5. Report Date
SOFTWARE ASSURANCE APPROACHES, CONSIDERATIONS, AND LIMITATIONS FINAL REPORT	October 2016
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Mats Heimdahl, University of Minnesota; Nancy Leveson, Massachusetts Institute of Technology. Julie Redler, Melanie Felton, and Grady Lee are from Safeware Engineering Corporation	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Safeware Engineering Corporation 180 Nickerson Street, Suite 110 Seattle, WA 98109	
	11. Contract or Grant No. DTFACT-11-C-00004

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U.S. Department of Transportation Federal Aviation Administration National Headquarters 950 L'Enfant Plaza N SW Washington, DC 20024	Final Report July 2011–November 2014
	14. Sponsoring Agency Code AIR-134

15. Supplementary Notes

The Federal Aviation Administration William J. Hughes Technical Center Aviation Research Division COR was Srini Mandalapu.

16. Abstract

The cost of developing software in compliance with RTCA/DO-178B/RTCA/DO-278 is generally high. Nevertheless, these standards have helped to ensure the development of software systems of high integrity with excellent operational histories. The "Alternative Approaches to Software Assurance" three-phase study was undertaken to evaluate the current state of software assurance processes and propose alternative approaches with the potential to streamline the process and reduce the assurance costs without compromising safety.

Phase 1 work focused on three areas: an examination of alternative methods, a comparison of aerospace industry standards to other safety-critical industry's standards, and a poll to query aviation industry personnel on their experience with DO-178B and DO-278. The findings from Phase 1 did not highlight any alternative approaches that could replace DO-178B or DO-278. The authors recommended looking at technical advances that could still meet the goal of the study but were not necessarily alternatives to DO-178B and DO-278. The Phase 1 findings directed the team to look at techniques that could help users of the standards to streamline the process (and realize cost benefits) by ensuring the requirements were complete and correct early in the development process. The goal of Phase 2 was to conduct an in-depth study of techniques that warranted further study from Phase 1, including: hazard analysis; human reviews; model-based specification and analysis; architectural modeling and analysis; and collection of information regarding how each approach helps in streamlining the certification process and which approaches are best used for commercial off-the-shelf and legacy software.

The research from the first two phases directed the team to further focus on Systems Theoretic Process Analysis (STPA), model-based development, and formal verification in the third phase. Although these methods have been around for some time, there have been advancements in model-based development and formal verification that deemed it worthwhile to re-visit them. The Phase 3 work also highlighted how STPA can catch more system and software errors in the requirements than the traditional hazard analysis techniques, such as fault tree analysis. The analysis demonstrated how STPA could be applied to a flight guidance system and how hazard causes could be mitigated. The research also looked at cost savings that were realized by Rockwell Collins when they used model-based development and by Airbus when they used formal verification on their projects. A discussion about the pitfalls of using model-based development and formal verification was also included.

17. Key Words	18. Distribution Statement
Software assurance, Alternative approaches, Requirements assurance, Software architecture assurance, Quality assurance, STPA, Systems, Hazard analysis, Model-based development, Formal verification	This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	175	

TABLE OF CONTENTS

Page

EXECUTIVE SUMMARY vii

1. INTRODUCTION 1

1.1 PHASE 1: FIND ALTERNATIVE APPROACHES TO DO-178B AND DO-278 2
1.2 PHASE 2: TECHNICAL ADVANCES TO STREAMLINE THE SOFTWARE DEVELOPMENT PROCESS 2
1.3 PHASE 3: FURTHER INVESTIGATION OF TECHNICAL ADVANCEMENTS 3

2. PHASE 1 RESEARCH 4

2.1 DOCUMENTS AND STANDARDS REVIEWED 4

2.1.1 National Academies Press Report: “Software for Dependable Systems: Sufficient Evidence?” 4
2.1.2 Evaluation of Other Industry Software Assurance Standards 9

2.2 POLL RESULTS OF ALTERNATIVE APPROACHES TO SOFTWARE ASSURANCE 12

2.2.1 Poll Summary 13

2.3 OVERVIEW OF ASSURANCE METHODS 14

2.3.1 Dynamic Analysis 14
2.3.2 Static Analysis 16
2.3.3 Quality Assurance 17
2.3.4 Verification and Validation of Non-Software Life-Cycle Products 18
2.3.5 Service History 18
2.3.6 Software Metrics and Reliability Models 18
2.3.7 Hazard Analysis and Safety Engineering 23
2.3.8 Model-Based Development and Automatic Code Generation 24
2.3.9 Assurance Cases 25
2.3.10 Incremental Integration of Components 26
2.3.11 Reverse Engineering 27
2.3.12 Phase 1 Summary 28

3. PHASE 2 RESEARCH: TECHNICAL ADVANCES TO STREAMLINE THE SOFTWARE DEVELOPMENT PROCESS 29

3.1 STREAMLINING THE PROCESS: INCORPORATING SAFETY AND REDUCING COST 29

3.1.1 Requirements Assurance 30
3.1.2 Software Architecture Assurance 41
3.1.3 Implementation (coding) Assurance 42

4. PHASE 3 RESEARCH 47

4.1 TECHNICAL ADVANCES TO AID IN FINDING SAFETY CONSTRAINTS 47

4.1.1 STPA Hazard Analysis 47

4.2 CASE STUDIES ON COST SAVINGS RELATED TO MODEL-BASED DEVELOPMENT 60

4.2.1 Model-Based Development 60

4.3 FORMAL VERIFICATION 69

4.3.1 Reducing Rework and Test Effort Through Formal Verification 70
4.3.2 Reducing Testing Efforts Through Formal Verification 73
4.3.3 Formal Methods Summary and Recommendations 74

5. RESULTS AND FURTHER WORK 77
6. REFERENCES 79

APPENDICES

A—POLL ON ALTERNATIVE APPROACHES TO SOFTWARE ASSURANCE
B—SYSTEMS THEORETIC PROCESS ANALYSIS

LIST OF FIGURES

Figure		Page
1	Example NextGen control structure diagram	33
2	Example NextGen inadequate control actions	35
3	Example flight control panel	48
4	Example PFD from speedbirdair.com	49
5	FGS control structure diagram	51
6	Interactions between the FMS and FGS	54

LIST OF ABBREVIATIONS AND ACRONYMS

AADL	Architecture Analysis and Design Language
ADS	Air Data System
ADS-B	Automatic Dependent Surveillance-Broadcast
ALT	Altitude hold
AP	Autopilot
APPR	Approach mode
CNS/ATM	Communications, Navigation, Surveillance/Air Traffic Management
COTS	Commercial off-the-shelf
CTL	Computation tree logic
FCP	Flight control panel
FCS	Flight control system
FD	Flight director
FGS	Flight guidance system
FHA	Functional Hazard Assessment
FMEA	Failure modes and effects analysis
FMS	Flight Management System
FTA	Fault Tree Analysis
GA	Go Around
GPS	Global Positioning System
HDG	Heading Select
JAXA	Japan Aerospace Exploration Agency
IAS	Indicated airspeed
LTL	Linear Time Temporal Logic
MC/DC	Modified condition/decision coverage
NAV	Lateral navigation
NextGen	Next Generation Air Transportation System
PFD	Primary flight display
PSA	Preselected altitude
RC/DC	Reinforced condition/decision coverage
RSML	Requirements State Machine Language
RSML^{-e}	Requirements State Machine Language without Events
RTCA	RTCA, Inc. (formerly Radio Technical Commission for Aeronautics)
SCADE	Safety Critical Application Development Environment
SCR	Software cost reduction
STPA	Systems Theoretic Process Analysis
STAMP	System Theoretic Accident Model and Processes
SYNC	Synchronization
TCAS II	Traffic Alert and Collision Avoidance System II
UML	Unified Modeling Language
V&V	Validation and verification
VAPPR	Vertical approach mode
VGA	Vertical Go Around
VS	Vertical speed

EXECUTIVE SUMMARY

The cost of developing software in compliance with RTCA/DO-178B and/or RTCA/DO-278 is generally high. Nevertheless, these standards have helped ensure the development of software systems of high integrity with excellent operational histories. The primary goal of this study was to evaluate the current state of software assurance processes and propose alternative approaches with the potential to streamline the process and reduce the assurance costs without compromising safety. The study was conducted in three phases. In the first phase of this study, a review of the safety-critical industry's software assurance was conducted. The standards considered in this review include international software standards and other standards practiced in the development of space and medical systems. All the software assurance processes that were reviewed did not provide any better alternatives to the processes in DO-178B or DO-278.

Phase 1 included querying aviation and safety industry personnel to gain insight as to how they felt DO-178B and DO-278 were serving their industry, what other methods they were using to help ensure their system was safe, what cybersecurity methods were in use, and what methods they would recommend for streamlining the software assurance process. The majority of respondents thought their company's software assurance standard was effective in developing critical software suitable for inclusion in safety-critical systems. Based on their answers to two questions, participants were of the opinion that getting the requirements correct would aid in streamlining the process; their top recommended assurance method was to formulate correct requirements.

Phase 1 also assessed a variety of assurance methods, including dynamic analysis (black box and white box testing); static analysis (syntax checks, formal mathematical proofs of correctness, Fagan inspections, human reviews, and formal methods); service history; software metrics and reliability models; hazard analysis; model-based development; assurance cases; entrance and exit criteria; and incremental integration of components. Phase 1 concluded that DO-178B and DO-278 have served the community well from a quality perspective—avionics software is relatively free from errors, albeit quite expensive. There is no evidence to support the necessity to radically change how DO-178B is applied in the community. Nevertheless, there are technical advances that could help reduce the high costs associated with critical systems software development. Some of these methods appeared promising and warranted further study in Phase 2. These methods included hazard analysis; human reviews; model-based specification and analysis; and architectural modeling and analysis.

The goal of Phase 2 was to conduct an in-depth study of each method listed and to collect information regarding how each approach helps to streamline the certification process and which approaches are best suited for commercial off-the-shelf (COTS) and legacy software.

Research conducted in Phase 2 provided evidence that the hazard analysis technique, Systems Theoretic Process Analysis (STPA), and perspective-based reviews of the requirements can streamline the software assurance process. This is achieved by finding errors during the requirements phase when changes can be made relatively easily as compared to later stages of development. STPA is a technique that can be used when COTS and legacy software are also part of the system. The interfaces between subsystems must be known and can be controlled so that they do not cause a hazard.

Model-based specification and analysis tools can help practitioners streamline the process of requirements assurance by capturing high-level requirements and safety constraints; tracing them to lower level or subsystem requirements; developing a model of those requirements; and allowing for change so that the requirements match the intended behavior of the system.

Using modeling again in the architecture assurance process can further streamline the software assurance process by analyzing and testing the models before any architecture is built. The models can be perfected through an iterative process in which requirements and safety constraints are compared against a model of the architecture.

For Phase 3, to demonstrate potential improvements to the requirements assurance process, a flight guidance system (FGS) was analyzed using STPA. The results of that analysis were compared to another hazard analysis technique, namely fault tree analysis (FTA). The comparison demonstrated that the STPA found three times more safety issues with the requirements than the FTA and provided more specific recommendations to mitigate those issues. To further assess the applicability of STPA as an assurance alternative, it would be desirable to team with an aviation partner to assess STPA in conjunction with ARP 4761 on a project currently in development.

In, Phase 3, studies that Rockwell Collins performed on cost benefits of model-based development were evaluated. In summary, the interactive refinement of the natural language requirements, the specification model formalizing these requirements, and the design model serving as a basis for the development helped provide a design model essentially free from faults. Consequently, any requirements-related problems were identified and resolved early, with the resultant implementation derived from the design model essentially fault-free. Therefore, the normally costly testing process was event-free, leading to significant schedule and cost savings. To accelerate adoption of model-based techniques and to reduce the chances of bad outcomes (and the possible delay or outright rejection of promising techniques), modeling notation independent guidance documents on modeling methodology should be developed.

In addition to the cost savings, the formal modeling positioned the Rockwell Collins team to rapidly respond to requirements changes arriving late in the development process. Rather than updating the models and rerunning a large number of test cases, the team could update the specification model (i.e., accept the requirements change), rerun the verification, identify needed design model changes, modify the design model to meet the new requirements, and finally verify that the new design model met the changed specification model. This process, based on formal verification, was far more cost effective than any based on testing of the design model. Had the requirements change led to the modification of possibly thousands of tests, these late changes would not have been cost effective.

The revisions in DO-178C and DO-333 allow for expanded use of formal techniques; one way in which formal techniques could be cost effectively deployed would be to replace or augment unit-testing efforts. A French team has successfully pursued this direction at Dassault-Aviation and Airbus. Their use of formal verification in lieu of testing has been cost effective in both organizations. It significantly cut costs in maintenance activities for which costly modifications of tests and retesting have been replaced by re-verification. Although formal verification holds

great promise, there are numerous ways the formal verification efforts can provide misleading or outright erroneous results (e.g., through misuse of abstractions or assumptions in the verification process). These issues are inherent in formal verification (independent of tools and modeling notations). Generic guidance should be developed on how to use formal verification techniques effectively.

Finally, the introduction of automation to replace manual processes may reduce the "collateral validation and verification" occurring as skilled engineers (versus tools) address various tasks. The negative effects, if any, of replacing humans with automation are not well-known and should be investigated further.

1. INTRODUCTION

Incomplete, inconsistent, ambiguous, and changing requirements have for decades been a problem in the software industry. In 1987, Fred Brooks pointed out this serious problem:

> "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements…No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later" [1].

Most software mistakes are introduced during requirements analysis, with the majority not being found until the latter stages of the development project. Studies have shown that the cost of correcting a requirements error grows dramatically the later in the product life cycle it is corrected [2–5]. It is also known that mistakes made in the requirements stage are more likely to affect the safety of a system than design and coding mistakes introduced in the design and implementation stage of development [6, 7]. Therefore, there are significant safety improvements and cost savings to be realized by improving the requirements elicitation processes to ensure the right requirements are found. Adopting proper requirements notations will ensure that the requirements are captured in a complete, consistent, and unambiguous manner.

Several early efforts to address this problem in the context of the avionics industry through modeling showed great promise. For example, to clarify and rigorously capture the requirements for the A-7 Corsair aircraft [8], a team led by David Parnas developed the Software Cost Reduction (SCR) method [9, 10]. Since then, this work has been extended into the CoRE methodology by the Software Productivity Consortium [11] and used to specify the avionics requirements on the Lockheed C-130J [12].

At the NASA Formal Methods Workshop in 2013, John Rushby pointed out that all commercial avionics software incidents reported to date are due to integration problems caused by incorrect component-level requirements. Specifically, the components in the software/systems architecture, even when working as required (according to the component-level requirements), do not meet the requirements at the higher level [13, 14]. Clearly, addressing the requirements discovery, requirements allocation, and requirements capture problems is critical from safety and cost-savings perspectives.

The "Alternate Approaches to Software Assurance" study, documented in this report, was a three-phase study undertaken to evaluate the current state of software assurance processes and to propose alternative approaches, with the potential to streamline the process and reduce the assurance costs without compromising safety. Phase 1 included querying aviation and safety industry personnel to gain insight into current uses of assurance techniques and how they felt DO-178B and DO-278 were serving their industry. Phase 1 also assessed a variety of assurance methods. Phase 2 conducted a study of a variety of promising techniques identified in Phase 1 and concluded that Systems Theoretic Process Analysis (STPA); model-based specification and analysis; and formal verification techniques merited further in-depth study in Phase 3. Phase 3 conducted in-depth studies of these approaches, provided a summary of the available evidence of effectiveness and cost, and provided recommendations for necessary further studies.

Phases 1–3 of the study are summarized in sections 1.1–1.3 and described in detail in sections 2–4.

1.1 PHASE 1: FIND ALTERNATIVE APPROACHES TO DO-178B AND DO-278

In Phase 1 of the study, the authors were tasked by the FAA to perform research on alternative approaches to software assurance methods, both process based and not. RTCA/DO-178B [15] specifies software assurance processes for developing software in airborne systems and non-airborne portions of communication, navigation, surveillance, and air traffic management (CNS/ATM) systems. DO-178C was released in the middle of this research and, therefore, was considered only at the end of the research during Phase 3. A report by the National Academies Press titled, "Software for Dependable Systems: Sufficient Evidence?" [16], sparked a debate on how software assurance should be achieved in industry. The report raised questions as to the cost effectiveness and difficulty of process-based approaches. The authors of this report concluded that the National Academies Press report lacked appropriate discussion of approaches to software safety in other industries. Other types of assurance methods were not mentioned. Only testing and formal methods were presented as alternative approaches, but there is an abundance of data on the efficacy of structured code walkthroughs. The problem with the solutions suggested (testing and formal methods) is that they are primarily useful only for showing consistency of the code with the requirements. The report fails to mention how people should deal with software requirements flaws, although it does begin by stating these flaws are a major problem.

Phase 1 work focused on three areas: an examination of alternative methods, a comparison of aerospace industry standards to the standards of other safety-critical industries, and a poll to query aviation industry personnel on their experience with DO-178B and DO-278. Phase 1 concluded that DO-178B and DO-278 have served the community well from a quality perspective—avionics software is relatively free from errors, albeit quite expensive. There is no evidence to support the need to radically change how DO-178B is applied in the community. Nevertheless, there are some technical advances that could help reduce the high costs associated with critical systems software development. Some of these methods appeared promising and warranted further study in Phase 2. These methods include hazard analysis; human reviews; model-based specification and analysis; and architectural modeling and analysis.

1.2 PHASE 2: TECHNICAL ADVANCES TO STREAMLINE THE SOFTWARE DEVELOPMENT PROCESS

Although the initial goal of this research was to find alternative approaches to software assurance processes DO-178B and DO-278, there was a change in direction for Phase 2. The findings from Phase 1 did not highlight any alternative approaches that could replace DO-178B or DO-278. The goal of Phase 2 was to conduct an in-depth study of techniques that warranted further study from Phase 1 (including hazard analysis; human reviews; model-based specification and analysis; architectural modeling and analysis) and collect information regarding how each approach helps in streamlining the certification process and which approaches are best used for commercial off-the-shelf (COTS) and legacy software. The FAA also requested coverage criteria be evaluated as a possible method for streamlining the assurance process.

The investigation found evidence that STPA and perspective-based reviews of the requirements could streamline the software assurance process. This could be achieved by finding errors during the requirements phase when changes can be made relatively easily as compared to later stages of development. STPA is a technique used when COTS and legacy software are also part of the system. The interfaces between subsystems must be known and able to be controlled so they do not cause a hazard.

Model-based specification and analysis tools could also help practitioners streamline the process of requirements assurance by capturing high-level requirements and safety constraints, tracing them to lower level or subsystem requirements, developing a model of those requirements, and allowing for change so that the requirements match the intended behavior of the system.

Using modeling again in the architecture-assurance process could further streamline the software-assurance process by facilitating analysis and testing of the models before any architecture is built. The models can be perfected through an iterative process in which requirements and safety constraints are compared against a model of the architecture. The ongoing System Architecture Virtual Integration (SAVI) Project is investigating this issue from a systems engineering perspective. The SAVI project is not focusing specifically on the software aspects of the aircraft, but instead takes a broad view of the modeling and virtual integration issues, including the full aircraft system.

The authors found that coverage criteria did not streamline the software-assurance process. There is considerable uncertainty regarding how good modified condition/decision coverage (MC/DC) really is as an exit criterion and whether the benefits warrant the cost. There is an ample opportunity for unaware contractors to accidentally (or deliberately) structure their code to make it easier to satisfy MC/DC; unfortunately, the resultant test cases may be highly inefficient. Until a better method is developed, the authors would recommend the continued use of MC/DC. However, the system reviewers should be directed to look at how MC/DC test cases are determined. The auto generation of test cases that specifically meet MC/DC should be viewed with certain suspicion until the approach to generation is fully explained and understood. The mechanism for determining failed/passed status should be fully explained and preference must be given to mechanisms that make comparison to at least some intermediate variable if the implementation is not strongly inlined.

1.3 PHASE 3: FURTHER INVESTIGATION OF TECHNICAL ADVANCEMENTS

The results from the first two phases indicated that the best method for streamlining the software assurance process is to get the requirements correct at the beginning of the development process. The participants from the Phase 1 study indicated that requirements analysis was one of the most effective methods for ensuring safe software and that getting the requirements correct would aid in streamlining the process. In Phase 2, the research showed STPA, model-based development, and architecture assurance exhibited promise in aiding the requirements assurance process and could potentially help those involved avoid costly re-work and retesting by catching errors early in the development.

The research from the first two phases directed the team to further focus on STPA, model-based development, and formal verification as a means to streamline DO-178B and DO-278 while

maintaining or enhancing safety. Although these methods have been around for some time, there have been advances in model-based development and formal verification that deemed their reassessment worthwhile. The Phase 3 work also highlighted how STPA can catch more system and software errors in the requirements than other hazard analysis techniques (i.e., fault tree analysis (FTA) and analysis of COTS and legacy software). The analysis demonstrated how STPA could be applied to a flight guidance system (FGS) and how hazard causes could be mitigated. The research also focused on the cost savings that were realized by Rockwell Collins through their use of model-based development and by Airbus through their use of formal verification on their projects.

2. PHASE 1 RESEARCH

2.1 DOCUMENTS AND STANDARDS REVIEWED

2.1.1 National Academies Press Report: "Software for Dependable Systems: Sufficient Evidence?"

This report is too limited for it to be useful as guidance on software assurance and certification. Only a narrow view is expressed and nothing is included about system safety. Further, the recommendations are vague and limited in scope.

Within the report, one of the problems is the use of the term "dependability" without a clear definition provided. The definition provided in the report, "A system is dependable when it can be depended on to produce the consequences for which it was designed, and not adverse effects, in its intended environment" [16], is circular.

In the report, "dependability" as a quality includes many different system properties, some of which are conflicting and often require tradeoffs. Mixing up all qualities into one general one makes any true statements about it too general to be useful.

The lack of clarity regarding the topic of the report is reflected in the conclusions and recommendations. The FAA certifies airworthiness by considering safety goals, not other system goals (e.g., achieving a reduction of 10% fuel consumption). However, safety and other system goals are not separated in the report and instead are lumped under the catch-all term "dependability." Although the report rarely mentions safety and does not include any safety engineering concepts, such as hazard and hazard analysis, almost all examples of dependability involve safety incidents.

A second important limitation of the report is that it considers only formal methods and testing as the choices for making systems dependable. All the other types of software assurances techniques, some of which have been found to be even more useful and effective than these two, are not mentioned.

Some claims in the report used to support the conclusion do not appear to be true. The following paragraphs provide some of the examples and discussions regarding the claims.

The statement "A second major class of problems arises from poor human factors design" is true, but human factors issues in the report are limited to bad user interface design. A much larger class of errors arises from the design of the software functions in a way that induces human error, such as "mode confusion" or "clumsy automation" [17, 18]. No solutions for these problems are mentioned in the report.

Regarding the claim "Much of the benefits of standards such as DO-178B … may be due to the safety culture that their strictures induce," the report does not define "safety culture." The usual definition involves the values and principles in an industry or organization in which decision-making about safety is based. It is the safety culture that underlies the standards that are written, not vice versa. DO-178B reflects the safety culture that exists in the industry; it does not create it:

> "Any component for which compelling evidence of dependability has been amassed at reasonable cost will likely be small by the standards of most modern software systems. Every critical specification property, therefore, will have to be assured by one, or at most a few, small components" [15].

This argument may be due to the fact that the formal methods proposed by the report are extremely expensive and difficult to use. However, in most systems, there are large parts of the software that are critical and need to be included in any certification arguments. Dependability is not defined, but aircraft safety cannot typically be assured by one or even a few small components.

2.1.1.1 Conclusion of the National Academies Press Report

The claim from the report that "More data are needed about software failures and the efficacy of development approaches" [16] is accurate, but the referred data about the efficacy of formal methods, which are assumed to be cost-effective in the report, are not supported with any references or citations.

2.1.1.2 The National Academies Press Report's Recommendations

This section presents some of the National Academies Press Report's recommendations and assertions [16] in bullets, followed by a discussion on the recommendation.

- "Make the most of effective software development technologies and formal methods. A variety of modern technologies—in particular safe programming languages, static analysis, and formal methods—are likely to reduce the cost and difficulty of producing dependable software."

The previous statement is a hypothesis, but there is no evidence in the report beyond hand waving to support this conclusion—including no cost data nor any data about difficulty of use. In fact, Leveson, one of the authors of the report, and her students have conducted several carefully designed scientific experiments to compare formal methods and specifications with less formal ones. In each of these experiments, the formal methods turned out to be more error-prone, and

less readable and understandable [19]. In these experiments, even computer science graduate students who were highly trained in the mathematical techniques that were used made more errors when they were given formal specifications. Published proofs of software correctness by qualified mathematicians later revealed that the previously mentioned claim was incorrect [20]. It is not practical to conduct the certification process based on techniques that only highly qualified mathematicians can read, understand, or use.

The report mentions the following set of assertions:

- Follow proven principles for software development
- Take a systems perspective
- Exploit simplicity—developers and customers must be willing to accept the compromises it entails

The set of simplicity argument in the report is not realistic. There are two types of complexity (as noted by Brooks [21]): "essential complexity" (stemming from the problem being solved) and "accidental complexity" (which arises in the solution to the problem). Essential complexity comes from the problem, not from the solution or software design. The software design can add complexity, but reducing essential complexity is a matter of the system engineers foregoing functionality. We use software precisely because it allows us to do things that we cannot do without it. There is no going back to the aircraft designs of the 1970s.

- "Make a dependability case for a given system and context: evidence, explicitness, and expertise. A software system should be regarded as dependable only if sufficient evidence of its articulated properties is presented to substantiate the dependability claim. This approach gives considerable leeway to developers to use whatever practices are best suited to the problem at hand. In practice, the challenges are sufficiently great of developing dependable software that developers will need considerable expertise and they will have to justify any deviations from best practices."

This recommendation provides no useful information about how to conduct software assurance. It seems to indicate that there is lot of freedom for the developers to do anything they want.

- "Demand more transparency, so that customers and users can make more-informed judgments about dependability."

The only problem with this recommendation is that formal methods, which the report supports, are far from transparent and almost impossible to understand, even for engineers who are experts on the system being built. Without such understandability, there is no way to check the proofs and the "dependability case" being proffered.

- "Make use of but do not rely solely on process and testing."
- "Base certification on inspection and analysis of the dependability claim and the evidence offered in its support."

The report does not say how to conduct the previously mentioned processes. It is worth noting that nowhere in the report does the word "hazard" or "hazard analysis"—or any of the basics of system safety engineering—appear. This seems like an important omission for a report covering safety. The limitations of such assurance cases are discussed later in this report.

- "Include security considerations in the dependability case."
- "Demand accountability and make it explicit. Although there is a need to deploy certifiably dependable software, it should always be made explicit who or what is accountable, professionally and legally, for any failure to achieve the declared dependability."

None of this is within the purview of the FAA. Elsewhere in the report, it says:

> "No software should be considered dependable if it is supplied with a disclaimer that withholds the manufacturer's commitment to provide a warranty or other remedies for software that fail to meet its dependability claims" [16].

This is a legal issue, not a technical one. The law in the United States explicitly excuses software from warranties, explicit or implied. Dependability claims are not usually made for any software—and it is not at all clear what such claims might be.

2.1.1.3 Implicit Recommendations

This section presents some of the National Academies Press Report's implicit recommendations, which were not called out as recommendations but were included in the executive summary. The following bulleted items are the implicit recommendations, followed by a discussion on the recommendations.

- "People are part of systems and, therefore, an estimate of the probability that they will behave as required should be part of the evidence for dependability."

There are no such estimates available in the literature for the probabilities associated with the types of mistakes humans make in relation to aircraft, and it is not possible to estimate those probabilities. Aircraft (and the software they use) have to be designed to reduce human error. Simply providing probabilities for human errors (even if such probabilities existed) would not make aircraft safer.

- "It is important to distinguish the requirements of a software system, which represent properties in the physical world, from the specifications of a software system, which characterize the behavior of the software system at its interface with the environment."

Software requirements are the required behavior of the software at its interface with its environment. They are not two different things.

2.1.1.4 Conclusions

The report does not seem to have any useful recommendations pertaining to software assurance beyond "using formal methods." There are no data provided (nor do the authors know of any that exist) that substantiate the claims made that formal methods are cost-effective. In fact, the small amount of data that do exist show just the opposite.

The only techniques for software assurance (i.e., testing and formal methods) mentioned are too limited. In fact, the one technique for which there are abundant scientific data of its efficacy is structured code walkthroughs, but it is not mentioned, nor are other types of assurance methods.

The most glaring omission is the failure to mention any of the approaches to software safety used in other industries. The problem with the solutions suggested (testing and formal methods) is that they are primarily useful only for showing consistency of the code with the requirements. The report fails to mention how people should deal with software-requirements flaws, although the report starts by saying these flaws are the major problem.

The word "safety," in the context of system safety engineering, is avoided in the report, and "dependability" (which is never really defined but seems much broader) is substituted. Almost all the examples used, however, are safety problems; however, the analysis of some of these problems appears flawed. For example, the analysis of the Warsaw A-320 accident provided in the National Research Council report is incorrect. It is not proper to say that the software specification did not reflect the system requirements in that case. Rather, the system requirements were wrong, or, at the least, they were unsafe. The engineers did identify the hazard (requirement) of not turning on the reverse thrusters while in flight. A hazard analysis and safety design technique might have identified the hazard. The decisions involved require difficult design trade-offs by system engineers. For example, one solution is to let the pilot override the software. The critical decision to make during system requirements is who has final authority—the software or the crew. This is not a software design or software analysis problem. The software can be "dependable" and can still be performed incorrectly if the decisions implemented in the software are as per the system engineering and the system and software requirements.

Certification in some industries that use system safety (such as military aircraft) involves:

- Identification of hazards and potential causes
- Evaluation of what has been done to eliminate or control hazards and potential causes
- The derivation of software safety requirements from system safety requirements
- Human factors analysis regarding how the design will prevent or control identified (through hazard analysis) operator behavior that could lead to losses

In taking this alternative approach, which is not mentioned in the report, standard Functional Hazard Assessment (FHA) for aircraft must be augmented because software does not fail in the way that hardware does and statistical reliability methods do not apply. System hazard analysis techniques can be used to identify any potential software contributions to system hazards, which will not be a failure probability but an unsafe behavior. Those unsafe behaviors can then be used

to identify required software safety requirements. Using those requirements, the software needs to be designed to ensure that such behavior is not allowed. Assurance must then be provided that the software satisfies the behavioral safety requirements. Most software assurance methods can be used to accomplish this goal, including formal methods, if appropriate. But the use of formal methods alone, as suggested in the report, cannot prevent these types of problems.

2.1.2 Evaluation of Other Industry Software Assurance Standards

RTCA/DO-178B and RTCA/DO-278 are software assurance standards used in the aviation industry for certifying software used in the airborne environment and non-airborne CNS/ATM systems. RTCA/DO-178B states that its purpose is to

> "provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements. These guidelines are in the form of: objectives for software life cycle processes, descriptions of activities and design considerations for achieving those objectives, and descriptions of the evidence that indicate that the objectives have been satisfied" [15].

This standard includes such processes as the software life-cycle process, software planning process, software development process, software verification process, software configuration management process, software quality assurance process, and certification liaison process. The standard also includes an overview of aircraft and engine certification and details of all the software life-cycle data.

The purpose of RTCA/DO-278 is

> "to provide guidelines for the assurance of software contained in non-airborne CNS/ATM systems. DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification, defines a set of objectives that are recommended to establish assurance that airborne software has the integrity needed for use in a safety-related application. These objectives have been reviewed, and in some cases, modified for application to non-airborne CNS/ATM systems. This document is intended to be an interpretive guide for the application of DO-178B/ED-12B guidance to non-airborne CNS/ATM systems" [22].

Part of the work performed during Phase 1 was to review other industries' standards and determine if there were approaches or methods called out in those documents that could be used as an alternative approach to RTCA/DO-178B and RTCA/DO-278. Three standards were reviewed by Safeware Engineering to determine if there were unique processes used by other safety critical industries that could enhance RTCA/DO178B and RTCA/DO 278. The standards were: IEEE 12207:2008—Systems and Software Engineering—Software Life Cycle Processes; NASA Software Assurance Standard: NASA-STD-8739.8; and ANSI/AAMI/IEC 62304:2006—Medical Device Software—Software Life Cycle Processes.

2.1.2.1 IEEE 12207:2008—Systems and Software Engineering—Software Life Cycle Processes

This international standard establishes a common framework for software life-cycle processes. According to the standards,

> "It applies to the acquisition of systems and software products and services, to the supply, development, operation, maintenance, and disposal of software products and the software portion of a system, whether performed internally or externally to an organization" [23].

IEEE 12207 provides a set of life-cycle processes, activities, and tasks for software that is part of a larger system and for standalone software products and services by including two major subdivisions. One subdivision provides a system context for dealing with a standalone software product or service or a software system. The other subdivision contains the software-specific processes for use in implementing a software product or service that is an element of a larger system. The software specific processes align with many of the processes detailed in DO-178B, such as the software requirements process, software design process, software integration process, software configuration management process, software quality assurance process, software verification process, and software review process. Other processes called out in IEEE 12207 are system life-cycle processes, such as the agreement process and the supply process, which do not provide any alternative approaches to software assurance.

Another difference between IEEE 12207 and DO-178B is conformance. The authors of IEEE 12207 realized that users of the standard may not need to use all of the processes called out in the standard. Users would be able to select a set of processes that are suitable for the project on which they are working. There are two ways in which implementation of this standard can be claimed: full conformance or tailored conformance. Full conformance claims can be made if all the requirements in the processes have been made and their outcomes documented. A tailored conformance is achieved by demonstrating that requirements for the processes, as tailored, have been satisfied using the outcomes as evidence. IEEE 12207, like DO-178B/C, is a recommended processes document that can be applied to any software development project. Both standards can likely be adopted to develop safety-critical software but do not have requirements/recommendations specific to safety analysis or safety verification contained in them.

2.1.2.2 NASA Software Assurance Standard, NASA-STD-8739.8

NASA's Software Assurance Standard NASA-STD-8739.8 approaches software assurance by looking at it from the acquirer's responsibilities, the provider's responsibilities, and dual responsibilities. The success of safe development of software depends on a well-written request for proposal and contract. It details the role of a software assurance manager who has approval authority. The standard applies to all software regardless of who developed it or when. It covers all life-cycle activities, including maintenance and retirement. Software assurance records are required and a minimum list is included. Section 7 of NASA-STD-8739.8 includes requirements for each of the software assurance disciplines, although these sections primarily hit the highlights and include a requirement to meet the criteria included in another standard.

As stated, it is not a standalone document and includes, by reference, the following standards:

- Government documents
  - NPD 2810.1 NASA Information Security Policy
  - NPD 2820.1 NASA Software Policies
  - NPR 1441.1 NASA Records Retention Schedules
  - NPR 7120.5 NASA Program and Project Management Processes and Requirements
  - NPR 7150.2 NASA Software Engineering Requirements
  - NASA-STD-8719.13 Software Safety Standard

- Non-government documents
  - IEEE 730-2002 IEEE Standard for Software Quality Assurance Plans

NASA-STD-8739.8 puts the emphasis on its Software Assurance Classification Assessment, Appendix A of the standard, which is used to determine the level of tailoring that might be allowed. The assessment assigns points to the system based on the components and the type of system.

Much of the process required by NASA is similar to that required by RTCA/DO-178B and RTCA/DO-278. The NASA-STD-8739.8 emphasis on a well-written contract will not work well in the aviation industry because the government is not the purchaser of the product and does not have a financial role in the product. In addition, by providing clear guidelines for when a product is eligible for tailoring, an assessment similar to the Software Assurance Classification Assessment in Appendix A of the standard could at least streamline any negotiations regarding tailoring required processes and assist the Designated Engineering Representative's (DER's) review of the system. It could also be used to determine which DER's experience best suits the project so that his or her criticism would encompass more engineering-based perspective.

2.1.2.3 ANSI/AMMI/IEC 62304:2006 Medical Device Software—Software Life Cycle Processes

IEC 62304:2006 is a standard that "provides a framework of life cycle processes and activities and tasks necessary for the safe design and maintenance of medical device software" [24] This standard has been derived from the approach and concepts of IEEE 12207 reviewed in section 2.1.2.1, but the standard does not include the system aspects, such as system requirements or system architecture.

IEC 62304 includes similar processes as DO-178B and DO-278: a software development process and a configuration management process, but it also includes a maintenance process that is much more prescriptive than DO-178B and DO-278. IEC 62304 requires a software maintenance plan; a document and evaluate feedback process; and a modification implementation process. Including a more detailed maintenance process into DO-178B and DO-278 is recommended.

2.2 POLL RESULTS OF ALTERNATIVE APPROACHES TO SOFTWARE ASSURANCE

An online poll was conducted to ask participants how they apply software assurance in their particular industry. The questions and responses are included in appendix A. Questions were written to gain insight into the participant's experience; job title and duties; the software assurance standards they use; standards they would recommend; software assurance methods they use and their assessment of those methods; their experience with implementation of cyber security; and their opinions regarding the benefits of third-party certification. The poll was open for 1 month. The responses were then compiled and analyzed.

Of 490 requests, 262 participants accessed the poll content. Of the 76 responses received, 64 participants' responses were useful for further analysis. Although the participant rate is not high enough, there is still valuable information that can be extracted from these data. The data presented here are from those 64 responses. Most of the participants were from the aviation industry. Within the pool of 64 people completing the questionnaire, job titles ranged from CEO to middle management to software design level engineers. Ninety percent of the participants had experience in the aviation industry, with defense and space systems a distant second and third, respectively. Fifty-five percent had more than 20 years' experience with software assurance and had worked on small-, medium-, and large-sized projects. Ninety-five percent of the participants' companies used DO-178B, and 35% used DO-278. It can be observed that the sum is more than 100% because companies could be using both standards. Eighty-three percent of the respondents thought their company's software assurance standards were effective in guiding the development of safe software. The top four methods used in the software assurance process were: testing, requirements analysis, hazard analysis, and code analysis. Most participants did not include cyber security as part of their software assurance processes. Almost half of the respondents thought the software assurance process increased cost, and 57% thought it reduced errors.

Figure A-4 in appendix A depicts the size of the projects on which the respondents had worked. Of the 27 participants that had worked on large projects, 63% and 59% had also worked on medium and small projects, respectively, as shown in figure A-4. Fifty-nine percent of those who had worked on large projects thought their software assurance standard's ease of use was moderate, and 22% of them thought it was difficult. Eighty-nine percent thought their software assurance standards were effective in developing safe software.

In terms of software assurance methods, those 27 participants that had worked on large projects thought testing, requirements analysis, hazard analysis, and static code analysis were the top software assurance methods used. Assurance cases and reliability modeling were the least used, at 7%. Eighty-five percent thought testing had a high or very high contribution to providing software that is safe, but 33% thought testing was a difficult method to use. Seventy-three percent thought requirements analysis had a high or very high contribution, and 31% thought this method was difficult to use. Hazard analysis and static code analysis were thought to provide a high or very high contribution, at 58% and 32%, respectively, and more than 40% responded that the ease of use was moderate. The data were similar for participants who had worked on small- and medium-sized projects.

The titles of the two individuals who did not know the size of the projects are director, engineering and manufacturing (with 5–10 years of experience), and certification engineer, with 11–15 years of experience.

More than 55% of the poll participants had more than 20 years of experience. The responses from each experience group were the same as the overall poll results in terms of contribution to safe software. For example, each experience group thought testing contributed greatly to providing safe software.

Fifty-three percent of the polled participants thought their software assurance standards were moderate in terms of ease of use. Thirty percent thought the standard was difficult, and 14% thought it was easy to use.

Of the 53 respondents who thought their company's software assurance standard was effective in developing safe software, 17% thought the standard was easy to use, 57% thought it was moderate, and 25% thought it was difficult to use.

Those participants polled considered testing, requirements analysis, and hazard analysis the highest contributing methods for facilitating safe software. Static code analysis was ranked below these three. Thirty-four percent of those participants responded that service history provided no, or few, contributions to software that is safe. Formal methods, assurance cases, and reliability modeling had a high percentage of responses for "not applicable," which could be interpreted as methods those respondents do not use.

Although 62% of the polled participants did not include cyber security as an element of software assurance, Safeware Engineering recommends that it be considered in the software assurance process.

In response to the open-ended question requesting recommendations for improving and streamlining the software assurance process, as shown in figure A-17 in appendix A, approximately 28% had no recommendations, 16% thought the requirements needed to be improved, and 16% believed increased training in software safety and software assurance would be important.

When the participants were asked if there were additional methods they would include in the software assurance process, 32% offered no additional methods, 27% wanted to ensure good requirements, and 14% recommended more testing.

2.2.1 Poll Summary

The majority of respondents thought their company's software assurance standard was effective in developing safe software. Although participants were positive towards the effectiveness of their standards, Safeware Engineering pointed out data and open-ended responses highlighting the importance of requirements analysis and the need to write complete and consistent requirements, beginning at the system level and continuing down to the software requirements. The data also showed that more than half the participants did not include cyber security as part of

the software assurance approach. Safeware Engineering recommends including cyber security as an additional software assurance requirement.

2.3 OVERVIEW OF ASSURANCE METHODS

Assurance can be defined as building confidence. But different types of confidence are possible and a clearer definition of confidence is needed. For example, assurance might be achieved by the software correctly implementing its requirements, but that will provide little confidence that the requirements are consistent with system engineering requirements and goals. Therefore, software assurance needs to be defined according to what is being assured.

Three important properties related to airworthiness that might be assured are that:

1. The software requirements are consistent with the system requirements.
2. The software implementation (code) satisfies the software requirements.
3. The software will execute without contributing to a system hazard or security breach.

Different assurance methods are capable of assuring certain software properties but not necessarily others. To achieve assurance of all the properties required for airworthiness, multiple assurance methods will be required. The key is to select a set that is comprehensive (i.e., it covers all of the properties of interest while not wasting resources through duplication).

The major types of software assurance methods are:

- Dynamic analysis
- Static analysis
- Quality assurance (defined here as conformance to specifications)
- Assurance of non-software life-cycle products
- Collecting service history

Assurance methods can also be classified as to whether they provide an argument for assurance of the property or whether they provide a measurement of some property (a metric) that is used to evaluate the existence of the required property.

2.3.1 Dynamic Analysis

Dynamic analysis involves executing the software to determine how it will behave. Various types of assurance methods are used to provide information or assurance while the software executes:

- Black-box and white-box testing–Black box testing uses test data generated only from the specifications (without knowledge of the internal structure of the program); white-box testing is based on the structure of the software design (i.e., the test data are derived by examining the program's logic). Although no one would execute critical code without testing it, there is, unfortunately, no way to exhaustively test software. In fact, only

relatively few parts of the software state space can be tested using reasonable resources (including time). Usually, a combination of white-box and black-box testing is used.

- Coverage analysis–Because only part of the software can be tested, coverage analysis is used to provide information about the completeness of the testing according to various types of coverage criteria. For example, one common criterion is to check how many paths through the program have been executed at least once. Another example is the MC/DC testing criterion included in DO-178B. Although there has been controversy over MC/DC testing (arising primarily because of its cost and the effort required), some studies have shown that it has uncovered errors not found by testing using other criteria [25]. Although coverage criteria are most often associated with white-box testing, they may also be applied to black-box testing—in this case with respect to the coverage of the requirements by the test cases.
- Automated test case generation–Generating test cases is difficult. Some ways to automate this process are appropriate, particularly ones that provide various types of specification or code coverage. Model-based specifications are especially amenable to automated test coverage from the specifications using various coverage criteria.
- Embedded assertions and other forms of monitoring run-time behavior–Monitors can either be external to the software or they can be embedded within the software. External monitoring (sometimes called auditing) may check the outputs; data passed between modules or processes; the consistency of global data structures; or the expected timing of modules or processes. Internal monitoring is most often implemented using assertions. Assertions are statements (Boolean expressions on the system state) about the expected state of the module at different points in its execution or about the expected value of parameters passed to the module. Assertions may take the form of range checks (that the values of internal variables lie within a particular range during execution), state checks (that specific relationships hold among the program variables), and reasonableness checks (that the values of inputs and outputs of modules are possible or likely). Assertions may be generated automatically when using some compilers and programming languages (e.g., Spark).

Unfortunately, writing effective code-level checks to detect software errors is difficult [26], and practicality usually limits the number of checks that can be made in a physical system constrained by time (i.e., hard real-time) and memory. Theoretically, such checks could be removed after testing and before the software is used operationally, but removing checks can significantly change timing and even software behavior, so they are often left in the operational code. External monitors are less intrusive, but it is very difficult to make such monitors independent, and they are limited in the types of errors they can detect.

Care needs to be taken to ensure that the added monitoring and checks do not cause errors. In a study of assertions added to detect software errors [7], the assertions added more errors than they detected. Recovery mechanisms may also be complex or error prone. A large percentage of the errors found in production software are located in the error-detection or error-handling routines, perhaps because they get the least use and testing.

Testing was highly ranked in importance by the poll participants.

2.3.2 Static Analysis

Static analysis can range from simple syntax checks to detecting nonconformance with the language rules to formal mathematical proofs of correctness (consistency between the requirements and the implementation of those requirements). Syntax checks are usually automated and performed routinely. Other types of static checks may involve looking for error-prone constructions or common errors (sometimes called Fagan inspections). Program structure checks involve generating graphs of the software structure and looking for flaws in that structure. Such checks can often be automated, although detecting more complex flaws usually requires human inspection. Module interface checks can detect inconsistencies in declarations of data structures and improper linkages between modules. All the static analysis techniques described so far are usually cheap (i.e., they can be performed by a compiler or simple tools). Although few carefully designed experiments have been performed to show the effectiveness of such checks, the low-cost usually makes them easy to justify.

A second type of static analysis technique involves human reviews. These reviews may be informal or formal. Formal reviews are usually associated with various documentation phases (e.g., Preliminary Design Review, Critical Design Review, etc.). After passing such reviews, software should be maintained under careful configuration control.

Two special types of human reviews have become very popular because of their effectiveness in finding errors: inspections (using checklists by technically capable people) and structured walkthroughs. Inspections and walkthroughs are time consuming and costly in terms of resources required, but they have also been found through experimentation to be very effective at finding errors. Structured walkthroughs in studies have been found to be particularly effective, identifying approximately 70%–80% of errors. Using walkthroughs allows most errors to be found early (before unit testing), when they can most easily and cheaply be fixed; the cost of doing the walkthrough is usually justified if total life-cycle costs are considered. If the software is too complex for walkthroughs, the implication is that it is too complex for any human to understand. In this case, a redesign should be considered.

The poll participants ranked static analysis methods as making a moderate contribution to safe software.

A final and more controversial form of static analysis is often labeled “formal methods” and involves formal mathematical analyses of the software. These analyses may be fairly simple and easily performed, and others may require PhD-level mathematical expertise. Some can be at least partially automated. One of the simplest automated analyses to perform is event sequence checking, for which expected event sequences are specified using a mathematical notation, and the event sequences that occur are compared with the specification of legal sequences.

Symbolic execution is another type of formal method for which the code is analyzed to identify the function being computed, which can then be compared to the specified function. Symbolic execution is related to more informal walkthroughs, for which symbolic values (rather than actual values) are tracked throughout the code. Some parts of symbolic execution can be automated, although the process requires the identification of invariants for loops. Over the years, tools have been designed to assist with the identification of invariants. One problem with

symbolic execution is a limitation in the complexity and size of the software that can be automated. Each path through the software must be examined. This can be impractical for software with large numbers of paths through the logic.

Model checking is a technique for specifying the software logic in the form of a binary decision diagram or logic formula and checking the model for various properties. Model checking on computer hardware has reputedly been very successful, but model checking of industrial software has achieved only limited success. One problem is an explosion of the state space, which may require simplifying the system and limiting the value of the information returned from the analysis. Another limitation involves analyzing properties of real numbers rather than simply Boolean values. A final limitation is identifying what the model should be checked for. Common properties, like deadlock, can be checked, but it is difficult to find general logic flaws.

The most rigorous form of formal methods uses theorem proving to validate various properties of the software, particularly the mathematical equivalence between the code and a formal specification of the required behavior. Although theorem-proving approaches have been popular with academics and those with PhDs in computer science, little use in industry has occurred since the idea was first proposed in the 1960s. A large part of the problem is the difficulty in writing the specifications in a formal mathematical language. Such specifications have been found to be very error-prone in practice and experimental evaluation [19, 27, 28], and errors have been found in published software proofs [29]. In addition, the discrete math languages required for the formal specification are usually not known by engineers. Therefore, it is difficult for application experts to determine whether the behavior specified is that required at the system level. The major problem not solved by any of these methods is specification flaws, which is the most common cause of errors in operational software.

There have been no carefully designed and controlled experimental studies to show that the difficulty and cost of using formal methods is justified by the results.

The poll results showed little support for, or use of, formal methods.

2.3.3 Quality Assurance

Quality assurance usually involves checking conformance of the software to various types of standards and accepted practices. Standards may be external or internal to companies. External (industry) standards tend to be limited to what everyone can agree to include in the standards. Frequently, these standards are not updated with the advances in the software development and verification tools and techniques because of the effort and time required to update them. Company standards are more easily updated and may require less general consensus, making them stronger than the industry standards. Company standards often incorporate extensive experience gained about the particular application (such as how to avoid common software errors found over time). Because of their value to the company, internal checklists and standards may be considered proprietary by the company involved.

2.3.4 Verification and Validation of Non-Software Life-Cycle Products

This assurance activity involves examination of other life-cycle products, such as specifications, user manuals, and other such documentation. Safe use and maintenance of software depends on these products. They also need to be subjected to assurance activities.

2.3.5 Service History

It is assumed that using service history as an assurance method means that if software were successfully used for a different system in the past, it can then be assumed to be safe in the current system. There are two problems with this assumption.

The first problem is the assumption that if software was safe in a different system, it will be safe in the current one. In fact, reused software has been at the heart of the majority of software-related spacecraft accidents [30]. If there are changes in the environment in which the software is embedded or on the hardware on which it executes, then safe execution in a different environment does not provide any assurance for a new environment.

In addition, software is continually being updated and changed, representing the second problem. The problem that occurred with flash memory on the Mars Rovers was part of an operating system (VxWorks®) that had been used successfully for hundreds of systems. However, small changes and improvements had been continually made on this software, one of which led to the Rover software problem.

Note that the poll participants ranked service history relatively low with respect to contributing to safe software.

2.3.6 Software Metrics and Reliability Models

There has been a long quest for metrics to provide assurance about properties of software. Metrics are commonly used for hardware and the same type of measurement would prove extremely useful for software. The great differences between hardware and software, however, have made this quest difficult.

The two general types of metrics that have been proposed are product and process. Product metrics measure some aspect of the software directly to provide information regarding how the software will behave in practice. Process metrics are more indirect: they measure aspects of the development process to try to predict properties of the software itself, assuming that the process will be reflected in the product.

If the property of interest could be measured directly, the problems would be simpler. Usually, however, a prediction of a property of interest is made by measuring it only indirectly. To predict anything using indirect measurement, a model of the relationship between the predicted variable and other measurable variables is needed. As such, there are three assumptions that underlie the use of software metrics [31]:

1. An accurate measurement of some property of software or the software process can be made.
2. A relationship exists between what we can measure and what we want to know.
3. This relationship is understood, has been validated, and can be expressed in terms of a formula or model.

Few metrics have been demonstrated to be predictable or related to product or process attributes. However, various types of metrics have been proposed:

1. Code: static and dynamic
2. Programmer productivity
3. Design
4. Testing
5. Management (cost, duration, staffing)

Because this report relates to approaches to certification of airworthiness, only 1, 3, and 4 are discussed. Productivity and management metrics are more relevant to schedule and budget concerns than to airworthiness.

2.3.6.1 Complexity and Software Design Metrics

Because of the ease of proposing a metric and the lack of emphasis on validation in software research, numerous metrics have been proposed. One set of metrics, called software complexity metrics, is claimed to somehow measure complexity, assuming that higher complexity will result in more errors being made. One of the earliest of these code-level metrics was Halstead's metrics, which he said were based on information theory. It appears that the basic assumption underlying Halstead's metrics is that most programs are produced by programmers going through a process of mental manipulation of the unique operators and operands so that there is an implicit limit on the mental capacity of a programmer [32]. Additionally, programming and programming languages were very different in the 1970s. Criticisms of this theory and approach to metrics have focused both on the theory itself and the reported empirical testing of the metrics that failed to support them [33–36]. The few early empirical studies, which reported good results, were done mostly by Halstead. They have been criticized on the grounds of small sample sizes and very small programs. It is notable that anyone still gives them any credence.

One popular complexity metric, McCabe's cyclomatic complexity [37], has some intuitive rationale. McCabe hypothesized that the difficulty of understanding a program is largely determined by the complexity of its control flow graph. To calculate a metric based on this hypothesis, the program's control flow is first changed into a connected graph. The cyclomatic number is the number of linearly independent paths in the graph or the number of regions in a planar graph. McCabe created a company that produces tools to create the graphs and compute

the cyclomatic complexity from code modules. He recommended a maximum cyclomatic complexity of 10. High control flow complexity does in fact make software harder to understand and could result in increased errors, but there are more dangerous types of complexity that are not included in this metric—for example, data structure complexity and algorithmic complexity. Often, in the attempt to reduce cyclomatic complexity, these other dangerous forms of complexity can increase. McCabe claims that cyclomatic complexity is a measure of testing difficulty and of the reliability of the modules.

In the 1970s, Larry Constantine [38, 39] came up with the concepts of coupling and cohesion, which is the degree to which software modules depend on each other. The assumption made is that high coupling between modules and low cohesion between the statements in a module make understanding the software, and therefore maintenance, more difficult and more error-prone. Various metrics have been suggested to measure coupling and cohesion. One simple one is to count the number of parameters passed between modules. The assumption is that understanding modules with a large number of parameters will require more time and effort, and that modifying such modules is likely to have side effects on other modules.

A variant on simply counting the number of parameters (e.g., fan-in, fan-out) is examining the calling structure among the modules. Fan-in is the number of modules that call a particular module. Fan-out is how many modules are called by the module. High fan-in means that many modules depend on this module, and high fan-out means the module depends on many other modules.

More complex types of coupling and cohesion metrics have been proposed, such as data bindings, which check not just whether parameters are passed between modules but how they are used in the modules. An example of a cohesion metric is the construction of the flow graph for the module and the determination of how many independent paths of the module go through the different statements. If a module has high cohesion, most of the variables will be used by statements in most paths. The highest cohesion is when all the independent paths use all the variables in the module.

A case can be made that many of these metrics provide useful information. The problem is determining which of the metrics is useful for what purpose. Static metrics of this type can only be a weak indicator of code quality. Consider the case in which a module starts the testing process with a particular complexity metric. As the code is exercised and errors are found and removed, the static complexity metric probably will not change unless the whole structure of the module is changed. Clearly the code is getting better (at least in terms of fixing errors), but structural complexity measures may not change. At best, such structural metrics can provide some clues as to which modules should be given the most emphasis in testing.

Another limitation of such metrics is that they ignore many of the factors known or suspected to have a large impact on software quality and reliability, such as the ability of the programmers, specification change activity, thoroughness of design documentation, computing environment, application area, particular algorithms implemented, characteristics of users, etc.

An important limitation occurs when standards implement requirements for particular metrics, such as requiring a McCabe cyclometric complexity value of less than 12. Such rules are easy to get around by introducing more obscure and harmful complexity to minimize the properties measured by a particular complexity metric. For example, control flow complexity can be reduced by using more complex data structures.

2.3.6.2 Metrics Derived From Dynamic Analysis (Executing Software)

When using dynamic analysis to generate the values of metrics, the goal is usually to either estimate the number of bugs left in the code or to estimate future failure times (operational reliability). Each is considered in turn.

One way to estimate the number of bugs remaining in software is to use the number already found during testing. Failure count models take the numbers found during a certain time interval and interpolate to estimate the total number of errors remaining. Basically, the failure counts are assumed to follow a known stochastic process with a time-dependent discrete or continuous failure rate.

In fault seeding models, the basic idea is to put a known number of faults in a program to try to estimate the number of unknown faults. The software is tested and the observed number of seeded and non-seeded faults detected are compared to estimate the number of non-seeded faults in the program. The idea is similar to the approach of tagging fish and then going fishing, comparing the number of tagged versus untagged fish, and using that number to estimate how many fish live in the lake.

Both failure count and fault seeding models rest on underlying assumptions that put the results in doubt when applied to real systems. For example, error seeding assumes that seeded faults are equivalent to inherent faults in terms of difficulty of detection. The truth of this assumption is very doubtful. To create enough seeded faults, software is often used that changes operators (e.g., a "+" becomes a "-") or inverts the order of two statements. These errors are easily found and would be found by any simple testing. The errors that remain after normal testing are usually more subtle. One of the controversial assumptions underlying these techniques is that there is a direct relationship between the characteristics of exposed and undiscovered faults. Another assumption is that the unreliability of the system will be directly proportional to the faults that remain, and there is a constant rate of fault detection. Another problem that arises is that seeding faults can mask the discovery of real faults in the program so that removing the seeded faults at the end of testing uncovers many other faults that were hidden by the seeded faults.

One question underlying the premise of this whole approach to metrics is: What does an estimate of remaining errors mean? Most reliability goals are stated in terms of operational reliability, not the number of remaining faults in a program. Another question is: Is software that has an estimated three remaining faults safer than software with five remaining faults? Establishing a direct relationship between fault density (i.e., the number of faults in the code), even if it could be determined, and operational software behavior, or "failures" is difficult because of the way software errors are distributed in the code, their differing severity, and the probability of the combination of inputs necessary to trigger a fault.

The alternative is to estimate failure rates or future interfailure times. There are two basic approaches for this estimation: input domain models and reliability growth models. Input domain models estimate program reliability using test cases sampled from the input domain. A set of test cases is generated from an input distribution representing the operational usage of the program and an estimate of program reliability is obtained from the failures observed during the execution of those test cases. Because of the difficulty of obtaining the input distribution, the input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate is made of the conditional probability that a program is correct for all possible inputs given that it is correct for a specified set of inputs. This process relies on the assumption that the behavior of a test case is the same as that of other points close to the test point. That assumption is suspect for software, which often has boundary areas and other unique points of failure.

An alternative is reliability growth models, which try to determine the future time between failures. Like the other reliability modeling techniques, this approach is based on standard techniques used for hardware. The question is whether assumptions about hardware failure apply to software. In general, reliability growth modeling requires executing the program using what is assumed to be an accurate profile of the operational usage environment and the inputs that will occur. When a failure occurs, the underlying error is found and removed (usually assuming this occurs in zero time) and execution is resumed. A standard probability distribution is used to then predict future times to failure. Basically, these techniques examine the sequence of elapsed times from one failure to the next to predict future failures.

There is a basic assumption underlying all of the reliability growth models that the occurrence of software failures is a stochastic process. The behavior of software, unlike hardware, which has random failures, is not stochastic. Software always produces the same output given the same input (assuming the computing platform is constant), unlike hardware, whose behavior can vary over time because of degradation and wear out. Several rationales have been advanced for applying stochastic modeling processes to software. The most common justification is the assumption that the selection of inputs during execution is unpredictable (random) and therefore stochastic with respect to whether an input is selected that will lead to a software failure or one that will not. Although this assumption is widely accepted in the reliability modeling community, there is evidence that it does not hold in practice. Examples of error bursts in aircraft flight have been detected, perhaps because the aircraft gets into states that trigger boundary cases in the software [40].

In practice, different reliability growth models tend to give varying results for the same data. There is no way to know *a priori* which model will provide the best results in a given situation. In general, many of the assumptions required for the mathematical techniques used in the models may be invalid for all or many programs. Not all models require the same assumptions, but none eliminates all of them. Examples of these assumptions are that software faults (and the failures they cause) are independent; the test space is representative of the operational input space; each software failure is observed; every debugging attempt is successful and faults are corrected without introducing new ones; and each fault contributes equally to the failure rate and all faults have the same probability of occurring.

Another problem is that the data collection requirements may be impractical. Inputs for testing must be selected randomly from an assumed input profile (input space). The assumed input profile may be incorrect or the usage of the software and the environment may change over time. Most important, accidents usually occur when the assumptions about the operational profile are incorrect, such as the loss of an F-18 when it got into an attitude the designers had assumed was not possible, and the software put the plane into a dive. Testing only the assumed inputs (states of the aircraft) will not detect such errors. In addition, testing the software according to the expected input profile requires mostly testing the software with test cases that have already been shown not to detect faults. Stress testing, boundary case testing, and trying to break the software—which are the most effective ways to detect faults during testing—cannot be used, so trying to get reliability numbers results in much of the testing effort being wasted in terms of finding errors.

A general problem with all these reliability modeling methods is that assumptions are required that do not fit real software. Academics, whose goal is to publish papers, are less concerned with the unreality of the assumptions than are practitioners who have to deal with real systems, operate under time constraints, and usually must rely on insufficient or incorrect information [41]. In the early days of reliability modeling, there was great enthusiasm in industry about the potential for these techniques. Reliability modeling, however, has not in practice provided accurate estimates or predictions and sophisticated reliability modeling techniques are seldom used today. More than 200 software reliability models have been proposed since the early 1970s, but none of the models can capture a satisfying amount of the complexity of software. Unrealistic constraints and assumptions are usually required. Finally, many of the reliability models assume that software is unvarying after it is first put into use, but in practice, software is frequently updated and changed. Even the smallest change could invalidate all previous data collected.

The other problem is that reliability is not equivalent to safety. One fault that leads to endangering the aircraft is more important than hundreds of faults that will have no impact on airworthiness. Software can perform correctly for millions of hours and be highly reliable, but if it has one fault that leads to an accident once a year, it would not be considered safe.

The large number of problems with reliability modeling may explain the low evaluation placed on them by the poll participants.

2.3.7 Hazard Analysis and Safety Engineering

The goal of airworthiness certification is to prevent accidents. At best, the software assurance techniques discussed so far show only that the software code correctly implements the requirements. These techniques do not tackle the problem of validating or generating the software safety requirements. Because the vast majority of accidents related to software have stemmed from inadequate software requirements, something else is clearly required. Hazard analysis, particularly STPA, has the potential to accomplish this goal. STPA can be applied to an existing system and its components (including software) to ensure that the requirements and design are safe—or it can be used in system engineering to generate the component safety requirements.

STPA provides information on how safety constraints could be violated. It is based on systems theory for which system components are viewed as a collection of interacting loops of control. For example, pilots exert control over the navigation system by inputting route data. The pilots receive feedback in the form of Global Positioning System (GPS) displays and aural alerts. Navigation software exerts control over autopilot (AP) software and hardware by sending course correction commands to the AP. Sensors and AP data give feedback to the navigation software in the form of aircraft heading, speed, and position. The safety of the system emerges because of the interactions of the pilot, software, and hardware. Safety analysts using STPA map out the system control structure, examining the interactions between the software, hardware, and human operators. The system control structure diagram and system specification information is used to identify safety-critical components and interfaces and to identify control actions (software outputs) that are potentially inadequate to maintain the safety of the system. Hazards are identified and translated into high-level requirements and constraints on the behavior of the system.

The benefits of an STPA analysis are that it provides a methodical means of performing a hazard analysis. The analysis is comprehensive in that it encompasses the software's behavior and interactions with other software, hardware, and operators during the entire system's runtime. The system behavior is represented in a simple control structure diagram, which allows for ease of review by all project stakeholders. After a thorough analysis, failure events and a wide range of inadequate controls were detected. STPA is focused because effort is concentrated where the control structure impacts safety the most.

STPA is relatively new, but its use so far shows great promise. When compared with standard hazard and failure analyses, such as FTA and failure modes and effects analysis (FMEA), STPA has done much better on real systems in terms of identifying design errors and requirements errors that could not be identified by the standard techniques. Although most of the research and industrial projects using STPA so far are in industries other than commercial aircraft (space, defense, medical, air traffic control, and others), a few experimental studies in aviation have begun.

The poll participants ranked hazard analysis and requirements analysis as important contributors to safe software.

2.3.8 Model-Based Development and Automatic Code Generation

High-level languages and modeling tools (such as Simulink) are popular for specifying the required functionality of the software. Code can be automatically generated from these high-level languages and models. These code generators are basically the same as any compiler. The only difference is that the distance between the language being used to specify what is wanted and the object code that performs the function is greater than for lower-level programming languages, such as Java or Ada.

Because automatic code generators are basically compilers operating on a high-level domain-specific notation, they must be treated like any other compiler. Either the code generator must be trusted as a development tool (as outlined in DO-330), or the correctness of the translated code must be verified through testing or formal verification (or both).

These types of language tools are very likely to be used increasingly for aircraft software. When the languages are close to those used by aerospace engineers (such as Simulink), they have the potential to reduce the problems associated with lapses in communication with software engineers about what the system and hardware engineers require. There are several drawbacks, however, even if the compilers or translators can be validated. The farther the specification language is from the object language, the more difficult it is to derive efficient code for hard real-time systems. Engineers often find the need to go into the generated code and make changes. Clearly that invalidates any assurance provided by validating the compiler. Another problem is that the automatically generated code often is not what would be generated by a human and is quite difficult to read and change without introducing errors if changes or tweaks are needed.

2.3.9 Assurance Cases

An assurance case encompasses documented evidence that provides a valid structured argument that a set of claim(s) about a system's properties are sufficiently justified. It consists of claims, arguments, and evidence. The producers of the safety case determine what claims, arguments, and evidence are provided.

Assurance cases have several issues and challenges that limit their usefulness and could even result in their being part of a hazard cause. Assurance cases are usually used in a goal-based certification environment. The product and process requirements in the current prescriptive product and process-based certification used in commercial aviation are specified, and an assurance case is not needed. Assurance simply involves checking that the product has the required features (such as fault-tolerant design or oxygen masks) and that the required development process was followed. Switching to a goal-based regulatory scheme would be problematic. The staffing requirements would be enormous because each case would be difficult and each review would be time-consuming. The qualifications required to determine whether each case was satisfactory would probably be impractical to provide. In addition, because information in the case would necessarily be proprietary, important stakeholders would be omitted from the certification process, which would likely be unacceptable.

A final limitation of assurance or safety cases is the problem of confirmation bias. There is a possibility that people tend to look for evidence to support their hypotheses. Evidence that does not support the goal is often ignored. Some psychological studies have found that reviewers of such arguments often miss the flaws in the argument. To avoid such limitations, a structured top-down approach needs to be adopted while gathering all evidence, rather than being selective.

The basic difference between assurance cases and the existing regulatory framework is that the group seeking certification decides what evidence to provide rather than the regulatory authorities and other stakeholders. Beyond the limitations specified in this section, those seeking certification can have conflicts of interest that may not lead to the highest standards being upheld.

Assurance cases were ranked relatively low by the poll participants.

2.3.9.1 Entrance and Exit Criteria

Entry and exit criteria are the set of conditions that should be met to commence and close a particular project phase or stage. For each of the software development life-cycle phases, entrance and exit criteria can be defined, documented, and signed-off on by all stakeholders.

For example, entrance and exit criteria could be defined for the Software Testing Process. Entry criteria represent the minimum eligibility to start the testing process on the build. Some generic examples of entrance criteria would be: all documents are up to date, the testing environment is successfully set up, all previous bugs have been resolved, and the formal walkthrough is completed. Exit criteria are the conditions at which the testing process can be stopped. Exit criteria examples would be: all tests have been successfully completed; regression testing has been completed; all bugs have been fixed; and final documents have been updated, reviewed, and signed-off on.

Entrance and exit criteria are helpful for keeping a program organized and on track as long as the criteria defined are adequate and appropriate. Problems arise when criteria are ill-defined and the program must re-evaluate and change the criteria because the program could not meet them. For example, if a program cannot meet the "all bugs have been fixed" criteria but must move on to the next phase because of schedule or budgetary constraints, tradeoffs must be negotiated so all the stakeholders have buy-in before the criteria can be modified. This buy-in process can be time-consuming and may degrade the level of software assurance if the criteria are lessened or diminished. It is essential that clear, well-defined criteria are agreed on for this method to be useful.

In addition, entrance and exit criteria are mostly used in the testing phases. Entrance and exit criteria could enhance the software assurance process if they were used in all phases of the project. An additional enhancement would be to ensure software safety entrance and exit criteria were incorporated into all software development life-cycle phases.

Entrance and exit criteria were not highly ranked by the poll participants as contributing to safe software.

2.3.10 Incremental Integration of Components

In general, incremental integration of components is a technique in which small parts of the system are developed and tested to eliminate errors at the subcomponent level. Once the component is considered error-free, it is integrated into the system in an effort to reduce errors and pinpoint any faults that arose during the integration step.

As incremental integration pertains to software, it is achieved by writing and testing code in small pieces and then combining the code into a working whole by adding one piece at a time. The perceived benefits of incremental integration are that faults are easy to locate because the new or changed component or its interaction with previously integrated components is the obvious place to look for a fault. In addition, the subcomponents are tested more fully because they are developed and tested before being integrated into a higher-level component. Unfortunately, components and tests sometimes are difficult to partition, and multiple

components may need to be incrementally added at the same time, which makes it more difficult to find the source of the errors or faults.

There are two types of incremental integration of components: top-down and bottom-up. The top-down method is performed by starting with the high-level components. These components are integrated and tested with stubs, or temporary code, for subcomponents. Then, the next lower level is integrated and tested incrementally. This is repeated until the bottom level is reached. The top-down method is beneficial in that designers develop the high-level system and can pinpoint architecture design problems early in the process. Interfaces are rigorously defined early so that interface errors found late—which can be extremely costly to fix—are avoided.

The bottom-up method is executed by combining lower-level modules to form builds or clusters. A special simple and short program is written to test the cluster, called a driver, at that interface level. Components at the next highest level are integrated and tested and new testing drivers are written for that level. This process is repeated until the top level is reached. The advantages to this method are no stubs are required, coding errors in critical modules may be found early, and lower-level units can be reused. Some types of interface faults can be more easily found but others are avoided or identified more easily by a top-down approach. A major drawback of the bottom-up method is the absence of a working system until integration is complete, which may mean that critical system design errors are found late, requiring very costly rework. Another disadvantage is that the developer has an increased workload in creating test drivers for each module.

2.3.11 Reverse Engineering

Reverse engineering is the process of determining the design of a device, object, or system through analysis of its structure, function, and operation. It often involves taking something (e.g., a mechanical device; an electronic component; a software program; or a biological, chemical, or organic matter) apart and analyzing its workings in detail to be used in maintenance when the original design documentation is not available—or to try to make a new device or program that does the same thing without using or simply duplicating (without understanding) the original.

Reverse engineering in the context of a system made up of hardware and software such as an aircraft, power plant, or breaking system for a car means performing the analysis necessary to create representations of the system at a higher level of abstraction.

Reverse engineering is used for many purposes: as a learning tool; as a way to make new, compatible products that are cheaper than what is currently on the market; for making software interoperate more effectively or to bridge data between different operating systems or databases; and to uncover the undocumented features of commercial products.

Using reverse engineering for safety and security concerns is difficult. The following issues related to certification are listed in Position Paper CAST-18: Reverse Engineering in Certification Projects [42]:

- Lack of a well-planned process
- Poor justification for reverse engineering
- Lack of access to experts and original developers
- Complex and poorly documented source code
- Abstraction and traceability difficulties
- Interface and integration problems
- Certification liaison process problems

To be successful, reverse engineering for safety and security issues must:

- Be used on high-integrity software (this type of software often does not need to be reverse engineered).
- Identify how the reverse-engineering process will satisfy the safety and security objectives of the project.
- Be well coordinated with all organizations involved in the safety and security process.
- Have a well-defined and planned process, and must be well-documented, including the goals of the process.

The reverse-engineering process [43] is labor-intensive, time-consuming, and usually yields substandard results when compared to other safety and security analysis processes, methods, and tools.

2.3.12 Phase 1 Summary

This review and analysis of software assurance techniques has been rather pessimistic in that no simple solution or reasonable technique was found to solve the problems. However, if any of the usual approaches to software assurance were highly effective, the current number of operational software errors would not be present. Most are useful to achieve some specific goals and certainly not using them would result in worse problems than are currently encountered. The problem is in deciding which ones should be required and used. Some general guidelines:

- The goal should be to select a set of techniques that are likely to detect different types of errors rather than essentially duplicative techniques.
- Assurance techniques that are more difficult and error-prone when compared with simply creating the software from the outset will have limited usefulness in providing assurance.
- Although some simple tasks can and should be automated, there is no replacement for human review in the assurance process, including both the review of the software artifacts themselves and of the assurance process.
- Software assurance should not just involve only the code but should cover all of the important artifacts of software development and maintenance, including specifications and user manuals.

- Deficiencies in the requirements specifications are the most significant contributor to operational software failures and contributions to accidents, so assurance of the requirements necessitates at least as much effort as code assurance techniques.
- Safety is not equivalent to reliability. Hazard analysis and safety techniques need to be used to assure software airworthiness, not just reliability enhancement and measurement techniques.

3. PHASE 2 RESEARCH: TECHNICAL ADVANCES TO STREAMLINE THE SOFTWARE DEVELOPMENT PROCESS

3.1 STREAMLINING THE PROCESS: INCORPORATING SAFETY AND REDUCING COST

Incorporating safety early in the software assurance process will aid in streamlining the process by locating errors early in the development phase and, consequently, helping to reduce the software development and assurance costs.

To achieve reductions in the cost of software assurance, it is crucial to (1) identify the potential safety problems in the domain, (2) devise an engineering solution that reduces the risks to an acceptable level, (3) define a systems architecture that isolates the critical functions to well-defined components, and (4) develop the components in the system to the appropriate level of assurance. If done well, the number of highly critical components will be small and the complexity of those critical components comparatively low. By reducing the scope of the critical software assurance, the assurance costs can be kept under control. Therefore, the key to reduced assurance costs starts with requirements assurance. Requirements assurance includes appropriate hazard analysis and system safety techniques to identify the components of the system that will require the highest level of assurance—there is a need for effective system safety analysis applicable to today's complex systems. Once identified, the requirements on these critical components must be defined and assurance must be provided that these are the right requirements to ensure system safety. There also needs to be assurance that the components (in the context of this study, the software components) in fact satisfy the requirements—there is a need for software architecture assurance. Finally, there needs to be assurance that the code is an accurate representation of the architecture and requirements—there is a need for software implementation (coding) assurance. A variety of techniques and tools were investigated in this study that could aid in assuring safety and streamlining the process by eliminating errors early in the development phase, including hazard analysis, model-based requirements tools, model-based architecture tools, human reviews, and coverage criteria.

3.1.1 Requirements Assurance

Software requirements assurance is the task of ensuring that software requirements are in place to meet all system requirements allocated to the software components.

> "The software requirements and design process are simply subsets of the larger system engineering process. System engineering views each system as an integrated whole, even though it is composed of diverse, specialized components, which may be physical, logical (software), or human. The objective is to design subsystems that, when integrated into the whole, provide the most effective system possible to achieve the overall objectives" [44, p. 17].

Methods that can improve the likelihood of the software requirements aligning with the system requirements include:

- Software hazard analysis–Techniques that can generate safety requirements from identified hazardous behavior or ensure that hazardous behaviors are handled in the requirements.
- Model-based specification and analysis–Building executable and analyzable models of the required black-box behavior of the system and validating that this behavior is not hazardous and is complete and consistent.
- Reviews involving human experts–Informal or structured methods to identify missing or incorrect requirements.

3.1.1.1 Software Hazard Analysis

Software hazard analysis should examine system and software requirements and design to determine potentially unsafe system actions, such as out-of-sequence, wrong event, inappropriate magnitude of data, and inadvertent command actions. These types of undesirable system actions could occur between components of a system; between a system and its environment; and between legacy components (e.g., software) and new components (software).

Software hazard analysis requires methods and processes that may not have been used on previous systems in which software was not such a key component. In the past, hazard analysis has often focused on faults and failures. Regarding one research study investigating aviation and space systems accidents:

> "Standards for commercial aircraft certification, even relatively new ones, focus on component reliability and redundancy and thus are not effective against system accidents. In the aircraft accidents studied, the software satisfied its specifications and did not 'fail,' yet the automation obviously contributed to the flight crews' actions and inactions. Spacecraft engineering in most cases also focuses primary effort on preventing accidents by eliminating component failures or preparing for failures by using redundancy. These approaches are fine for electromechanical systems and components, but will not be effective for software-related accidents" [45, p. 2].

New hazard methods are needed and, ideally, these hazard-analysis processes would be used at the system level and the software level, streamlining the process. Note that the notion of "safety" here is not limited to human safety, but also includes loss of mission, loss of equipment, and negative environmental impacts—all highly undesirable outcomes to which we do not want the software to contribute. The goal of the software assurance process is to ensure that the software does not contribute to the harm of humans, accomplishes its mission, protects equipment, and does not contribute to negative impacts outside of the system it immediately controls.

Software is quite different from hardware. For example, software does not fail in the traditional hardware sense (if software misbehaves, it is a design problem), software does not wear out, and software does not behave randomly. Therefore, techniques developed for hardware cannot be expected to be directly applicable to software. Because of the fast pace of technological change; the changing nature of accidents; the new types of hazards in software intensive systems; society's decreasing tolerance for single accidents; increasing interactive complexity and coupling in new systems; more complex relationships between humans and automation; and changing regulatory and public views of safety, new techniques for safety analysis suitable for today's complex software intensive systems are needed [46]. STPA is a new hazard-analysis technique that can be applied to systems, including software. STPA can also be used to analyze systems with COTS, legacy software, humans, and organizational structure, such as including FAA regulations and procedures in the analysis of the system.

3.1.1.1.1 STPA Hazard Analysis

STPA is based on System Theoretic Accident Model and Processes (STAMP). STAMP is an accident model that views systems in terms of overlapping control loops. A traditional model of accidents called a chain of events model, in which event A leads to event B, which leads to an accident, is inadequate in handling complex systems for which software, humans, and hardware are tightly coupled. The chain of event analysis of the accident is viewed linearly and it is difficult to include non-linear relationships, such as feedback or the safety culture of the organization or industry and factors that are directly related to the occurrence of accidents, into the model [46]. Another limitation of a chain-of-event model is determining what events to include in the chain. The reasons and motivation as to why the accident is being analyzed may determine the stopping point of the root cause. If the assignment of blame for the accident under investigation is the goal of the analysis, this type of analysis may never reveal why the event(s) occurred:

> "In an analysis by the author of recent aerospace accidents involving software in some way, most of the reports stopped after assigning blame (usually to the operators) and never got to the root of why the accident occurred—for example, why the operators made the errors they did and how to prevent such errors in the future or why the software requirements error was made and why it was not detected and fixed before the software was used" [47, p. 4].

Another development to consider is the increase in complex, interactive systems for which software plays an ever-increasing role in controlling the system. In the past, simple systems could be developed by testing and analyzing the individual components to ensure they would perform as intended. As the systems have increased in complexity and involve human, software,

and hardware components, the chain of events accident model is lacking. Some analysis techniques—for example, FTA or FMEA—were created long ago to analyze primarily electro-mechanical systems. These analysis techniques take a linear approach to analysis and can miss important interactions associated with individual components operating without failure but not interacting as intended with other components [45]. In analyzing a complex system with human, software, and hardware interactions, a systems and control-based approach should be considered. The goal of this approach is to control all the interactions between components so that the system is adequately controlled (i.e., no accidents can occur).

STAMP was used to aid in analyzing accidents:

> "The most basic concept in the new model is not an event, but a constraint. In systems theory, control is always associated with imposition of constraints. The cause of an accident, instead of being understood in terms of a series of events, is viewed as the result of a lack of constraints imposed on the system design and on operations, that is, by inadequate enforcement of constraints on behavior at each level of a socio-technical system" [46, p. 13].

STAMP treats accidents as a dynamic control problem, not a failure problem.

3.1.1.1.1.1 The STPA Process Steps

STPA is a manual hazard analysis technique based on STAMP and can best be described by detailing the process steps:

1. Identify system-level hazards and review/develop safety constraints.
2. Describe the system control structure.
3. Identify inadequate control actions that could lead to a hazardous state.
4. Determine how the software could produce inadequate control actions.

As an example, consider the FAA's Next Generation Air Transportation System (NextGen). NextGen is a new satellite-based national airspace system developed to replace the ground-based system in the United States. Using NextGen as an illustration of how to apply STPA, the first step is identifying the preliminary hazards: NextGen gives an advisory that violates the aircraft minimum separation distance. The corresponding safety constraint would be: NextGen must not give an advisory that violates the aircraft minimum separation distance.

The second step in the STPA process is to describe the control structure:

> "In systems theory, open systems are viewed as interrelated components that are kept in a state of dynamic equilibrium by feedback loops of information and control. In general, to affect control over a system, four conditions [are required]:
>
> 1. The controller must have a goal or goals.
> 2. The controller must be able to affect the state of the system.
> 3. The controller must be (or contain) a model of the system.
> 4. The controller must be able to ascertain the state of the system." [44]

Keeping these conditions in mind when describing the control structure will aid in determining a controller of the system and what is considered an interaction.
The following diagram (figure 1) depicts a high-level description of the NextGen control structure based on descriptions of the system available on the NextGen website. Note that Safeware Engineering was not privy to any NextGen specifications in developing this diagram; therefore, the diagram may not be complete or as detailed as it should be, but it will serve as a sufficient example for explaining the STPA process.

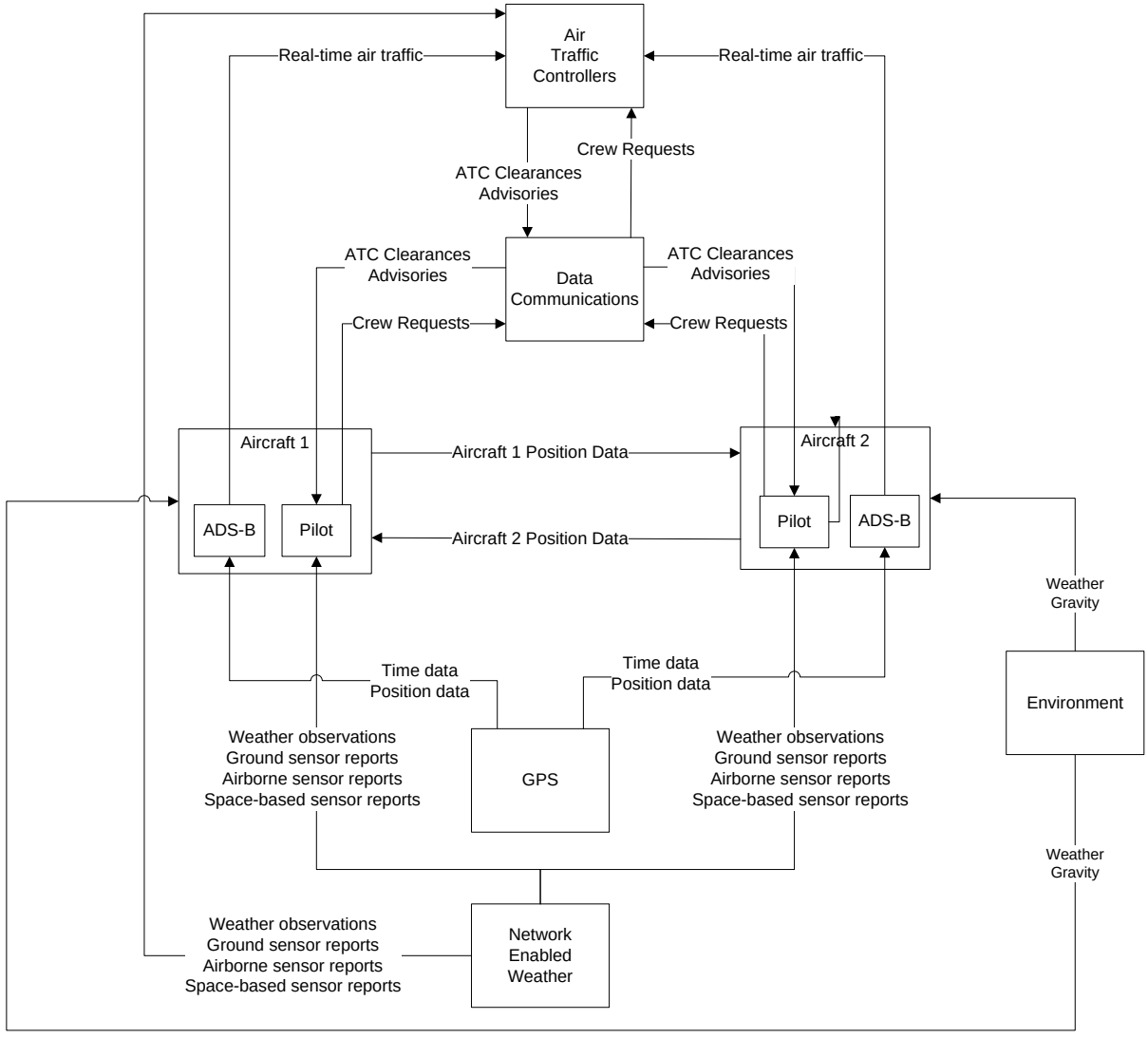


Figure 1. Example NextGen control structure diagram

Each box represents a controller of the system as defined by one of the four required conditions. A controller of the system could be a human, such as the pilots and air traffic controllers, or software (such as the automatic dependent surveillance-broadcast or ADS-B), COTS (such as the GPS), or hardware (such as sensors on the aircraft). The FAA regulatory body could be considered a controller of the system because it imposes control over air traffic controllers and pilots via regulations. The environment is also a controller of the system because it affects the

airplanes via weather that can influence (control) when an airplane can take off or which route it may have to fly. Once the controllers of the system are identified, the interactions between the controllers are added to the diagram. The interactions could include commands, data exchanges, and feedback. The goal of an STPA analysis is to ensure the system is controlled in such a way as to not cause a hazard. To accomplish this goal, the interactions between the controllers are examined. Potential control actions that could lead to minimum aircraft separation violations are identified.

This is accomplished by determining if:

1. A required control action between controllers of the system is not provided.
2. An incorrect or unsafe control action is provided.
3. A potentially correct or inadequate control action is provided too late or too early (i.e., at the wrong time).
4. A correct control action is stopped too soon.

For example, if the control action between the ADS-B and the air traffic controllers is examined, and inadequate control actions that could lead to a hazard are identified—using the four criteria listed above—the following inadequate control actions can be identified, as shown within the blue box in figure 2.

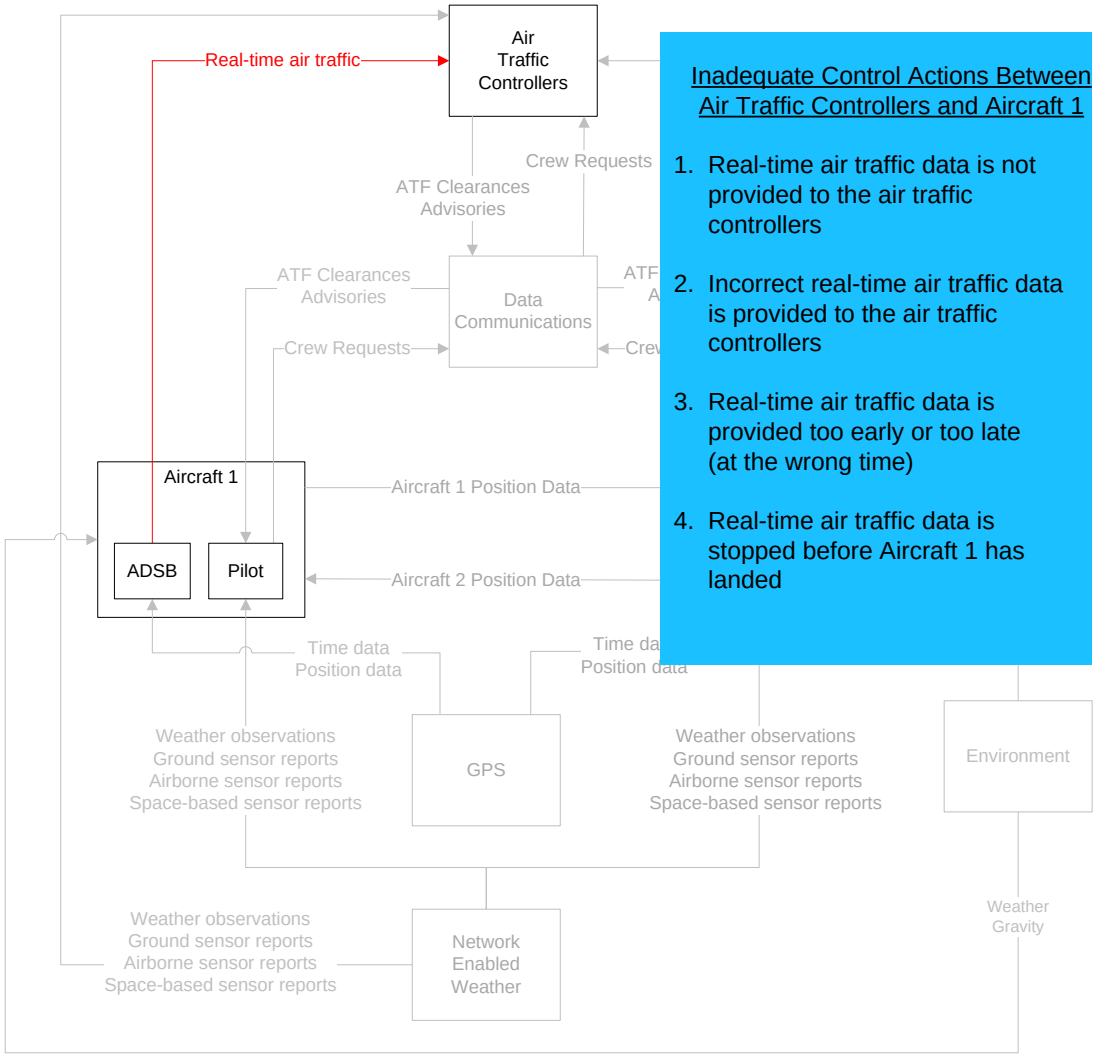


Figure 2. Example NextGen inadequate control actions

The next step would be to determine how those inadequate control actions identified in the blue box in figure 2 could occur. Examples of situations in which real-time air traffic data are not provided could be when a communication flaw from ADS-B arises or when a malicious denial of service attack on the connection develops. Depending on the type of communication there is between those two controllers, various items could be investigated: malicious attacks to the digital communications (security threats), hardwire errors, or radio frequency flaws. Again, this example could be more detailed if requirements of the system were known, but this example also demonstrates that project stakeholders can get a useful view of how the system should behave very early in the project life cycle—before requirements have been developed.

A second example of inadequate control action is incorrect real-time air traffic data being sent to the air traffic controller. GPS data are delivered to the ADS-B correctly, but the data are corrupted somewhere inside the ADS-B controller. Analysts would review scenarios in which the data could get corrupted.

A third type of inadequate control action would be if the air traffic data are provided too late or too early (at the wrong time) to the air traffic controllers because of timing issues. Analysts would review command prioritization issues, processor throughput capabilities, and data obsolescence to determine if data could be sent at the wrong time.

Finally, an example of a control action stopped too soon would be if the real-time air traffic data are stopped before airplane 1 has landed. This could be because the ADS-B trusts that airplane 1 has landed, when in fact it is still airborne (i.e., an example of a software logic error).

During analysis, the analyst would review the requirements to ensure that the inadequate control actions have been mitigated.

3.1.1.1.1.2 Assurance of COTS and Legacy Software Using STPA

One of the goals of this research was to find approaches that are best used for COTS software and legacy software. COTS and legacy software can be included in the STPA analysis. For example, in the NextGen STPA control structure, the GPS is a COTS component. As long as the interactions between the GPS and other NextGen controllers are known, the GPS is treated as one of the controllers in the system. The STPA analysis would be performed and the GPS controller would need to be controlled in such a way that it did not cause a hazard. The requirements model would capture and trace COTS and legacy-related requirements. Any changes to COTS or legacy-related requirements could be updated in the STPA analysis to determine if the system could become uncontrolled and cause a hazard.

3.1.1.1.1.3 STPA Advantages

Aviation systems are using more software to perform complex actions and interactions between systems. Traditional hazard analysis techniques, such as FTA, do a good job at finding failures, but in certain instances, software can cause a hazard without failing. STPA is an additional hazard analysis technique that can pinpoint specific design problems early in the requirements phase. Studies have shown that STPA can catch more hazard causes than other hazard analysis techniques. In 2002, a comparison between FTA and STPA for the Traffic Alert and Collision Avoidance System II (TCAS II) system showed that:

> "FTA was less comprehensive. That is, STPA included more inadequate control actions and provided much more complete information about why that action might occur. The FTA stopped investigating the cause of a potential inadequate behavior before STPA did" [47].

Another study was performed in 2010 on the Japan Aerospace Exploration Agency (JAXA) H-II Transfer Vehicle, for which an STPA analysis was compared to the FTA:

> "The comparison showed that STPA identified causal factors identified in the fault tree analysis, but STPA also identified additional causal factors that had not been identified by fault tree analysis. The additional factors included those that cannot be identified using fault tree analysis, including software and system design as well as system integration of the International Space Station, the H-II Transfer Vehicle, and the NASA/JAXA Ground Stations" [48].

In 2012, an analysis based on RTCA DO-312 (Safety, Performance and Interoperability Requirements Document for the In-Trail Procedure in the Oceanic Airspace (ATSA-ITP) Application) was compared to STPA on the NextGen system. RTCA DO-312 is intended to provide:

> "The minimum operational, safety, and performance requirements and interoperability requirements for the implementation of enhanced Airborne Traffic Situational Awareness for 'In-Trail Procedure'" [49].

The STPA analysis found 19 high-level safety requirements that were not found in the official NextGen documents [50].

General Motors performed an internal study for which two teams were tasked with deriving safety requirements. One team performed STPA. The other team followed the GM safety process. One of the conclusions of the study was:

> "[The] STPA technique is valuable and different from other techniques such as traditional FTA and FMEA" [51].

Another study reported on in October 2014 compared SAE ARP 4761, "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment," to STPA. The authors concluded:

> "In the reality of increasing aircraft complexity and software control, we believe the traditional safety assessment process used in ARP 4761 omits important causes of aircraft accidents. We need to create and employ more powerful and inclusive approaches to evaluating safety that include more types of causal factors and integrate software and human factors directly into the evaluation. STPA is one possibility, but the potential for additional approaches should be explored as well as improvements or extensions to STPA. There is no going back to the simpler, less automated designs of the past, and engineering will need to adopt new approaches to handle the changes that are occurring" [52].

3.1.1.1.1.4 STPA Recommendation

Software hazard analysis plays a key role in the requirements assurance process. STPA is a software hazard analysis technique that can identify more hazard causes than some other analysis

methods, such as FTA. STPA analysis is comprehensive because it encompasses software's behavior and interactions with other software, hardware, and operators. The analysis is methodical by organizing what is often done in an unstructured way. Because this method has been proven in the field and applied to real time, safety-critical complex systems, it was considered for further study in phase 3.

3.1.1.2 Model-Based Specification and Analysis Tools

Ensuring that all necessary requirements have been captured, that the requirements are consistent, and that they indeed collectively enforce the system safety constraints is a daunting task. With today's complex systems (often referred to as systems-of-systems), the hierarchical nature necessitates demonstrating that the components in the architecture at one level in the system's hierarchy work together to enforce all safety constraints imposed at that level. To enhance the capability to understand and develop increasingly complex systems, model-based specification and analysis has emerged as a powerful technique.

Traditionally, software development in critical systems has been largely a manual endeavor. Validation that the right system is being built has been achieved through requirements and design inspections and reviews. Verification that the system is developed to satisfy its specification is archived through inspections of design artifacts and extensive testing of the implementations. As mentioned earlier, in critical software systems, the validation and verification phase is particularly costly and may consume a large percentage of the software development resources. The current trend toward model-based development is one attempt to address this problem. In model-based development, the development effort is centered on a formal or semiformal model of the proposed system (in this case, software system). This model is typically expressed in some graphical or pictorial language because it is well-liked by engineers and software developers. Through manual inspections, tool-supported verification, and simulation and testing, the models can be shown to possess desired properties. The implementation can often be automatically (and, hopefully, correctly) generated from this model, and the assurance effort of the implementation can be significantly reduced without loss of quality. There are currently several commercial and research tools that aim to provide part or all of these capabilities. These include Simulink and Stateflow from Mathworks; Safety Critical Application Development Environment (SCADE) from Esterel Technologies; Statemate and Rhapsody from Rational; SpecTRM from Safeware Engineering; the Architecture Analysis and Design Language (AADL); and various unified modeling language (UML) tools from a collection of vendors.

The capabilities of model-based development enable development teams to follow a different development process than traditionally used in the critical systems industry. The development will now be centered on the model and the assurance activities have been largely moved from testing and analyzing the code to analyzing and testing the model. Through an iterative process in which requirements and safety constraints are compared against models that capture the details of the required behavior, both the requirements and constraints, the models are perfected. Productivity improvements, in particular reduction in assurance costs, will be achieved through reduced emphasis on unit testing of code; increased reliance on automated analysis tools applied in the requirements and model domain; and significantly reduced cost of rework of the system during the late-stage testing activities. In addition, accommodating and evaluating the inevitable requirements changes that tend to materialize late in projects become less costly with this

approach (it is easier to evaluate the impact on the system and, when the impact is known, to correctly update the production software and retest). Anecdotal evidence has shown that by focusing efforts on effective requirements, model verification, and validation activities early in the life cycle, few problems are discovered during testing—leading to a reduction (if not outright elimination) of the very expensive rework process typically plaguing system development.
The model-based development notations and tools generally use a variety of diagrams to capture the structure and behavior of the system being modeled. The structural view emphasizes the static structure of the system; systems components; relationships or connections between the components; data flowing through these connections; and attributes of the components. The behavioral view emphasizes how the state of the components and component interactions evolve over time.

To be effective as aids in the requirements assurance process, it is desirable that the tools and notations they support have the following attributes:

- The modeling must be applicable from the earliest requirements.
- The models must be refinable as the requirements are refined.
- They provide tracing between high-level requirements and lower-level (design/architecture) requirements and models.
- They provide a way to mark safety critical requirements.
- They provide a graphical/pictorial representation of the requirements.
- The representation must be easy for all stakeholders to read and review.
- The must be analyzable for completeness and consistency of requirements and models.

3.1.1.2.1 Model-Based Specification and Analysis Tools Recommendation

Model-based specification and analysis tools can help practitioners streamline the process of requirements assurance by capturing high-level requirements and safety constraints; tracing them to lower level or subsystem requirements; developing a model of those requirements; and allowing for change so that the requirements match the intended behavior of the system.

3.1.1.3 Human Reviews

Like all of the requirements assurance methods proposed by Safeware Engineering, structured walkthroughs save time and money by finding and correcting errors early in the life cycle. Part of the value of this method comes from having reviewers (rather than just designers) with different technical backgrounds, experience, and expertise review the requirements and provide input. In addition to improving the requirements, structured walkthroughs validate and improve the related life-cycle work products and can provide professional growth to participants by giving them an opportunity to look at different development or maintenance approaches [53].

Most studies indicate that inspections are beneficial, although this might be due to unsuccessful studies tending not to be published, especially if by an unknown researcher [54]. In the survey of published results, Laitenberger and DeBaud reported that inspections revealed between 19% and 93% of defects. However, this was only a percentage of all defects found. Defects that were not

found during testing or some other part of the process were not counted. Nor did they report whether the defects would have been found during testing.

Issues that can decrease or eliminate the effectiveness, and increase the cost, of a walkthrough include:

- Inadequately prepared reviews
- Defensive author(s)
- Overly long walkthroughs
- Allowing discussions of solutions rather than just defects [55, pp. 100–101].

The technique used to read the software products may also increase or decrease the effectiveness of the product. Reading techniques include ad-hoc reading, checklist-based reading, reading by stepwise abstraction (for code only), and scenario-based reading. Ad-hoc reading provides little to no guidance to the reviewers and depends heavily on their abilities. Checklists provide a set of questions that should guide the reviewers, but the questions are often too generic, do not usually come with directions, and may focus only on types of defects found in the past. Stepwise abstraction requires the reviewer first to attempt to determine code behavior by starting with a few lines and building up, then compare the actual function with the specification. Scenario-based reading provides each inspector with a different set of questions or detailed instructions so that each inspector has a different focus [54].

Scenario-based reading, proposed by Basili in the mid-1990s, has suggested three different approaches: defect-based reading, function-based reading, and perspective-based reading [54]. Defect-based reading focuses each inspector on a different class of defects. A controlled experiment supported the hypothesis that defect-based reading was a better approach for finding defects than ad-hoc or checklist-based reading. By contrast, the initial experiments on function-based reading appeared to favor ad-hoc reading, but the results may have been skewed by the experience level of the participants. In function-based reading, the participants look at items from a function point analysis, "which defines a software system in terms of its inputs, files, enquiries, and outputs" [54].

With perspective-based reading, reviewers look at the document from a particular requirements user's point of view. These could be the customer, the system designer, or the system tester, although they might include other perspectives, such as the maintainer [56]. These perspectives should not be confused with the roles held during a structured walkthrough, which include presenter, moderator, reviewers, and scribe. Some online sites appear to mix these two with "roles" such as "presenter," "chairperson," "scribe," "maintenance oracle," and "standards bearer" but do not provide the detailed instructions for either of the last two roles/perspectives [57]. Perspective-based reviewers first create a relevant representation of the requirements. Most of these representations can be used to save effort later. For example, the tester perspective would generate test cases for each requirement by asking about the completeness and appropriateness of the requirement, determining the sets of conditions that needed to be tested, and recording specific tests that could serve as the basis for the actual testing effort. This technique was tested using 150 students and 25 professional developers [56] and improved the defect detection rate by as much as 30% over other less-structured methods [58]. However, it

appeared to provide a greater benefit for less-experienced users and also provided them with training benefits [56]. A later study conducted in Brazil compared perspective-based reading to a defect-based checklist. The results were marginally in favor of perspective-based reading, with one document being equal and the other showing a slight statistical improvement [59].

3.1.1.3.1 Human Reviews Recommendation

Based on the research, it would appear that the most benefits overall stem from conducting perspective-based readings during structured walkthroughs of the requirements. The benefits of perspective-based reading in addition to defect detection include training of junior personnel and artifacts that can be reused later in the development cycle—for example, during testing. These benefits help to alleviate cost concerns. A defect-based reading or checklist is almost as good at finding defects but does not provide long-term benefits.

3.1.2 Software Architecture Assurance

The purpose of software architecture assurance is to relate the requirements identified by the system and software requirements with specific software components and identify those software components, which control or affect the hazards, as safety-critical. It also examines the independence/dependence and interdependence of the software components. Safety-critical software components are either components that trace to a hazard, as identified in the system and software hazard analysis, or that directly or indirectly influence safety-critical software components.

3.1.2.1 Architecture Modeling and Analysis

Building on the modeling in the requirements phase of the software assurance process, architectural modeling, and analysis can give early indications of whether a design can be created that meets the requirements. Similar to all the steps in the process, the goal is to determine as early as possible if the requirements allocated to the architectural components will guarantee safe operation of the system and ensure that the component requirements are met by the component behavior.

Similar to the requirements model, the architectural model should have a graphic/pictorial representation improving understandability by all reviewers. Ideally, the tool chosen will have been used for the requirements as well, reducing errors during transfer, and streamlining the process. It should provide traceability up to the requirements and down to the code and have the ability for the user to designate safety-critical architecture components.

The architecture model should also be analyzable and executable. The analysis should aid in determining if the design is correctly and completely specified. The ability to execute the model can increase the analysis capability by allowing pre-coding testing of the design, including timing requirements.

Tools used for modeling software architecture should:

- Have a graphical/pictorial interface to improve readability.
- Provide an executable model.
- Provide analysis.
- Be used for modeling the requirements and architecture.
- Relate architecture components to high-level requirements and hazards.
- Designate safety-critical architecture components.
- Ensure that requirements are correctly and completely specified in the high-level architecture design.

3.1.2.1.1 AADL

A recent development worth mentioning is the interest that surrounds the AADL. AADL is a notation that includes descriptions of both hardware and software components and their interactions. This notation has gathered significant interest in the research community over the last several years. Of particular note is the support of an extension mechanism (called an annex) that can be used to extend the language to support additional features, such as the inclusion of existing modeling notations (e.g., the ones discussed earlier or domain-specific notations developed for a particular purpose). AADL has seen extensive use in the research community and it fills a central role in the ongoing Aerospace Vehicle Systems Institute's (AVSI) System Architecture Virtual Integration (SAVI) program. Nevertheless, although AADL has been used in research and practice, there are no known widely distributed commercial tools supporting AADL and its extension mechanisms.

3.1.2.1.2 Architecture Modeling Recommendation

The architecture assurance process can further streamline the software assurance process by analyzing and testing the models before any architecture is built. The models can be perfected through an iterative process in which requirements and safety constraints are compared against a model of the architecture.

3.1.3 Implementation (coding) Assurance

3.1.3.1 Test Adequacy Coverage Criteria

Software testing is generally performed for two basic reasons. First, software can be tested for validation purposes. The software is tested to determine if it does what it is supposed to do—test as a means of validating the software and the requirements against the system's and users' needs. Second, software can be tested for verification purposes—test to determine whether or not the software conforms to its requirements (or specifications). For the purpose of this report, Safeware Engineering will consider the notion of verification testing as testing to provide assurance that the software conforms to its requirements. This type of software assurance activity is within the purview of DO-178B (and its successor DO-178C) [15] and testing serves a prominent role as the preferred means of assuring that the software satisfies its requirements. To determine the rigor of the testing process, these standards partially rely on test adequacy coverage criteria.

3.1.3.2 Coverage Criteria

Coverage criteria are used as a measure of how well a program is exercised by a test suite. There are a number of coverage criteria, including statement coverage, function coverage, decision coverage, condition coverage, condition/decision coverage, MC/DC coverage, and observable MC/DC. Statement coverage is a very basic type of code coverage that can only report whether a statement has been executed, which potentially leaves many types of code constructs untested. Function coverage is another basic type of code coverage analysis that can report whether a function has been called or not; it does not say anything about what was executed inside it or how or why the function was called.

Decision coverage requires that every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once. However, some decisions are not tested in this analysis (note that a decision in the accepted terminology refers to any Boolean condition containing Boolean operators; a condition is an atomic Boolean expression containing no Boolean operators). Take for example the decision structure of choosing A or B and consider the two test cases: (i) A = true, B= false *and* (ii) A = false, B = false. These two test cases will toggle the decision between true and false. However, in decision coverage, the effect of B = true never needs to be tested because the coverage requirement has already been met. If there is some problem arising only when B is true, this may not be revealed by a test suite providing decision coverage.

Condition coverage includes the requirement that every point of entry and exit in the program has been invoked at least once and every condition in a decision in the program has taken all possible outcomes at least once. With condition coverage, the decision (A or B) can be met with the test cases: A = true, B = false; and A = false, B = true. In this case, however, the outcome of the decision does not take on all possible outcomes (it is always true) as it does in decision coverage [560].

Condition/decision coverage combines the requirements for condition and decision coverage. There must be sufficient test cases to toggle the decision between true and false, as in decision coverage, and all possible outcomes must be met. For instance, the decision (A or B) with the test cases A = true, B = true, and A = false, B = false would meet the condition/decision requirement; the test case A = true, B = false is not needed for this coverage criterion to be met. Note that in this case, if the decision was erroneously encoded as (A and B), the test cases would be unable to detect this problem even though condition/decision coverage has been achieved.

MC/DC is an advanced type of code coverage analysis. It builds on top of condition/decision coverage and, as such, it too requires that all code blocks and code branches have been tested. MC/DC adds the requirement that all conditions (atomic Boolean expressions) making up a decision (a complex Boolean expression) have been shown to drive the overall branch decision, independent of other conditions. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, requiring a minimum of n+1 test cases for a decision with n conditions. For example, the decision structure of choosing A or B requires three test cases to fulfill MC/DC: (i) A = true, B= false, (ii) A = false, B = true, and A = false, B = false. For

decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the other coverage methods discussed so far.

Multiple condition coverage is even more rigorous than MC/DC. It requires that every combination of inputs to a decision be executed at least once. Although this coverage is the most complete, it is also difficult to meet the MC/DC coverage because of the amount of testing required. For a decision with n inputs, multiple condition coverage requires 2^n tests—therefore, rendering this criterion infeasible in practice.

At this point, it is worthwhile to recall the intended usage of coverage criteria. Test cases can be derived from a multitude of sources (e.g., the requirements, design, user manuals, domain experts, and, of course, code itself). Test cases derived without consideration to the internal structure of the software are commonly called black-box tests (the software is a black box). If the internal structure of the software is used when deriving the test cases, it is referred to as white-box (or glass-box) testing because the software structure can now be observed and used when constructing tests. The intended usage of test-adequacy coverage criteria was a measure of the quality of black-box tests derived from the requirements. One would derive a test suite based on the behavior as defined in the software requirements, run these tests on the software, measure coverage of, for example, MC/DC, and then determine the course of action based on the coverage achieved. If low MC/DC is achieved, there are several possible causes. First, the analyst might have done a poor job deriving test cases and must go back to the requirements and derive additional test cases. Second, the analyst may have done a good job deriving tests, but the requirements are incomplete and there is software added in the design process not covered by the requirements. A review and revision of the requirements (and the development process) are in order. Finally, it is possible that the uncovered portions are simply not possible to cover and the analyst can determine this fact and document the reasons. Either way, the coverage criterion is used as a quality measure of tests derived elsewhere. The coverage is not used as a guide for determining what tests to run. This point is worth remembering because the emergence of powerful test case generation tools now enables users to quickly and automatically generate tests that provide a predefined coverage. This, however, can be a risky proposition leading to very poor testing results.

Researchers and practitioners studying coverage criteria have observed that the requirement of testing to meet the MC/DC criterion is a concern in the avionics industry. Meeting the MC/DC criterion is said to be much more costly and time-consuming than satisfying a weaker structural coverage criterion, such as function or decision coverage. However, according to a study conducted by Dupuy and Leveson, meeting MC/DC was not significantly more difficult than only satisfying condition/decision coverage and found errors that meeting lower level structural coverage (in this case functional and condition/decision) could not find [61]. Vilkomir and Bowen point out that MC/DC fails to check for false operation type failures, in which the system starts in an alternate scenario. They went so far as to suggest a perhaps more rigorous coverage, which they called reinforced condition/decision coverage (RC/DC) [62]. RC/DC does not appear to have further research support, but this may be because it requires more test cases than MC/DC or because it was researched in the context of one particular formal language. Despite what some may want, as these studies point out, requiring a less rigorous level of coverage does not appear to be the right answer.

Further complicating the question of what is really achieved through the use of the MC/DC criterion is the research conducted by a research group led by Professor M. Heimdahl. In one study, they compared the results of satisfying MC/DC for an implementation in which much of the calculations and logic had been inlined (i.e., no temporary variables were used to store Boolean values for use in later computations, all decisions were expanded to their full complexity) versus test sets for a non-inlined implementation. Test suites chosen to satisfy MC/DC for the inlined implementation "outperformed the non-inlined test suites in the range of 10%–5940%" [63]. In addition, the oracle (the mechanism that determines whether the test passed or failed) used the mechanism that considered the values of internal variables and outputs and performed better than the mechanism that only looked at system level outputs. It is also worth noting that none of the combinations found 100% of the injected faults.

Current research into stronger coverage criteria, such as observable MC/DC, appears promising [64]. Observable means that the change should be observable in the system level outputs. Initial research indicates that the success level of this coverage criterion is neither dependent on the implementation structure nor on whether test results are compared to just output or to intermediate variables as well. It also appears to generate better test cases. Sometimes it does require a larger test suite than MC/DC and, further, it is also sometimes harder to meet the obligations of MC/DC. Until these issues are worked out, it may not be useable by industry, but this course of study appears to be going in the right direction.

There is a considerable lack of clarity regarding the use of structural coverage criteria—such as MC/DC as a measure of goodness of a testing effort. MC/DC has served the community well over the last several decades; the failure rate of critical avionics software is extremely low and the industry as a whole has a remarkable safety record. However, there is considerable uncertainty regarding how good MC/DC really is as an exit criterion, whether the cost is commensurate with the benefits, and whether there is ample opportunity for unaware (or unscrupulous) contractors to accidentally (or deliberately) structure their codes to make it easier to satisfy MC/DC; unfortunately, the resultant test cases may be highly inefficient. Therefore, Safeware Engineering recommends staying with the testing guidelines defined in DO-178B (and DO-178C). Note, however, that there are recent developments discussed in the next section that must be considered going forward.

3.1.3.3 Test Generation

Recent developments in automatic test generation allow the generation of tests from the model of the software (e.g., the work at the University of Minnesota, the REACTIS tools from Reactive Systems Inc., and recent tools developed at The Mathworks) or from the code itself (e.g., work in heuristic search, concolic testing, and symbolic execution). These tools allow a user to specify a coverage criterion (for example, MC/DC), and the tools will find a suite of tests that provide the coverage. This approach can be an extreme case of white-box testing; the coverage criterion and the artifact to be tested (for example, the source code) are used to derive the test cases.

In principle, this represents a success for software engineering research: a mandatory (and potentially costly and time-consuming) engineering task has been automated. However, although there is some evidence that using structural coverage to measure the quality of black-box generated tests and to guide random test generation, the effectiveness of test suites can be

questionable. This is because the criteria for the test suites generated automatically to satisfy various structural coverages has not been established. Heimdahl's group at the University of Minnesota has investigated this issue in depth. In an early study, they found that test inputs generated specifically to satisfy three structural coverage criteria via counterexample-based test generation (a technique relying on a class of tools known as model checkers) were less effective than tests generated by picking random input data [65].

The results of a larger study [66] show that for most systems included in the study, automatically generated test suites generally perform significantly worse than randomly generated test suites of equal size. To complicate matters, test suites generated to satisfy structural coverage performed dramatically better than randomly generated test suites of equal size for one particular example that was specifically selected because its software structure was significantly different from the other systems. Safeware Engineering can draw one conclusion from these results: Automatic test generation to satisfy common structural coverage criteria (including MC/DC) simply does not provide effective test cases (even though they satisfy the coverage criterion in question). This observation indicates that satisfying even a highly rigorous coverage criterion, such as MC/DC, is a poor indication of test suite effectiveness; how the tests have been generated is of great importance when judging the quality of a test effort. The use of a coverage criterion, such as MC/DC as a metric to determine the quality of a test suite generated by hand (black-box tests from the requirements) or a randomly generated suite (as Chilensky and Miller recommend in their seminal work on MC/DC [67]), does appear to be effective.

These observations have serious implications and point to the dangers inherent in the increased use of test automation. The results from the poll conducted in phase 1, the "Alternative Approaches to Software Assurance Poll," indicate the need for more fundamental research in how a coverage criterion, test generation approach, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with the lack of knowledge of their effectiveness is worrisome, and careful attention must be paid to their use and acceptance going forward.

3.1.3.4 Loss of "Collateral Validation and Verification"

Based on observations related to test-automation above, at this point it is worthwhile to include a short discussion on the possible repercussions of increased automation in the software and systems engineering processes. Although Safeware Engineering supports increased automation, there are possible pitfalls (see section 3.1.3.3) for which manual processes are replaced by tools. For example, the kinds of tests generated by many test-generation tools would never have been derived by a human tester crafting tests based on years of domain expertise. Blindly replacing the latter with the former could have disastrous consequences (even if both test suites achieved MC/DC). Manual processes—design, coding, testing, or flight testing critical software—draw on the collective experience and vigilance of many dedicated engineers (primarily software engineers) who provide "collateral validation" [68] as they are working on the software. Experienced engineers designing, developing code, or defining test cases provide informal validation of the software systems; if there is a problem with the specified functionality of a system, they have a chance of noticing and taking corrective action. As engineers are replaced with automation, however, this validation process might be lost. For example, as the code for a system is designed and developed, problems with the models and requirements may be exposed

(the engineer doing the programming might have built a few of these systems in the past and, based on that experience, may detect some problems). If users rely on code generation from the models, this informal input from the engineer would be lost and possible undetected problems might make it through the development process. Similarly, as engineers craft test cases to satisfy MC/DC, careful thought goes into the process and problems in requirements, design, and code that may be detected serendipitously. With test automation, this process will be lost.

The concerns related to the possible loss of collateral validation are due to a dearth of studies addressing the effect of increased automation. There is evidence to show what may be lost (or gained) when replacing manual processes with automation. Some automation (e.g., automating the inspection for the adherence to coding styles or verifying that a model adheres to various constraints) can be highly effective and efficient. The aim with automation is to replace expensive and time-consuming manual activities with tools. The automation does not have to be perfect; rather, it simply needs to perform better than the manual processes it replaces. Nevertheless, as this discussion of testing above indicates, there is reason to adopt automation with some caution before the full implications are known. Safeware Engineering believes more research in this area is warranted to determine how to approve software developed, analyzed, and tested with automated tools.

3.1.3.5 Coverage Criteria Recommendation

Until a better method is developed, the authors would recommend the continued use of MC/DC, with the caveat that system reviewers should be directed to look at how MC/DC test cases are determined. Caution has to be exercised in using auto generation of test cases specifically to meet MC/DC until the approach to generation is fully explained and understood. Also, the system reviewer should be directed to look at the mechanism to determine whether the test was passed or failed (with preference given to mechanisms that make comparisons to at least some intermediate variable if the implementation is not strongly inlined).

4. PHASE 3 RESEARCH

4.1 TECHNICAL ADVANCES TO AID IN FINDING SAFETY CONSTRAINTS

4.1.1 STPA Hazard Analysis

In Phase 3 of this study, STPA was used to analyze a FGS using the requirements specification in the following paper: NASA/CR-2003-212426, Flight Guidance System Requirements Specification [69]. The authors of this paper built a requirements model using the Requirements State Machine Language Without Events (RSML^{-e}) modeling tool. Although the model represented the mode logic of a typical regional jet aircraft, it did not describe an actual or planned product nor did the report make any claims of accuracy or completeness of the specification. Several aspects of a full FGS were omitted, such as fault detection and recovery. Because the STPA analysis was conducted on the selected FGS from 2003 or before, the requirements and control structure diagram differ from newer FGS currently in use. However, the way in which STPA is used is still relevant.

4.1.1.1 FGS

The FGS is a component of the overall flight control system (FCS). It compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state.

The FGS subsystem accepts input about the aircraft's state from the attitude heading reference system, Air Data System (ADS), Flight Management System (FMS), and navigation radios. Using this information, it computes pitch and roll guidance commands that are provided to the AP. When engaged, the AP translates these commands into movement of the aircraft's control surfaces necessary to achieve the commanded changes about the lateral and vertical axes.

The flight crew interacts with the FGS primarily through the flight control panel (FCP), an example of which is seen in figure 3. The FCP includes switches for turning the flight director (FD) on and off, switches for selecting the different flight modes, such as vertical speed (VS), lateral navigation (NAV), Heading Select (HDG), and altitude hold (ALT) as well as approach (APPR), the VS/Pitch Wheel, and the AP disconnect bar. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of the selected mode's button.

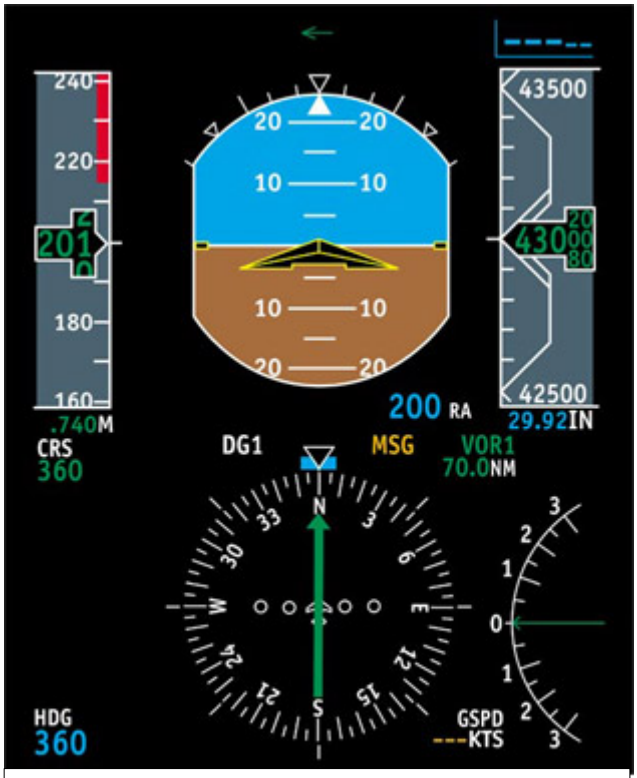
Figure 3. Example flight control panel (from NPR.org)

A few key controls, such as the Go Around (GA) button and AP disengage switch are provided on the control yokes and throttles and routed through the FCP to the FGS. Navigation sources are selected through the Display Control Panel, with the selected navigation source routed through the Primary Flight Display (PFD) to the FGS.

The FGS has two physical sides, or channels—one on the left side and one on the right side of the aircraft. These provide redundant implementations that communicate with each other over a cross-channel bus. Each channel of the FGS can be further broken down into the mode logic and flight control laws. The flight control laws accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. The mode logic determines

which lateral and vertical modes of operation are active and armed. These are annunciated, or displayed, on the PFDs along with a graphical depiction of the flight guidance commands generated by the FGS.

An image of the PFD is shown in figure 4. The PFDs display essential information about the aircraft, such as airspeed, VS, attitude, the horizon, and heading. The active lateral and vertical modes are displayed (annunciated) at the top of the display. The large sphere in the center of the PFD is the sky/groundball. The horizontal line across its middle is the artificial horizon. The current pitch and roll of the aircraft is indicated by a black wedge with a yellow outline representing the aircraft in the middle of the sky/ground ball (absent in figure 4).



Primary Flight Display

Figure 4. Example PFD (the "Primus 1000" PFD) from speedbirdair.com

The graphical presentation of the pitch and roll guidance commands on the PFD are referred to as the FD. The pitch and roll guidance commands are shown as a yellow outline of a wedge in the sky/ground ball, as shown in figure 4. When the AP is not engaged, these are interpreted as guidance for the pilot. When the AP is engaged, these indicate the direction in which the aircraft is being steered by the AP.

4.1.1.2 Applying STPA to a FGS

See section 3.1.1.1.1 for a review of the STPA process. The first step in analyzing the FGS is to identify system level hazards and review/identify safety constraints. For this analysis, the hazard—the aircraft drops below or climbs above the preselected altitude (PSA)—was used because it was also analyzed in an FTA performed by the same authors of the NASA/CR-2003-212426, Flight Guidance System Requirements Specification [69]. The results of the STPA analysis were compared to the FTA to demonstrate the benefits of using the STPA analysis when software, hardware, COTS, and systems analysis are required.

The second step in the STPA analysis is to describe the system control structure by building a control structure diagram, as shown in figure 5.

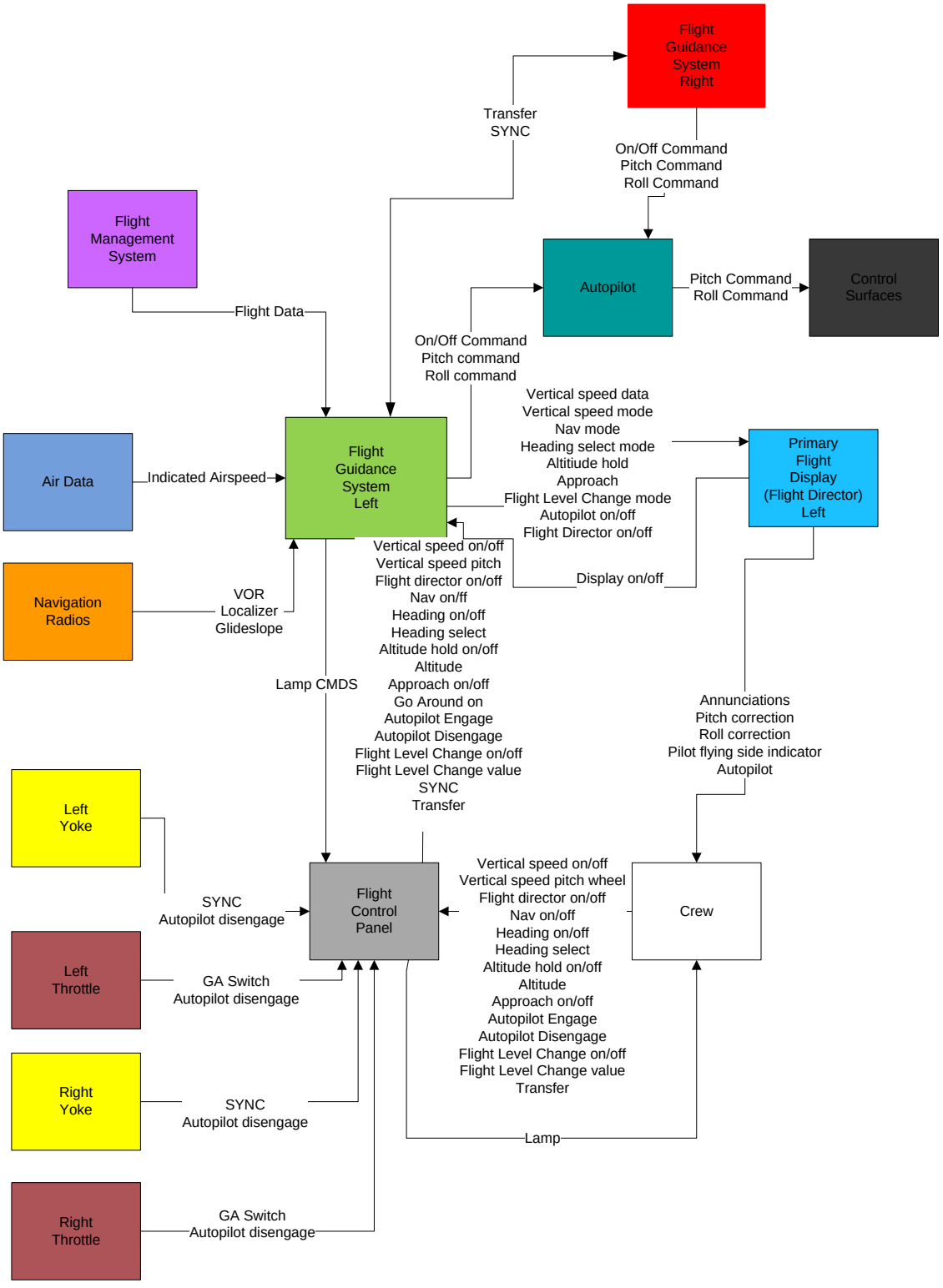


Figure 5. FGS control structure diagram

Each box in figure 5 represents a controller of the system, as defined by one of the four conditions:

1. The controller must have a goal or goals.
2. The controller must be able to affect the state of the system.
3. The controller must be (or contain) a model of the system.
4. The controller must be able to ascertain the state of the system [64].

A controller of the system could be a human (such as the pilots and air traffic controllers), software (such as the FGS or COTS), or hardware (such as sensors on the aircraft). Once the controllers of the system are identified, the interactions between the controllers are added to the diagram. The interactions could include commands, data exchanges, and feedback. The goal of an STPA analysis is to ensure that the system is controlled in such a way that it does not cause a hazard. To accomplish this goal, the interactions between the controllers are examined. Potential control actions that could lead to aircraft dropping below the PSA are identified. This is accomplished by determining if 1) a required control action between controllers of the system is not provided, 2) an incorrect or unsafe control action is provided, 3) a potentially correct or inadequate control action is provided too late or too early (i.e., at the wrong time), or 4) a correct control action is stopped too soon.

The components of the system that exert control are highlighted in the boxes in figure 5 and are described as follows:

- Flight Control Panel (FCP)–This panel is the primary device with which the flight crew interacts with the FGS. The crew can change vertical and lateral modes via this panel; turn the FD on and off; engage and disengage the AP; and transfer control between the left and right FGS. The Synchronization (SYNC) and AP disengage can also be activated on the yokes and sent to the FCP, and can also be routed through to the FGS. The pilots can also activate the GA switch and AP disengage on the throttles via the FCP.
- Autopilot (AP)–The AP commands the control surfaces based on the pitch and roll commands generated by the FGS.
- Navigation Radios–The FGS receives navigation information from several sources, including Very High Frequency Omni-Directional Range (VOR) for NAV, Localizer for precision lateral approach, and Glideslope for precision vertical approach.
- Air data system (ADS)–The ADS provides information about the aircraft's state, sensed from the surrounding air mass such as the pressure altitude and IAS.
- Flight management system (FMS)–The FMS's primary function is to manage the flight plan. The FMS contains flight data (waypoints, airways, airports, and runways) that are used by the FGS to provide guidance commands along a flight plan.
- Flight director (FD)–This panel displays the pitch and roll guidance commands to the pilot and copilot on the PFD.

4.1.1.3 STPA Results

The third step in the STPA analysis is to identify inadequate control actions that could lead to a hazardous state. Two of the controller interactions are examined in depth to demonstrate COTS/Legacy applicability and how the analysis can find missing requirements.

4.1.1.3.1 STPA Analysis Results for COTS Elements

For this analysis, the requirements for the FMS were not available, making it comparable to COTS or the legacy software component. The only known interaction between the FMS and the FGS was flight data. As for any component, the analysts examined what would happen if inadvertent flight data were received by the FGS. Inadvertent could mean either that the data arrived unexpectedly or that the data were wrong. If it cannot cause the hazard we are looking at, then we can move on to the next possible unsafe control action. In this case, regardless of how the FGS received the inadvertent flight data, the FGS could send incorrect commands to the pilot or the AP, which, if followed, could result in the aircraft flying to an incorrect altitude and the hazard needing to be examined. Of course, there is always the possibility that the data just happen to result in a command that does not cause a problem, but relying on happenstance is never a sound concept when designing safety-critical systems.

To help reviewers and developers understand that the scenario is hazardous, the analyst can include some possible causes. For this analysis, it is not necessary to attempt to list all the possible ways inadvertent flight data could be received. Also, the current requirements did not state what the flight data included. To account for this, a caveat was added at the beginning of the analysis of that interaction, stating: "It is unclear at this time what the Flight Data includes. This analysis is based on it including pre-selected waypoints and flight levels. As the requirements mature, this analysis will need to be revisited." The possible causes listed were "Flight Guidance System restart" and "Requirements for the Flight Management System are not part of this analysis." The second lets the reviewer know that the cause could lie in the FMS, but as with a COTS system, those causes are outside of his or her control or analysis. The analyst can look at the results and provide requirements that a COTS system must meet to be selected—or ways for the system currently under construction to mitigate potential hazard causes.

Finally, the analyst will look at ways to mitigate the issues. The suggested mitigation was:

> "If it is intended that the FGS receive flight data from the Flight Management System during flight, all data received should be time stamped. The pilot's role in approving course corrections subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined."

As examined more closely in the second example, the mitigation section is intended to spur thinking, not to be prescriptive or all-inclusive. For example, it might not be possible or sufficient to choose an FMS in which the data are time stamped, in which case the developers will need to come up with an alternative, such as a reasonableness check. Also, it is important to note that one mitigation might not be sufficient.

The analysts should choose a presentation format that best helps them and the developers to understand the analysis. For this project, the analysis was placed in appendix B and the areas that require attention are summarized in the main body of the report. See appendix B, page B-36, for the analysis of the application of the FMS to the FGS.

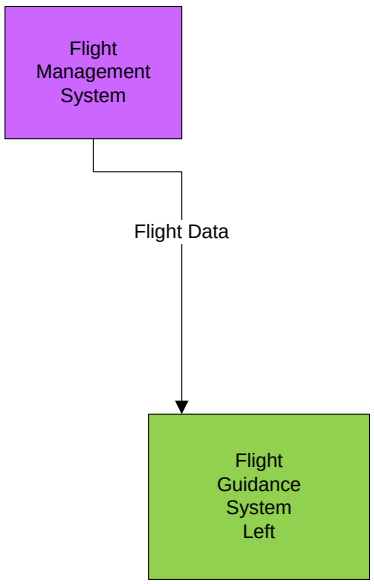


Figure 6. Interactions between the FMS and FGS

4.1.1.3.2 STPA Analysis Results for Missing Requirements: AP Status From FGS to Crew

In examining the feedback from the FGS to the crew via the PFD and FCP (appendix B, p. B-2), it was discovered that the feedback could be incorrect if the FGS did not actually know the status of the AP. No feedback was found from the AP to the FGS, and it was determined that the lamp was illuminated based on the desired state of the AP rather than on the actual state of the AP. The recommended mitigation is:

> "While there is an AP on lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the Lamp should be based on this feedback."

An analysis of this feedback was then performed (see appendix B, p. B-7) to determine what additional requirements this feedback required.

If the AP was unable to provide feedback on its health and status, the analysts and developers would need to determine another method for the FGS to determine if the AP was actually on. This might mean a requirement for new hardware, such as an altimeter, and new requirements for margins of error considered acceptable when the FGS attempted to determine if the AP was functioning as desired.

4.1.1.3.3 Summary of Recommended Mitigations for STPA

The findings for step three of the process of listing inadequate control actions are detailed in appendix B. A summary of all the findings is as follows:

- Flight Guidance System (FGS)–Requirements are needed to specify how commands from the off-side FGS are handled. Requirements are also needed to limit when the GA mode can be triggered in the FGS software. Requirements are needed to specify how the pilot transfer will occur and if any other modes or components will be affected in the FGS software. Requirements are needed to specify the FGS prioritization between the pilot and the AP commands. These requirements should include whether to leave the AP on if the throttle or yoke is active. Requirements do specify feedback to the pilot as to which FGS is in control, but requirements need to specify what happens during transitioning. Feedback to the pilot could help to mitigate this scenario. Requirements specify an AP lamp, but it is not based on a health and status message from the AP. An AP lamp should be based on feedback from the AP rather than the requested state of the AP. Requirements are also needed to specify what happens if the FGS loses power or is restarted. Requirements are needed to specify how the pitch values are checked for correctness.

  Although there are requirements that specify the triggering of the GA mode, it is unclear what the FGS software does with the GA command. More details need to be specified as to how the software handles the GA input command from the FCP.
- Vertical Speed Data–All commands and data should have expected rates and obsolescence defined. Components receiving data need to have clear requirements on how to handle data received at the wrong time. The requirements should specify how the system behaves during power loss, restart, and initialization. The requirements should specify how the software processes the FD's on/off commands in regard to the VS data. The requirements should specify what prioritization the FD on/off command should have, thereby reducing the possibility of the command being permanently delayed.
- Vertical Speed (VS)–The VS switch is intended to toggle VS on and off. Select_VS and Deselect_VS need to be mutually exclusive in the specification. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The VS light currently appears to be based on the actual rather than the requested state; however, the VS is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design. Requirements need to specify response times to pilot inputs.
- Pitch Wheel–Requirements need to specify how intended pitch wheel motions can be separated from inadvertent motions. At the least, changes in mode should be highlighted in such a way as to catch the pilot's attention, such as blinking until confirmed.

Requirements are needed to either limit the amount of pitch wheel change allowed in a given time or to at least make sure the pilot can tell that the wheel has been moved. Specify maximum response times to pilot inputs and obsolescence for all data and feedback. Provide feedback for all pilot inputs.

- Flight Level Change mode (FLC)–The FLC switch is intended to toggle FLC mode on and off. The macros Select_FLC and Deselect_FLC are not mutually exclusive and need to be corrected because both could be true at the same time. Because of the potential of confusion with the same input performing two different functions, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The FLC light currently appears to be based on the actual rather than the requested state; however, the FLC is internal to the FGS, and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design.
- ALT–The ALT switch is intended to toggle ALT mode on and off. The macros Select_ALT and Deselect_ALT are not mutually exclusive and need to be corrected because both could be true at the same time. The pilot must be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The Alt light currently appears to be based on the actual rather than the requested state; however, the ALT is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design. Signal-processing requirements should provide minimum and maximum system response times for pilot input.
- Vertical Approach Mode (VAPPR)–Vertical Approach Switch is intended to toggle VAPPR mode on and off. The macros Select_VAPPR and Deselect_VAPPR are not mutually exclusive and need to be corrected because both could be true at the same time. Because of the potential of confusion with the same input performing two different functions, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The APPR light currently appears to be based on the actual rather than the requested state; however, the APPR is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design. Signal-processing requirements should provide minimum and maximum system response times for pilot input.
- Transfer–Requirements should be considered to evaluate the correctness and reasonableness of the transfer input. Signal-processing requirements should provide minimum and maximum system response times for pilot input. Ensure that there is clear notification to pilots of which side is in command. Care must be taken that this reflects actual rather than requested state. The pilot flying arrow currently appears to be based on

the actual rather than the requested; however, the pilot flying is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design.

- Vertical Go Around (VGA)–Select_VGA and Deselect_VGA need to be mutually exclusive in the specification. The pilot must be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The GA light currently appears to be based on actual rather than requested; however, the GA is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough so as to prevent mistakes in the design, which will require a review of the design. Requirements are needed specifying when data can be used and when data should be considered obsolete. Signal-processing requirements should give minimum and maximum system response times for pilot input.
- AP–Although there is an AP on lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback. AP Engage and AP Disengage need to be prioritized. Requirements are needed that will prevent both the AP and the pilot from commanding controlled surfaces. Requirements are also needed on the AP to define behavior while transferring, during initialization, and during restart.
- SYNC–Determine a margin of difference that is acceptable: ensure clear notification to the pilots if the data displayed on the FDs differ by more than that acceptable margin. Implement limits on course correction commands sent to the AP. Ensure that there is clear notification to pilots of which side is in command. This could prove more of an issue if one of the flight displays is turned off or not working. Requirements should be considered to evaluate the correctness and reasonableness of this input. Signal-processing requirements should give minimum and maximum system response times for pilot input.
- Indicated Airspeed (IAS)–Requirements need to provide clear priorities of when to process IAS. For example, the requirements should state how long an overspeed condition can exist before it is registered. Requirements are needed to ensure an overspeed condition will be registered. Requirements must be provided to determine if the IAS is reasonable and to address when data become obsolete and how the FGS should respond. Requirements need to be added that address what to do if the IAS becomes obsolete.
- Flight Management System (FMS)–It is unclear at this time what information is sent from the FMS to the FGS. This analysis assumes the flight data include pre-selected waypoints and flight levels. As the requirements mature, this analysis will need to be revisited. If it is intended that the FGS receives flight data from the FMS during flight, all data received should be time stamped. The pilot's role in approving course corrections subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined, as does how the system will respond to partial messages.

- All Input and Output Commands–Requirements are needed to specify what to do if a partial command is sent or received. Requirements must be in place that specify what to do if an AP On command arrives before the system is finished implementing the AP Off command. At the least, feedback to the pilot must reflect the actual state of the system.
- Feedback From AP–At this point, health and status feedback from the AP to the FGS is needed but not provided. It is not known what form the feedback will take. Some common forms would be a query from the FGS to the AP; a continuous or discrete signal from the AP when it is on; a continuous or discrete signal from the AP indicating whether it is on or off; and regular messages from the AP to the FGS stating what the AP is doing. The following should provide guidance as the requirements are written.

4.1.1.4 STPA Conclusions

The STPA analysis of the FGS highlighted several strengths. First, analysis can include a variety of controllers to the system, including hardware, software (including COTS), and human elements. Many of these elements are often treated as separate systems because they may be developed by different manufacturers and their interactions may never be examined. For example, the navigational aids might be developed by a different company than the FGS. The interface between these two systems may be overlooked in traditional hazard analysis.

Second, the analysis of the interactions of the FGS with the FMS shows how STPA can be used to handle new software interacting with legacy software or COTS. The specification used in this analysis did not have FMS requirements. The analysis made assumptions about the outputs of the FMS to mitigate hazard causes associated with the FMS. In addition, STPA can be used early in the development to flesh out requirements. This was particularly evident in the interactions associated with the AP in which the analysis indicated that feedback from the AP to the FGS was missing. The suggested mitigation was “to add feedback.” Because STPA is early in development, it is not too late to add a requirement for the system and analyze the data to determine if the plane appears to be controlled by the AP. If this mitigation were chosen, it might be necessary to add a requirement for additional hardware, such as a gyroscope or GPS. While fleshing out the requirements, it also helped allocate the requirements for components, regardless of whether they were hardware or software.

Third, STPA is scalable. The analysis has been performed on large-scale, complex, distributed systems. STPA has been used at the Missile Defense Agency for a Non-advocate Safety Assessment of the Ballistic Missile Defense System. The analysis can be done on a component, subsystem, or system of systems. The analyst could analyze one component or a set of components depending on the scope. In this analysis, the FCP—although it was one component of the system—was treated as part of the FGS, allowing for the focus on the crucial interactions between the crew and the system.

STPA provides a powerful tool for focusing the safety effort and the requirements elicitation process, as long as developers do not get sidetracked thinking they have to figure out all the ways the controller could fail or that the mitigations listed are the only ones available.

To fully demonstrate the benefits of STPA, further application of this analysis tool should be studied. The authors recommend teaming up with aviation partners to use STPA on a project currently in development. Assess results of the STPA analysis to determine applicability in ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.

4.1.1.5 STPA vs. FTA

One way to assess the benefits of an STPA is to compare it to other traditional hazard analysis techniques currently used in industry. An FTA was performed on the FGS and its results were discussed in the paper Software Safety Analysis of a Flight Guidance System [70]. An FTA is a feedback technique for which one starts with the system level hazards and attempts to work backward by identifying all possible causes of the hazards. Although the name implies that the technique is limited only to faults, it should be emphasized that FTA is a general, visual technique that is used to trace higher level events (such as hazards) down to their contributing events. These contributing events could be failures, or errors, in addition to faults. The authors of the FTA found 23 base events that could generate one of the four major hazards associated with FGS functional failures. For example, their findings listed:

- FGS-FD communications channel failure
- FGS output overwhelms FD
- FGS sends incorrect values
- Error in flight control level algorithm

In contrast, the STPA made 70 recommendations regarding specific requirements that should be added to the specification.

The difference between an FTA and STPA is in the approach. With an FTA, the analyst is forced to determine all the ways in which the system could fail. This is difficult to do early on in the development process when the design has not yet been solidified or is dependent on the experience level of the analyst.

The benefit of an STPA is that it provides a roadmap to get the analyst thinking of ways the elements of the system could become uncontrolled based on the research of past accidents. The analysts can ask themselves about the following four states of a particular interaction among controllers to assess which of the following states are applicable:

1. A required control action between controllers of the system is not provided.
2. An incorrect or unsafe control action is provided.
3. A potentially correct or inadequate control action is provided too late or too early (i.e., at the wrong time).
4. A correct control action is stopped too soon.

The analysts of the FGS STPA have limited experience with aviation guidance software, yet were able to make specific recommendations regarding requirements that needed to be added to

the specification. In addition, a cost savings can be obtained if the requirements are analyzed early in the design, when changes minimally impact cost. This is achieved by finding errors during the requirements phase, when changes can be made relatively easily as compared to later stages of development.

4.2 CASE STUDIES ON COST SAVINGS RELATED TO MODEL-BASED DEVELOPMENT

4.2.1 Model-Based Development

Software development in critical systems has traditionally been largely a manual endeavor. Validation that the right system is being built has been achieved through requirements, model inspections, and reviews. Verification that the system is developed to satisfy its specification is achieved through inspections of design artifacts and extensive testing of the implementations. In critical software systems, the V&V is particularly costly and may consume 50%–70% of the software development resources. If techniques could be devised to help reduce the cost of V&V, dramatic cost savings could be achieved. The trend toward model-based development recognized in DO-331 is one attempt to address this goal [71].

In model-based development, the development effort is centered on a formal or semi-formal model of the proposed software system. Through manual inspections, formal verification, and simulation and testing, it can be determined that the model possesses desired properties. The implementation can then potentially be automatically generated from this model. There are currently several tools that aim to provide part or all of these capabilities. These tools include, for example, Simulink and Stateflow from Mathworks [72, 73]; Esterel and SCADE from Esterel Technologies [74, 75]; Statemate [76]; and various UML tools from other vendors.

In DO-331 [77], models are defined as follows:

> "A model is an abstract representation of software aspects of a system that is used to support the software development process or the software verification process."

In this context, only models with the following characteristics are considered:

- The model is completely described in an explicitly identified modeling notation. The modeling notation may be graphical/textual.
- The model contains software requirements/a software architecture definition.
- The model is of a form or type that is used for direct analysis or behavioral evaluation as supported by the software development process or the software verification process.

The standard recognizes that models may bring benefits in the following areas:

- Providing unambiguous expression of requirements and architecture.
- Supporting the use of automated code generation.
- Supporting the use of automated test generation.
- Supporting the use of analysis tools for verification of requirements and architecture.
- Supporting the use of simulations for partial verification of requirements, architecture, and executable object code.

At this point, it is worth mentioning that models and formal verification are closely coupled; without models, there can be no formal verification. Even for verifying code (which is really a model of the intended execution), models of the desirable behavior are needed (the specification model in the language of DO-333). Therefore, if formal verification takes place, there will be aspects of model-based development present—and model-based development will most likely be used in conjunction with some form of formal verification.

Based on research of the published literature, interactions with colleagues involved in model-based development, and the authors' own experiences, the authors feel model-based development holds enormous promise and that significant potential cost savings can be realized through its implementation. In the following sections, the possibilities are reviewed and, where appropriate, the potential pitfalls of model-based development are discussed.

4.2.1.1 Unambiguous Expression of Requirements and Architecture.

As pointed out in the introduction, missing, incomplete, inconsistent, and erroneous requirements have a major impact on schedule, cost, and safety. Therefore, addressing the requirements problem is of critical importance. Model-based development provides a mechanism for rigorously and unambiguously capturing the often complex requirements for critical aviation-related software systems.

To show how practicing engineers could use formal modeling, Nancy Leveson led a team that successfully captured the required behavior of TCAS II, a collision-avoidance system installed on all commercial aircraft seating more than 30 passengers, which uses a research language called the Requirements State Machine Language (RSML) [78]. The successes with the A-7 Corsair and TCAS II projects can be credited solely to the modeling efforts. There was neither tool support for analysis of the models nor any support for code generation or test generation, therefore demonstrating that use of modeling techniques with well-defined notation has major benefits in producing clear, unambiguous, and correct requirements. The modeling efforts themselves forced stakeholders to clarify the requirements (no room for ambiguity) and the models served as outstanding communication tools allowing the stakeholders to address disagreements in terms of the systems' desired functionality. Similar results have been observed in other modeling efforts for which the act of modeling enforces clarity and encourages a rigorous and comprehensive requirements elicitation process.

4.2.1.2 Automated Code Generation

Part of the model-based development vision was that after a model has been created, production code could then be generated directly from the model. This code-generation capability was seen as the main advantage of model-based development and has been heavily marketed through vendors such as Mathworks and Esterel Technologies. Given that some manual coding may now be eliminated, various cost savings were claimed in the literature [79–81]. The following paragraphs provide an evaluation of these claims and some recommendations.

Information on how much time and effort is spent on actual coding in critical avionics applications is notoriously difficult to obtain because it is guarded as a corporate secret. The following discussions are based on the small amount of data available and on anecdotal experience obtained through discussions with several avionics manufacturers.

After reviewing the internal literature regarding the project money spent solely on coding in software development costs, it is not clear how much savings can be realized by adopting automated code generation versus traditional coding. Some industry estimates indicate the coding effort to be in the range of 20%–25%. In the authors' opinion, it is closer to 5%. If the coding effort is in the range of 20%–25%, the use of automated code generation can reduce this effort by half (there will still be coding needed to integrate the generated code with the operating environment). In some instances, with certain systems, there may be the potential to eliminate the coding costs entirely, leading to cost savings of 10%–25%. If this is actually closer to 5%, as the authors contend, the cost savings realized through the adoption of automated code generators is marginal.

At this point, a distinction will be made between two directions in code generation: (1) trusted code generation, for which there is some argument (in the form of a certification kit) that allows one to qualify the code generator as a development tool and (2) regular code generation, for which the generated code is treated as if it was hand-coded.

In the case of a qualified code generator, the concept is that all V&V activities would take place in the model domain and the generated code could be deployed without any additional testing. The initial cost savings claims using this approach were extraordinary [80]. For example, Esterel Technologies made the following claims regarding development using their SCADE tools and its verification tool, the Design Verifier [80]. By using code generation, the manual coding (comprising 20% of the development effort) could be eliminated for 20% cost savings. If the code generator were qualified and used as a development tool, the costly unit testing—estimated to be 30% of the development costs—could be eliminated for total savings of 50%. If, in addition, the design verifier was relied on to verify that certain system properties were true, system testing and integration testing could be reduced for an additional 10 percentage point savings. Therefore, by using their tool suite with a qualified code generator, they claim savings of 60%. Although there are cost savings to be had in code generation, in the authors' opinion, these cost-savings estimates are somewhat unrealistic and based on flawed assumptions.

When relying on a trusted code generator, the only guarantee (if the code generator can be trusted) is that the generated code will implement exactly the same function as the model from which it was generated. There is no guarantee that the model actually implements its

requirements. The test obligations imposed by DO-178C on the source code (e.g., MCDC) do not go away, but simply move into the model domain. Instead of testing the source code to demonstrate that it implements the requirements, the model must be tested to demonstrate that it implements the requirements.

With respect to explaining the cost savings, it is commonly understood that the models were requirements and were subject to the requirements V&V efforts outlined in ARP 4754 [82]. When entering the DO-178B process (it was DO-178B as opposed to DO-178C at the time of discussion), the requirements are, by definition, correct. If code can now be generated that is guaranteed to meet the requirements (through a trusted code generator), the source code will, by definition, meet its requirements. By this reasoning, the rigorous testing requirements in DO-178B could be eliminated because there is no such V&V requirement in the ARP 4754 domain. In the authors' opinion, this reasoning is not defensible. If one can generate source code from a model, the model is simply a program in a different notation and should be treated as such. Either the model or the generated source code must be tested up to DO-178B(C) standards and no dramatic cost savings can be realized directly through code generation. Current tool suites support this type of testing, and the testing capabilities can be effectively used in model-based development. Of concern is that the claims of cost savings are not credible. DO-331 explicitly addresses this issue and requires that the generated code be treated as any other code.

With respect to the claimed cost savings of using formal verification to reduce some testing efforts, there is some promise in this area, which is further discussed in section 4.3.

The cost of qualifying a code generator is high, and the time it takes to modify and re-qualify such a tool is rather lengthy [80]. Given the high cost and marginal benefits of a qualified code generator, it seems to be more beneficial to treat a code generator as an unqualified tool (far cheaper and more flexible than a qualified code generator) and instead realize cost savings through the increased clarity and rigor of models, the possibility for cost-savings test generation from the models, and the simulation and formal verification of the formal models before coding commences, thereby significantly reducing or eliminating costly rework.

4.2.1.3 Automated Test Generation

In the context of model-based development, there are several different testing activities. The testing tasks in this framework tend to be divided into two distinct activities: model validation testing and conformance testing. The former activity ensures that the model captures the behavior wanted from the software. The model is executed and tested to prove that it satisfies the true software requirements. The latter activity ensures that the code developed from the model (either manually or automatically through code generators) correctly implements the behavior of the model (i.e., the implementation conforms to the specification and no "bugs" were introduced in the coding stage or by the automated code generator).

To accomplish the model validation activity in model-based development, a set of requirements-based functional tests was currently developed from the natural language (or formally defined) high-level requirements to evaluate the required functionality of the model. The tests used in this step are developed from the requirements by domain experts, much like requirements-based tests in a traditional software development process. The model validation

testing is generally intended to validate the requirements and the model. During the validation, the following questions need to be answered:

1. Do the scenarios designed to test the requirements make sense to the domain experts and make sense in the context of the systems?
2. Does the model behave correctly with respect to these requirements?

A validation test is intended to show that a requirement has been met and to illustrate how it is met. There are verification techniques (e.g., model-checking) that in some cases can be used to verify that a model satisfies its requirements. Such proofs, however, do not illustrate why and how the requirements hold, which will be discussed in detail in section 4.2.1.4. Therefore, model-verification techniques and model-validation testing must be used as complementary techniques; even if it can be proven that a requirement is met in the model, several test cases are typically needed to illustrate how and why this is the case.

Recent developments in automatic test generation allow the generation of tests from models of the software (e.g., the REACTIS tools from Reactive Systems Inc., recent tools developed at The Mathworks, and research efforts at various universities) or from the code itself (e.g., work in heuristic search, concolic testing, and symbolic execution). These tools allow a user to specify a coverage criterion (e.g., MC/DC) and the tool to explore a model (or source code) and generate a suite of tests that provide the desired coverage.

These test-generation capabilities can be successfully and cost effectively used to ensure, for example, that the code generated from an untrusted code generator conforms to the behavior defined in the model from which the code was generated.

There are little data related to the cost-effectiveness of model-based development (or any other software-engineering activity for that matter). Research has shown, however, that test generation scales well and can generate high levels of coverage in a short amount of time. Therefore, automatic test generation is likely to be cost-effective. Nevertheless, there are some cautionary observations that must be considered.

4.2.1.3.1 Test Generation Pitfalls

In principle, the scalability and efficiency of test generation from models (or source code) represents a success for software engineering research and a powerful adoption in practice: a mandatory, and potentially extremely costly and time-consuming, engineering task has been automated. However, although there is some evidence that structural coverage is used to measure the quality of generated tests and to guide test generation, the effectiveness of test suites automatically generated to satisfy various structural coverage criteria has not been established. Heimdahl's group at the University of Minnesota has investigated this issue in depth. An early study revealed that test inputs generated specifically to satisfy three structural coverage criteria via counterexample-based test generation (a technique relying on a class of tools known as model checkers) were less effective than randomly generated test inputs in showing that the implemented code actually satisfied the model from which it was derived [65].

The results of a larger study [66] showed that, for most systems included in the study, automatically generated test suites performed significantly worse than random test suites of equal size. To complicate matters, in one example that was specifically selected because its software structure was significantly different from the other systems, test suites generated to satisfy structural coverage performed dramatically better than random test suites of equal size.

One conclusion can be drawn from the conclusions of Safeware Engineering: automatic test generation to satisfy common structural coverage criteria (including MC/DC) is not guaranteed to provide effective test cases (even though they satisfy the coverage criterion in question). This observation indicates that satisfying even a highly rigorous coverage criterion, such as MC/DC, is a poor indication of test suite effectiveness; how the tests have been generated is of great importance when judging the quality of a test effort. The use of a coverage criterion—such as MC/DC as a metric to determine the quality of a test suite generated by hand (black-box tests from the requirements) or a randomly generated suite (such as Chilensky and Miller recommend in their seminal work on MC/DC [67])—does appear to be effective.

These observations have serious implications and point to the dangers inherent in the increased use of test automation. The review of relevant literature conducted under this contract indicates the need for more fundamental research regarding how a coverage criterion, a test-generation approach, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with the lack of knowledge concerning their effectiveness is worrisome; careful attention must be paid to their use and acceptance.

4.2.1.4 Analysis Tools for Verification of Requirements and Architecture

Because the models used in model-based development have well-defined semantics, it is possible to use various types of tools for analysis of the models. In this context, verification or analysis to be conducted without executing the models will be considered (as opposed to testing or simulation that execute the models). Such analysis can be divided into two categories: (1) analysis for desirable properties internal to a model (e.g., consistency, freedom from deadlock, and completeness); and (2) verification that one model conforms to (often referred to as “implements”) another model (e.g., the design model implements the specification model).

Analysis of internal desirable model properties can be viewed as an extended type of well-formedness checking beyond what would be done for simple syntax- and type-checking. This type of analysis has been explored in the context of various modeling languages. The notion of completeness and consistency of requirements and models was first defined by Heimdahl and Leveson [83]. Completeness in this context means that the model has a response specified for each possible situation, and consistency means there are no conflicting requirements in the model. Analysis of completeness and consistency has been implemented with research tools such as Nimbus [84], SCR [85], and the Linear Time Temporal Logic (LTL) tool from Fondazione Bruno Kessler. Commercially, this type of analysis has had its best success in static analysis tools for source-code checking for properties such as the freedom from NULL pointer dereferencing, division by zero, and possible numeric overflow and underflow. Note that this type of analysis can help point out problems with a model (or program) when the model is syntactically incorrect, type incorrect, or inconsistent in some manner. However, if the model

passes all well-formedness tests, it is possible that it is correct, although there is no guarantee. Similarly, a program (source code) with no possible NULL pointer dereferencing or division by zero, for example, could be correct, but it is not known for sure. All that is known is that it is not obviously wrong.

As with other aspects of model-based development, there are little data available to judge the potential of costs savings when using tools for verification in the modeling domain. One notable exception is the formal verification efforts undertaken by Rockwell Collins Inc., first as a research project and later adopted in software production for air transport avionics [86, 87].

In their work, the Rockwell Collins teams developed a successful methodology leveraging model-based development and, in particular, the verification capabilities in model-based development. In the first phase of requirements elicitation, they followed a traditional process and collected the system requirements as informal "shall" statements. The next step was modeling, for which a model of the system to be developed was created by hand. This model was intended to exhibit the behavior informally stated in the "shall" requirements statements.

Throughout creation of the model (expressed in a notation called RSML^{-e} in the research effort and Simulink/Stateflow in the production projects), the execution/simulation capabilities in the modeling environments were used to informally confirm that the models behaved as expected. To further increase the confidence in the models, the Rockwell Collins team made extensive use of formal verification. In this stage, they manually translated the shall statements into formal properties stated over the model expressed in either computation tree logic (CTL, a temporal logic used by the verification tools used in Rockwell Collins's work) or LTL (another logic used by the tools) and used a class of verification tools known as model checkers to investigate whether or not the model satisfied the formalized "shall" requirements. If a property was found to not hold, the necessary changes were made to either the model or the property (representing a "shall" requirement). If it was discovered that the model was behaving correctly (the model in this context can be considered a design model in the language of DO-331) but the property was wrong (here a property can be considered a specification model in the language of DO-331), the property was investigated to determine if it had been formalized erroneously (i.e., it did not capture the true intent of the "shall" requirement from which it was derived) or if the "shall" requirement itself was incorrect. In the former case, the property was modified and the verification repeated. In the latter case, the property and the "shall" requirements were closely investigated and any changes were fed back to the earlier stages in the development effort (because any changes to the "shall" requirements might have repercussions with respect to other components in the systems architecture). Additional details regarding the verification efforts related to the Rockwell Collins experiences will be discussed along with formal verification in section 4.3.

In the production efforts, Rockwell Collins did not use any of the verification results for certification credit; the verification was performed solely to improve the quality of the "shall" requirements (and the requirements formalized in the specification model) and the design model. After the verification process was completed, the requirements and model were both of extremely high quality, the implementation phase was consequently straightforward, and Rockwell Collins realized significant cost savings when the code derived from the models passed

through the rigorous testing process with very few faults found, resulting in minimal rework. The cost savings came from producing a correct software system from the beginning with no rework of code and tests, and no late requirements changes that needed to be fed back to an earlier process.

4.2.1.5 Model-Based Development Summary and Recommendations

Model-based development techniques have made inroads in the development process and have demonstrated the potential for substantial cost savings and quality improvements. Therefore, Safeware Engineering is supportive of increased usage of model-based development.

As with all new technologies, however, there are possible pitfalls that may impact the effectiveness of the approach and imperil future adoption. All models are abstractions, the choice of modeling notation is crucial, and the results from automation can be deceiving. In the following sections, a short discussion of these issues and recommendations for future research directions are provided.

4.2.1.5.1 Abstraction in Modeling

According to statistician George Box, all models are wrong, but some are useful. That is, all models are abstractions of the reality the model represents and will, by definition, omit details from this reality. Hopefully, the omissions and simplifications are sound in that the omissions do not affect the conclusions drawn from the model. Unfortunately, in Safeware Engineering's experience, it is quite easy for an organization to begin accepting the model as reality and not adequately question the conclusions drawn from simulation, analysis, and verification in the model domain. This is not a flaw of modeling (without abstractions and omissions, the models would not be useful for their purpose), but a problem pertaining to how the models are used in organizations, particularly organizations that are new to model-based development and may not be familiar with the limitations of the approach.

It is worth noting that abstractions can come from several sources. The engineers may omit detail from a model and focus only on, for example, the timing aspects of a system for schedulability analysis. This abstraction is explicit and must be justified if the results are used in the development process or for certification credit. Other abstractions are implicit in the choice of modeling notation. For example, SCR makes the assumption that only one monitored control variable changes at any time; two variables changing simultaneously are not allowed by the modeling notation semantics. Similarly, many notations, such as Statecharts, SCADE, and Stateflow, make the assumption that the computation of the model is infinitely fast as compared to the environment in which it is executing; therefore, the model will always complete its computation before the environment with which it communicates can change. These are both useful abstractions from reality (for which two quantities obviously can change at the same time and computations take time) because they simplify the semantics of the languages and allow for certain types of analysis. If any results derived from analyzing the models expressed in these languages are to hold in the system the models represent, one must ensure that these implicit assumptions hold in the actual system. Explicit and implicit abstractions (in particular, if not well documented) can have a serious impact on the validity of any results attributed to inspection, testing, and analysis of a model.

Most books related to modeling and model-based development focus on the notations and how they can model various aspects of a system. Missing is the guidance regarding how to use modeling, how to understand and document abstractions, and how to determine whether or not an abstraction is sound (i.e., it does not exclude potential safety-related behaviors in the model). A notation agnostic guidebook on the modeling in critical software systems would be a useful resource for the community. This guidebook would be different than the "Formal Methods Case Studies" in preparation by Rockwell Collins. This report is a great resource to illustrate how various formal notations can be applied, but it does not discuss the various traps and pitfalls related to modeling.

4.2.1.5.2 Choice of Modeling Notation

Inspections will always be an invaluable tool when determining whether a set of properties accurately captures the desired requirements and whether a model describes the system the customer wants. Consequently, any modeling language must be readable and understandable enough so that all stakeholders can be involved in the inspections process. Unfortunately, there has been a traditional dichotomy between formality and readability; developers of formal modeling notations did not concern themselves with readability to any large extent and developers of modeling notations widely used in industry were generally not concerned with rigorous semantics. In Safeware Engineering's experience, and based on the studies cited herein [5, 78, and 88], the syntax and semantics of a modeling language are critically important for the success of a project.

Little investigation has been done regarding the readability and understandability of modeling notations. A first step can be found in Zimmerman et al. [89], in which a pilot experiment was performed indicating that complex conditions expressed in tabular notations, as found in SCR, RSML, and SpecTRM [90], might be more readable than the logic-gates found in notations (such as Simulink and SCADE). They also found that the notion of internal events in state machines (as found in Statecharts, Stateflow, and UML) may hinder readability as compared to the data flow semantics found in, for example, SCADE, SCR, and SpecTRM. Ironically, Statecharts, SCADE, Simulink, Stateflow, and UML are by far the most influential notations in industry practice.

Syntax and readability are not the only aspects of a modeling notation that need to be considered. As is recognized in DO-331, although the modeling efforts themselves are important, having a notation that is amenable to code generation, test case generation, and formal verification is crucial to attain the full benefits of the modeling efforts. Safeware Engineering recommends careful evaluation of the syntax, semantics, and support for automation before a modeling notation is incorporated into a process. Investing resources in modeling to clarify requirements, but winding up with models that cannot be leveraged in later stages of the development process, may lead to wasted efforts.

Significantly more research into modeling language design and usability is needed so the research community can better influence the next generation of modeling languages and tools.

4.2.1.5.3 Model-Based Development Automation Can Be Deceiving

Automation can be misleading and may provide results that give the illusion of quality when, in reality, there are serious flaws with the process.

Until a better method is developed, Safeware Engineering would recommend the continued use of MC/DC with the caveat that system reviewers should be directed to look at how MC/DC test cases are determined (auto generation of test cases specifically to meet MC/DC must be viewed with certain suspicion until the approach to generation is fully explained and understood) and what the mechanism is for determining if the test was passed or failed (with preference given to mechanisms that make a comparison to at least some intermediate variable if the implementation is not strongly inlined).

4.3 FORMAL VERIFICATION

The RTCA Standard DO-333 [91] defines formal methods as follows:

> "Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The mathematical basis of formal methods consists of formal logic, discrete mathematics, and computer readable languages. The use of formal methods is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to establishing the correctness and robustness of a design."

A formal notation enables the precise and unambiguous capture of the requirements or a model of software (or systems); advantages with formal notations are similar to the ones related to model-based development and are discussed in the section on model-based development above. Given a formal model, DO-333 points to numerous possible usages. For example:

- Unambiguously describing requirements of software systems
- Enabling precise communication between engineers
- Providing verification evidence of the compliance of one formally specified representation with another
- Demonstrating freedom from exceptions
- Demonstrating freedom from deadlocks

For example, "freedom from exceptions" and "deadlock" are cases of demonstrating that one formal model satisfies some desirable property expressed in another formal model. Therefore, with an appropriate and understandable notation, formal modeling can provide the advantages of the model-based development discussed earlier in this report and augment those advantages with powerful verification techniques.

In the context of this report, formal verification is defined as a technique to demonstrate that the model *M* satisfies the property *P*. For example, the model *M* may be a concurrent program and the property *P* may be freedom from deadlock. In the context of model-based development, the

model *M* could be a formal design model, and the property *P* could be a formal specification model. It is exceedingly rare that a property *P* holds in model *M* without making additional assumptions (referred to here as *A*). For example, a property might only hold in a model if only one input signal at the time changes or all inputs are within a certain range. Therefore, in general, using formal methods, it can be established that given assumption *A*, model *M* satisfies property *P*. The importance of properly validating assumptions, properties, and models in formal verification are discussed later in this section.

DO-333 makes a distinction between three categories of formal analysis methods:

1. Deductive methods that rely on a mathematical argument (i.e., a proof) that *M* together with *A* satisfies *P*. There are powerful interactive theorem provers that assist in the construction of these proofs, but manual effort is typically required.
2. Model checkers that rely on exhaustive exploration of the possible behaviors of a model to determine if the property is satisfied in all cases.
3. Abstract interpretation that creates an abstract description of the model under investigation to assist in the determination of whether or not the property holds.

The exact approach to formal verification is not important when discussing its usage. The different techniques have their pros and cons (e.g., model checkers can be easy to use and provide an explanation as to why the property does not hold if that is the case). However, model checkers may not scale to models of significant size and some are limited to finite state models. This report does not distinguish between different techniques—they all achieve the same goal (i.e., demonstrate that *M* and *A* satisfy *P*)—but they might go about this demonstration in different computational ways. One common trait pointed out in DO-333, however, is that all techniques considered here must be sound (i.e., they will only report that *P* is satisfied in *M* if that is really the case. They will never say that *P* is satisfied when, in actuality, it is not).

As mentioned throughout this report, data supporting the effectiveness and potential cost savings of formal methods are notoriously rare. Most published case studies are conducted to demonstrate that a technique is capable of scaling to larger systems, is computationally more efficient than previous approaches, or is capable of handling features of the formal descriptions other approaches are not (such as infinitely large state spaces or real variables). This research has revealed few publications that provide unbiased support for the cost effectiveness of formal verification. This report singles out two radically different applications of formal verification in the domain governed by DO-178B/C. First, Rockwell Collins' efforts will be discussed regarding the model-based project referenced earlier in this report. In this project, Rockwell Collins extensively used formal methods to improve the quality of the software requirements and reduce costs through reduced rework—no certification credit was taken for the verification efforts. Second, several European companies have eliminated unit testing for highly critical software by replacing it with formal verification of the source code. In this case, they are taking certification credit for the verification by substituting MC/DC testing with formal proofs.

4.3.1 Reducing Rework and Test Effort Through Formal Verification

Consider the Rockwell Collins model-based development experience discussed in section 4.2.1.5 and covered in several publications [86, 87]. Although Rockwell Collins applied the same

overall approach to several projects, examples were used from the research effort because more details of the work are available in the published literature. The considered Rockwell Collins projects are FGS models based on a fielded product [86], an FCS for business and regional jets [92], and several aspects of the display manager for a new air transport aircraft [87]. The other projects had the same experiences, but little detail is available because of the proprietary nature of the models and requirements.

The subject of the research study was the FGS, which is a component of the overall FCS in a commercial aircraft. The FGS can be broken down to mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state, and compute the pitch and roll guidance commands. In the study, Rockwell Collins used the mode logic for horizontal and vertical modes. They collected the system requirements as informal "shall" statements. In this project, these requirements were relatively mature and thought to be well-understood. The next phase, modeling, consisted of constructing an executable model by hand that was believed to exhibit the behavior informally stated in the "shall" statements; in the research effort, they used a formal research notation called RSML^{-e} [86]; in the production efforts, they used Simulink and Stateflow [87]. As the models were built, the formal nature of the notations forced precise capture of the behavior of the system under development, and numerous ambiguous, unclear, and inconsistent informal requirements were discovered and corrected. In addition, throughout the creation of the models, they were executed to informally confirm that they behaved as expected. These models can be considered design models in the DO-331 sense.

In the formal verification phase, the informal "shall" statements were manually translated to formal CTL or LTL statements (depending on what verification tools were used). This translation process was undertaken by two independent engineers. If there was any discrepancy in the formalization, a small expert committee convened to resolve the issue, to settle on one formalization, and to possibly revise the "shall" statement to make it more clear and precise. Again, the formalization process helped further improve the informal requirements.

Rockwell Collins then used model-checking technology to confirm whether the property held over the model. In the research project, research tools were used that had been developed at the University of Minnesota, which translated the formal model to an open-source model checker (NuSMV). In the subsequent production projects, they used a proprietary translation infrastructure called Gryphon, which was capable of targeting several verification back-end tools.

The process of creating a design model from the English prose requirements led the team to go back and clarify the English statement of the requirements. In the same way, translating the English statements into a formal specification model also prompted them to go back and clarify the English statement. In addition, the formal verification that the model satisfied the requirements (formalized as CTL or LTL properties) led to additional insight into the requirements specification problem.

To illustrate the discovery process, a published example was used [86, 93]. All but four of the requirements used in this project were proprietary, so a limited selection was available to use as an example. Consider the well-validated and non-controversial requirement below:

> "If Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the Flight Control Panel."

After formalization into CTL, this property did not verify in the model. The model-checker pointed out two ways in which this property could be violated. First, if another event arrived at the same time as the HDG switch was pressed, that event could preempt the HDG switch event. Second, if this side of the FGS was not active, the HDG switch event was completely ignored by this FGS side. There are two FGSs installed on an aircraft; one is active and the inactive one operates as a hot spare.

The information provided by the verification (the model checkers used here provide what is called a counterexample, essentially a sequence of inputs that will lead to a violation of the property) led to a modification of the requirement:

> "If this side is active and Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the FCP (providing no higher priority event occurs at the same time)."

The description of the requirement has the distinct advantage of providing a more accurate description of the system's intended behavior. A clear definition of what constituted a "higher priority" event needed to be added to the requirements.

During this verification process, the team found that the process of proving the properties forced them to go back and modify/clarify a large number of the English requirements (these were the already clarified requirements derived during the formalization process). Consequently, the corresponding formal specification properties also had to be modified.

When developing formal models of any substantial system, the models will most likely be incorrect with respect to the real needs of said system. In this case, three complementary artifacts—informal English language requirements, requirements formalized as a specification model using LTL or CTL properties, and an executable formal design model—served to check each other in a rigorous validation process. Had only the executable design model been built and validated through testing, chances are significant flaws would have remained. Similarly, had a formal specification model been available and a correct-by-construction tool used that would have helped refine the specification model to an implementation, the implementation would certainly have been grossly incorrect with respect to the system's real needs (given that the specification model needed significant changes as the Rockwell Collins verification process progressed). From these experiences, it seems clear that a rigorous validation process for both design models and specification models (formal properties) must be in place to ensure that any formal artifacts serving as the basis for downstream automation—such as code generation or test generation—are correct.

The Rockwell Collins projects, the interactive refinement of the natural language requirements, the specification model formalizing these requirements, and the design model serving as a basis for the development helped provide a design model essentially free from faults. Consequently, any requirements-related problems were identified and resolved early and the resultant implementation derived from the design model was essentially fault-free; therefore, the normally costly testing process was event free, and significant schedule and cost savings were realized. If the test requirements could be reduced, automated, or outright eliminated, additional significant cost savings could be realized.

In addition to the cost savings, the formal modeling positioned the Rockwell Collins team to rapidly respond to requirements changes arriving late in the development process. Rather than updating the requirements, then updating the models and rerunning a large number of test cases, the team was able to update the specification model (that is, formalize and accept the requirements change), rerun the automated verification, identify needed design model changes, modify the design model to meet the new requirements, and automatically verify that the new design model met the changed specification model. This process based on formal verification was far more cost-effective than one based on testing of the design model.

4.3.2 Reducing Testing Efforts Through Formal Verification

The rigorous testing mandated by DO-178B/C is a major cost driver in critical airborne software development. The revisions in DO-178C and DO-333 allow for expanded use of formal techniques. One area in which formal techniques could be cost effectively deployed is in replacing or augmenting unit testing efforts. A French team has successfully pursued this direction at Dassault-Aviation and Airbus [93]. The use of formal verification in lieu of testing has been cost-effective in both organizations and significantly cut costs in maintenance activities for which costly modification of tests and re-tests has been replaced by re-verification. In this application of formal methods, the results are used for certification credit—the formal verification replaces testing activities previously required.

In this work, the source code for a software function or module is treated as the model *M*, and formalized low-level requirements are treated as the property *P*, to be verified using trusted automated means (as defined in DO-333). In this verification framework, the formalized low-level requirements are a formal contract for the software component. There is a need for assumption *A* to be able to complete the verification. For example, there may be assumptions related to the run-time environment in which the software will execute or assumptions on the ranges of parameters passed to the software module/function under consideration.

The verification used in lieu of testing (showing that the code conforms to the low-level requirements) is similar to the verification activities discussed in section 4.3.1, in which Rockwell Collins showed the design model conformed to the specification model. In the former case, however, because the verification is used to replace a required testing activity, the objectives of that testing activity must be met through the verification. Therefore, the activities related to the formal verification must demonstrate objectives, such as coverage (all requirements have been tested and the source code has been tested up to the appropriate coverage). When using verification, all possible behaviors of the source code are investigated (akin to exhaustive testing, but done without actually executing the program); there is no need to demonstrate

coverage of the code. Coverage of the requirements is determined by showing—through inspections and traceability links—that all low-level requirements pertaining to the software of interest have been correctly formalized as a contract. Through rigorous traceability and inspection processes, the objectives of DO-178C can be established around a kernel of formal verification of the source code (in this case, written in C).

In the published reports, the authors—including representatives from both Air Bus and Dassault—claim significant cost savings were realized [93]. It is clear that the cost savings do not materialize through the first iteration of applying the approach. Formalizing contracts; formalizing and validating assumptions; and identifying possibly dead code are relatively costly activities; there were no cost savings as compared to testing the first time using this technique by applying a software module. As the module evolved, however, there were significant cost savings in the regression verification (as opposed to regression testing) stages of the development because costly maintenance of large numbers of test cases could be eliminated; the cost of maintaining the module was significantly lowered with the use of formal verification.

4.3.3 Formal Methods Summary and Recommendations

From the literature search conducted and Safeware Engineering's experience, it is known that formal methods hold great potential to assist in the V&V of models and source code. As previously discussed, costly activities (such as rework, late requirements changes, test maintenance, and unit testing) can be helped or outright eliminated through the application of formal techniques. Nevertheless, formal methods are not considered to be a panacea, and there are potential pitfalls worth discussing.

4.3.3.1 Abstraction in Formalization.

As mentioned in the discussion of model-based development, modeling in general involves some level of abstraction. This is highly desirable because the models are created to better explain certain aspects of a system under development. Nevertheless, the explicit abstractions introduced in the modeling effort, implicit abstractions introduced in the choice of modeling notations, and additional assumptions (*A*) made when performing formal verification must all be documented, validated, and justified. Although DO-331 and DO-333 address the issues of abstraction and assumptions, Safeware Engineering's experience with formal verification indicates that this may become an area of concern.

4.3.3.1.1 Formal Methods Abstraction Recommendation

As mentioned as a recommendation in section 4.2.1, guidance material on the issues of abstraction and assumptions in the modeling and formal verification domain would be highly desirable.

4.3.3.2 Formalization of Specification Models

When performing formal verification, one attempts to determine if a property *P* is satisfied in model *M*. This, of course, necessitates the formalization of both *P* and *M*. In Safeware Engineering's experience, it is quite easy to provide models of *P* that may not capture what the

engineer thought was captured. As an example, consider the experience from the verification of the Rockwell Collins FGS model.

In this project, it became clear to the engineers formalizing the properties that great care needs to be taken in the formalization so that the proofs are meaningful. For example, the modeling of FGS macros (basically Boolean functions encapsulating complex decisions) was frequently undertaken to encapsulate commonly used properties (e.g., encapsulating a complex condition in a macro named "When_Lateral_Mode_Manually_Selected." Note that this macro appeared in the design model M).

These macros were frequently used when the "shall" requirements were formalized as properties *P* for the specification model. In this case, the properties were captured as CTL expressions. In many cases, the macro was used as the antecedent of an implication. For example:

AG (m_When_Lateral_Mode_Manually_Selected.result -> Onside_FD_On)

In English, it is always the case (AG) that when the condition When_Lateral_Mode_Manually_Selected is true, the Onside_FD will be on. This seemed like a sensible formalization of the "shall" requirement. The problem here is that the specification model *P* is using information (a modeling element) from the design model *M*. In this case, if the macro When_Lateral_Mode_Manually_Selected was over-constrained in the design model (or even contradictory, and therefore always false), this proof would succeed, but it would be rather meaningless. By relying on model elements in *M* when capturing the properties in *P*, it is easy to develop design and specification models that are seemingly sensible but that lead to meaningless verification results.

4.3.3.2.1 Formal Methods Model Specification Recommendation

Safeware Engineering recommends that guidance be developed to describe the best practices in modeling and to explain how to develop specification model and design models that do not interfere with each other in the verification process.

4.3.3.3 Loss of "Collateral Validation and Verification"

This last discussion on possible issues applies to both the model-based development and formal verification. In any development step for which activities that were previously manual are replaced with automation, the following is a potential issue.

Based on Safeware Engineering's observations related to test automation, it is worthwhile to include a short discussion on the possible repercussions of increased automation in the software and systems engineering processes. Although Safeware Engineering supports increased automation, there are possible pitfalls (see section 4.2.1.3) when manual processes are replaced by tools. For example, the kinds of tests generated by many test-generation tools would never have been derived by a human tester crafting tests based on years of domain expertise. Blindly replacing the latter with the former could have disastrous consequences (even if both test suites achieved MC/DC). Manual processes, be they for design, coding, testing, or flight testing critical software, draw on the collective experience and vigilance of many dedicated engineers

(primarily software engineers), engineers who provide collateral validation [68] as they are working on the software. Experienced engineers designing, developing code, or defining test cases provide informal validation of the software system; if there is a problem with the specified functionality of the system, they have a chance of noticing and taking corrective action. As engineers are replaced with automation, however, this validation process might be lost. For instance, as the code for a system is designed and developed, problems with the models and requirements may be exposed (the engineer doing the programming might have built a few of these systems in the past and, based on experience, may detect some problems). Had users relied on code generation from the models, this informal input from the engineer would be lost and possible undetected problems might make it through the development process. Similarly, as engineers craft test cases to satisfy MC/DC, careful thought goes into the process and problems in requirements, design, and code may be detected serendipitously. With test automation, this process will be lost.

With respect to the introduction of model-based development and formal verification, there are many opportunities for extensive automation. Manual processes—design, coding, or testing—draw on the collective experience and vigilance of many dedicated software professionals who provide collateral validation as they are working on the software. Experienced professionals designing, developing code, or defining test cases provide informal validation of the software system; if there is a problem with the specified functionality of the system, they have a chance of noticing and taking corrective action. As an example, consider the FGS requirements from the discussion on model-based development. Although the facts that the FGS had to be active and that no higher priority events were received at the same time as the HDG switch were not explicitly stated in the requirements, the engineers implemented the FGS functionality correctly; these problems were caught and corrected in the manual development process. When replacing these manual efforts with automation, proper validation of the formal artifacts on which the automation is based becomes absolutely essential; there may be no safeguards in the downstream development activities to catch critical flaws in the formal model (i.e., the collateral validation has been lost).

4.3.3.3.1 Formal Methods “Collateral Validation and Verification” Recommendation

The concerns related to the possible loss of collateral validation are based on the suspicion (and the studies related to the testing) that there have been no studies addressing the effect of increased automation. There is evidence to show what may be lost or gained when replacing manual processes with automation. Automating the inspection for the adherence to coding styles or verifying that a model adheres to various constraints, for example, can be highly effective and efficient. The aim with automation is to replace expensive and time-consuming manual activities with tools. Therefore, the automation does not have to be perfect; it simply needs to perform better than the manual processes it replaces. Nevertheless, there is reason to adopt automation with some caution before the full implications are known. Safeware Engineering believes more research in this area is warranted to determine how to certify software developed, analyzed, and tested with automated tools.

5. RESULTS AND FURTHER WORK

One goal of the research outlined in this report was to find alternative approaches to DO-178B and DO-278 to streamline the processes while maintaining quality and safety. Another goal was to find methods that could analyze COTS and legacy software. The initial work included an alternative methods assessment, an aviation industry poll, and a safety-critical industry standards comparison. The finding was that there were no obvious alternatives to DO-178B and DO-278 that could streamline the process. The authors recommended looking at technical advances that could still meet the goal of the study but were not necessarily alternatives to DO-178B and DO-278.

The phase 1 findings directed the team to look at techniques that could help users of the standards to streamline the process (and realize cost benefits) by ensuring the requirements are complete and correct early in the development process. Finding the appropriate safety requirements for the system under development and then allocating the appropriate safety requirements to the software components are of critical importance. A relatively new safety analysis technique known as STPA has the potential to make the safety analysis process more effective and efficient. A case study was performed on an FGS to demonstrate the effectiveness of STPA in providing requirements assurance. To reduce the costs associated with software development—in particular, the very high costs associated with the verification tasks—model-based development and formal verification were assessed. Both sets of techniques have been recognized as promising, and guidance is available in DO-331 and DO-333. This study looked at cost savings that were realized by Rockwell Collins and Airbus using model-based development and formal verification. A discussion on the pitfalls of using model-based development and formal verification was also provided.

The following paragraphs detail the findings of each technical advancement:

STPA–The STPA analysis of the FGS highlighted several strengths. First, analysis can include a variety of controllers to the system, including hardware, software, and human elements. In addition, the analysis of the interactions of the FGS with the FMS demonstrates how STPA can be used to handle new software interacting with legacy software or COTS. STPA can be used early in the development to flesh out requirements of these components. Last, STPA is scalable. The analysis can be done on a component, subsystem, or system of systems, as was the case for the Non-Advocate Safety Assessment of the Ballistic Missile Defense System. In addition, the analysis was compared to a traditional hazard analysis (FTA) performed on the same FGS. The comparison demonstrated that the STPA found three times more issues with the requirements than the FTA; it also provided more specific recommendations to mitigate those issues. One drawback of STPA is that the tools to automate this analysis are still in development. Typically, analysts use Microsoft® Visio® to build the control structure diagram and a word processing program, like Microsoft Word®, to capture the analysis.

Model-based development–Model-based development centers the development efforts on models of the desired software behavior (or some aspects of the desired behavior). These models help clarify the required behavior; can serve as a basis for testing and test generation; enable code generation; and, if formal, can be subjected to various types of analysis. Model-based development promises increased productivity and increased quality. There are few published

data to support these claims, but the anecdotal evidence available indicates that model-based development appropriately applied can provide significant savings. This report covered the findings and described successful uses of model-based development in the civil aviation domain.

Formal verification refers to the usage of mathematical techniques to reason about various aspects of models or source code. Such reasoning can be far more comprehensive than traditional testing and can foster significant cost savings. Again, there are few available data on the cost savings of using formal methods in the development process. Nevertheless, there are credible reports of significant advantages of using these techniques in the development and certification of critical avionics systems. This report reviewed formal verification efforts and described two approaches to the use of formal verification in practice:

1. An approach for which verification was used to improve the requirements elicitation and modeling efforts.
2. Another approach for which formal verification was used in lieu of costly unit testing.

Future work–Because the usage of new safety analysis techniques, model-based development, and formal verification are not yet in widespread adoption, there are opportunities to misuse the techniques. The report provided a brief discussion on possible pitfalls and provided recommendations for future efforts that may assist in the effective adoption of such techniques to help the industry reduce the development costs of critical software without affecting quality and safety.

In conclusion, the authors of this report recommend the following future work:

- Team up with aviation partners to use STPA on a project currently in development. Assess results of the STPA analysis to determine applicability in ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.
- Accelerate adoption of model-based techniques and reduce the chances of bad outcomes (and possible delay or outright rejection of promising techniques). Notation-agnostic guidance documents on modeling methodology should be developed.
- Model-based development enables automation. Recent advances in automated test generation hold great promise. Unfortunately, generated tests may not be as effective as manually developed tests. A study as to which test generation approaches provide effective tests and which requirements, model, and code test adequacy coverage criteria are suitable for test generation needs to be conducted.
- Formal verification holds great promise, but there are numerous ways the formal verification efforts can provide misleading or outright erroneous results (e.g., through the misuse of abstractions or assumptions in the verification process). These issues are inherent in formal verification (independent of tools and modeling notations), and generic guidance as to how to effectively use formal verification techniques should be developed.
- The introduction of automation to replace manual processes may reduce the “collateral validation and verification” that occur as skilled engineers address various tasks. The negative effects, if any, of replacing humans with automation are not well-known and should be investigated further.

6. REFERENCES

1. Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1997, Vol. 19.

2. Boehm, B., Software Engineering Economics, Englewood Cliffs, NJ: Prentice-Hall, 1981.

3. Davis, A., Software Requirements: Object, Function, and States, Englewood Cliffs, NJ: Prentice-Hall, 1993.

4. Ramamoorthy C., Prakesh, A., Tsai, W., and Usuda, Y., "Software Engineering: Problems and Perspectives," *IEEE Computer*, October 1984, pp. 191–209.

5. van Schouwen, A., "The A-7 Requirements Model: Re-Examination for Real-Time Systems and an Application to Monitoring Systems," Technical Report 90–276, Hamilton, Ontario: Queens University, 1990.

6. Leveson, N., Safeware Engineering: System Safety and Computer, Reading, Massachusetts: Addison-Wesley Publishing Company, 1995.

7. Lutz, R., "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," in *IEEE Symposium on Requirements Engineering*, San Diego, 1993.

8. Parnas, D. and Madey, J., "Functional Documentation for Computer Systems Engineering" (Volume 2). Technical Report CRL 237, Hamilton, Ontario: McMaster University, September 1991.

9. Heitmeyer, C., Labaw, B., and Kiskis, D., "Consistency Checking of SCR-Style Requirements Specifications," in *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

10. Heitmeyer, C., Kirby, J., and Labaw, B., "Automated Consistency Checking of Requirements Specification," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, July 1996, Vol. 5, No. 3, pp. 231–261.

11. Faulk, S., Brackett, J., Ward, P., and Kirby, J., "The CoRE Method for Real-Time Requirements," *IEEE Software*, September 1992, Vol. 9, No. 5, pp. 22–33.

12. Faulk, S., Finneran, L., Kirby, J., Shah, S., and Sutton, J., "Experience Applying the CoRE Method to the Lockheed C-130J Software Requirements," in *Ninth Annual Conference on Computer Assurance*, Gaithersburg, MD, June 1994.

13. Rushby, J., Keynote address, NASA Formal Methods Workshop, 2013.

14. Rushby, J., “New Challenges in Certification for Aircraft Software,” in *Proceedings of the Ninth ACM International Conference on Embedded Software* (EMSOFT 2011), October 2011, Taipei, Taiwan.

15. RTCA, DO-178B, “Software Considerations in Airborne Systems and Equipment Certification,” RTCA, 1992.

16. Software for Dependable Systems: Sufficient Evidence? Jackson, D., Thomas, M., and Millett, L., eds., Washington: The National Academies Press, 2007.

17. Billings, C., Aviation Automation: The Search for a Human-Centered Approach, CRC Press, 1996.

18. Weiner, E. and Curry, R., “Flight-Deck Automation: Promises and Problems,” Ergonomics, May 19840, Vol. 23, No. 10, pp. 170–177.

19. Zimmerman, M., Leveson N., and Lundqvist, K., “An Investigation of the Readability of Safety-Based Specification Languages,” in *2002 International Conference on Software Engineering*, Orlando, FL, May 2002.

20. De Millo, R., Lipton, R., and Perlis, A., “Social Processes and Proofs of Theorems and Programs,” *Communications of the ACM*, May 1979, Vol. 22, No. 5, pp. 271–280.

21. Brooks, F., “No Silver Bullet—Essence and Accidents of Software Engineering,” *IEEE Computer*, April 1987, Vol. 20, No. 4, pp. 10–19.

22. RTCA, DO-278, “Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance,” RTCA, May, 2002.

23. IEEE Standard 12207-2008 – Systems and Software Engineering, Software Life Cycle Processes, IEEE Computer Society, 2008.

24. IEC, 62304:2006, “Medical Device Software—Software Life Cycle Processes,” International Electrotechnical Commission, Geneva, 2006.

25. Dupuy, A., Safety-Critical Software Testing in Airborne Systems, MIT Master’s Thesis, Aeronautics and Astronautics Department, May 1999.

26. Leveson, N., Cha, S., and Shimeall, T., “The Use of Self-Checks and Voting in Software Error Detection: An Empirical Study,” *IEEE Transactions on Software Engineering*, September 1983, Vol. SE-9, No. 5, pp. 569–579.

27. Zimmerman, M., Investigating the Readability of Formal Specification Languages, Master’s Thesis, MIT, May 2001.

28. Dulac, N., Viguier, T., Leveson, N., and Storey, M.A., "On the Use of Visualization I Formal Requirements Specification," in *International Conference on Requirements Engineering*, Germany, September 2002.

29. Gerhart, S. and Yelowitz, L., "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on Software Engineering*, May 1976, Vol. SE-2, pp. 195–207.

30. Leveson, N., "The Role of Software in Spacecraft Accidents," *AIAA Journal of Spacecraft and Rockets*, July 2004, Vol. 41, No. 4.

31. Kitchenham, B., "Empirical Studies of Assumptions That Underlie Software Cost-Estimation Models," *Journal of Information and Software Technology*, April 1992, Vol. 34, No. 4, pp. 211–218.

32. Shen, V.Y., Conte, S.D., and Dunsmore, H.E., "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support," *Computer Science Technical Report*, 1981, pp. 81–376.

33. Fenichel, R., "Heads I Win, Tails You Lose," *ACM Computing Surveys*, 1979, Vol. 11, No. 3, p. 277.

34. Lassez, J., van der Knijff, D., and Shepherd, J., "A Critical Exampling of Software Science," *Journal of Systems and Software*, December 1981, Vol. 2, No. 2.

35. Malange, J.P., "Critique de la Physique du Logiciel (Critique of Software Science)," Technical Report IMN-)-23, University of Nice, France, October 1980.

36. Moranda, P.B., "Is Software Science Hard?" *ACM Computing Surveys*, December 1978, Vol. 10, No. 4, pp. 503–504.

37. Stevens, W., Myers, G., and Constantine, L.L., "Structured Design," *IBM Systems Journal*, 1974, Vol. 13, No. 2, pp. 115–139.

38. McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, December 1976, Vol. SE-2, No. 4, pp. 308–320.

39. Yourdan, E. and Constantine, L.L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

40. Eckhardt, D., NASA Langley Research Center, personal communication.

41. Romeu, J., "A Discussion of Software Reliability Modeling Problems," *Journal of the Reliability Analysis Center*, 2000, pp. 1–5.

42. Certification Authorities Software Team (CAST), *CAST-18 Position Paper: Reverse Engineering in Certification Projects*, rev. 1, June 2003.

43. “Reverse Engineering” Software and Digital Systems, FAA report DOT/FAA/TC-15/27, February 2016.

44. Leveson, N.G., “Intent Specifications: An Approach to Building Human-Centered Specifications,” *IEEE Transactions on Software Engineering*, January 2000, Vol. 26, No. 1, p. 17.

45. Leveson, N.G., “The Role of Software in Recent Aerospace Accidents,” http://sunnyday.mit.edu/accidents/issc01.pdf (accessed on 03/10/15).

46. Leveson, N., “A New Accident Model for Engineering Safer Systems,” *Safety Science*, April 2004, Vol. 42, No. 4.

47. Leveson, N., “Comparison of STPA to FTA TCAS II: Model-Based Analysis of Socio-Technical Risk,” Technical Report, Engineering Systems Division, Massachusetts Institute of Technology, June 2002.

48. Ishimatsu, T., Leveson, N., Thomas, J., Katahira, M., Miyamoto, Y., and Nakao, H., “Modeling and Hazard Analysis Using STPA,” Presented at the *Conference of the International Association for the Advancement of Space Safety*, Huntsville, Alabama, May 19–21, 2010.

49. RTCA DO-312, “Safety, Performance and Interoperability Requirements Document for the In-Trail Procedure in the Oceanic Airspace (ATSA-ITP) Application,” Washington: RTCA, June 19, 2008.

50. Fleming, C.H., Spencer, M., Thomas, J., Leveson, N., and Wilkinson, C., “Safety Assurance in Nextgen And Complex Transportation Systems,” *Safety Science*, in press.

51. Sundaram, P. and Hartfelder, D., “Compatibility of STPA With GM System Safety Engineering Process,” in *2013 STAMP Workshop Presentations, MIT Partnership for a Systems Approach to Safety (PSAS)*, Boston, Massachusetts, 2013.

52. Leveson, N., Wilkinson, C., Fleming, C., Thomas, J., and Tracy, I., “A Comparison of STPA and the ARP 4761 Safety Assessment Process,” MIT PSAS Technical Report, Boston, October 2014.

53. Software Engineering Methodology, Appendix C, Conducting Structured Walkthroughs, DOE G 200.1-1, 1997.

54. Laitenberger, O. and DeBaud, J.M., “An Encompassing Life-Cycle Centric Survey of Software Inspection,” IESE-Report No. 065.98/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, October 1998.

55. Yeates, D. and Wakefield, T., *Systems Analysis and Design*, 2nd edition, Pearson Education Limited, 2004, pp. 100–101.

56. Shull, F., Russ, I., and Basili, V., "How Perspective-Based Reading Can Improve Requirements Inspections," *IEEE Computer*, July 2000, Vol. 33, No. 7, pp. 73–79.

57. "Structured Analysis Wiki, Appendix D," http://yourdon.com/strucanalysis/wiki/index.php?title=Appendix_D (accessed on 03/10/15).

58. Zhang, Z., Basili, V., and Shneiderman, B., "Perspective-Based Usability Inspection: An Empirical Validation of Efficacy," *Empirical Software Engineering: An International Journal*, March, 1999, Vol. 4, No. 1, pp. 43–69.

59. Maldonado, J., Carver, J., Shull, F., et al., "Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness," *Empirical Software Engineering: An International Journal*, March 2006, Vol. 11, No. 1.

60. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., and Rierson, L.K., "A Practical Tutorial on Modified Condition/Decision Coverage," NASA/TM-2001-120876, May 2001.

61. Dupuy, A. and Leveson, N., "An Empirical Evaluation of the MC/DC Coverage Criterion on the Hete-2 Satellite Software," in *Proceedings of the Digital Aviation Systems Conference* (DASC), Philadelphia, Pennsylvania, October 2000.

62. Vilkomir, S. and Bowen, J., "From MC/DC to RC/DC: Formalization and Analysis of Control-Flow Testing Criteria," Formal Aspects of Computing, pp. 42–62, 2006.

63. Whalen, M., Heimdahl, M., Rajan, A., and Staats, M., "On MC/DC and Implementation Structure: An Empirical Study," in *Proceedings of the 27th Digital Avionics Systems Conference (DASC '08)*, IEEE, St. Paul, Minnesota, October 2008.

64. Whalen, M., Gay, G., You, D., Staats, M., and Heimdahl, M., "Observable Modified Condition/Decision Coverage," in *Proceedings of the 35th International Conference on Software Engineering*, San Francisco, California, May 2013.

65. Heimdahl, M., Devaraj, G., and Weber, R., "Specification Test Coverage Adequacy Criteria-Specification Test Generation Inadequacy Criteria?" in *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

66. Staats, M., Gay, G., Whalen, M.W., and Heimdahl, M.P., "On the Danger of Coverage Directed Test Case Generation," in *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Tallinn, Estonia, April 2012.

67. Chilenski, J.J. and Miller, S.P., "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, September 1994, pp. 193–200.

68. Heimdahl, M.P., “Safety and Software Intensive Systems: Challenges Old and New,” in *Proceedings of the International Conference on Software Engineering Track on the Future of Software Engineering*, 2007.

69. Miller, S.P., Tribble, A.C., Carlson, T.M., and Danielson, E.J., “Flight Guidance System Requirements Specification, NASA/CR-2003-212426,” Rockwell Collins, Cedar Rapids, Iowa, 2003.

70. Tribble, A.C., Miller, S.P., and Lempia, D.L., “Software Safety Analysis of a Flight Guidance System,” NASA/CR-2004-213004, March 2004.

71. France, R. and Rumpe, B., “Model-Driven Development Of Complex Systems: A Research Roadmap,” *Future of Software Engineering 2007*, Briand, L. and Wolf, A., eds., IEEE-CS Press, 2007.

72. MathWorks, MathWorks corporate website, www.mathworks.com (accessed on 03/11/15).

73. MathWorks, Simulink product website, www.mathworks.com (accessed on 03/11/15).

74. Esterel Technologies, Corporate website,” www.esterel-technologies.com (accessed on 03/11/15).

75. Esterel-Technologies, SCADE Suite product description, http://www.esterel-technologies.com/v2/scadeSuite-ForSafetyCriticalSoftwareDevelopment/index.html (accessed on 03/11/15).

76. Harel, D.H., Lachover, A., Naamad, A., et al., “Statemate: A Working Environment for The Development of Complex Reactive Systems,” *IEEE Transactions on Software Engineering*, April 1990, Vol. 16, No. 4, pp. 403–414.

77. RTCA, “DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A,” December 13, 2011.

78. Leveson, N., Heimdahl, M., Hildreth, H., and Reese, J., “Requirements Specifications for Process-Control Systems,” *IEEE Transactions on Software Engineering*, September 1994, Vol. 20, No. 9, pp. 684–707.

79. Bouali, A. and Dion, B., “Formal Verification for Model-Based Development,” in *Proceedings of the 2005 SAE World Congress*, Detroit, Michigan, April 11–14, 2005.

80. Beine, M., Otterbach, R., and Jungmann, M., “Development of Safety-Critical Software Using Automatic Code Generation,” in *2004 SAE World Congress*, Michigan, March 8–11, 2004.

81. Marcil, L., “MBD, OOT and Code Generation: A Cost-Effective Way to Speed up HMI Certification,” in *Proceedings of the SAE Aerospace Electronics and Avionics Systems Conference*, Mesa, Arizona, 2012.

82. SAE International, “ARP4754 A Guidelines for Development of Civil Aircraft and Systems,” December 21, 2010.

83. Heimdahl, M. and Leveson, N., “Completeness and Consistency in Hierarchical State-Based Requirements,” *IEEE Transactions on Software Engineering*, June 1996, Vol. SE-22, No. 6, pp. 363–377.

84. Thompson, J., Whalen, M., and Heimdahl, M., “Requirements Capture and Evaluation in Nimbus: The Light Control System,” *Journal of Universal Computer Science*, July 2000, Vol. 6, No. 7, pp. 731–757.

85. Heitmeyer, C., Jeffords, R., and Labaw, B., “Automated Consistency Checking of Requirements Specifications,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, July 1996, Vol. 5, No. 3, pp. 231–261.

86. Miller, S.P., Tribble, A.C., Whalen, M.W., and Heimdahl, M.P., “Proving the Shalls: Early Validation of Requirements Through Formal Methods,” *Journal on Software Tools for Technology Transfer (STTT)*, February 2006.

87. Miller, S.P., Whalen, M.W., and Cofer, D.D. “Software Model Checking Takes Off,” *Communications of the ACM*, February 2010, Vol. 53, No. 2, pp. 58–64.

88. Wyk, E.V. and Heimdahl, M.P., “Flexibility in Modeling Languages and Tools: A Call to Arms,” in *Proceedings of the IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Columbia, Maryland, September 2005.

89. Zimmerman, M.K., Lundqvist, K., and Leveson, N., “Investigating the Readability of State-Based Formal Requirements Specification Languages,” in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

90. Leveson, N., Reese, J., and Heimdahl, M., “SpecTRM: A CAD System for Digital Automation,” in *Proceedings of the 17th Digital Avionics Systems Conference*, November 1998.

91. RTCA, “DO-333 Formal Methods Supplement to DO-178C and DO-278A,” December 13, 2011.

92. Cofer, D., Whalen, M.W., and Miller, S.P., “Software Model Checking for Avionics Systems,” in *Proceedings of the 27th Digital Avionics Systems Conference (DASC ’08)*, 2008.

93. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., and Monate, B., “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience,” *IEEE Software*, May–June 2013, Vol. 30, No. 3.

APPENDIX A—POLL ON ALTERNATIVE APPROACHES TO SOFTWARE ASSURANCE

In phase 1 of this research, an online poll was conducted to ask participants how they apply software assurance in their particular industries. Questions were written to gain insight into the participants' experiences; job titles and duties; the software assurance methods they use; standards they would recommend; software assurance methods they use and their assessment of those methods; their experiences with implementation of cyber security; and their opinions regarding the benefits of third party certification. The poll was open for one month and was closed on March 17, 2012. The poll responses that were compiled follow.

FAA ALTERNATIVE APPROACHES TO SOFTWARE ASSURANCE POLL RESULTS

(Compiled on March 20, 2012 by Safeware Engineering Corporation)

Question 1: What Is Your Job Title?

Answers:

- Executive Engineer
- Certified Software Architect
- FAA Aerospace Engineer
- Hardware Design Team Leader
- Staff Engineer
- Software Engineering Team Leader
- FAA DER/Certification Manager
- Senior Engineer
- Software Quality Engineer
- Certification Project Engineer Mgr (I am also an FAA DER with Level A SW & AEH delegation)
- Aerospace SW Engineer
- Process Engineering
- Senior Software Engineer
- Embedded Software Engineer
- Electrical DER and ODA Unit Member
- Electrical DER and ODA Unit Member
- President and Designated Engineering Representative
- CNS Systems Engineer for the USAF working for Jacobs Technology, Inc
- Software Technical Manager
- VP Advanced Development
- Sr. Systems Engineer
- Senior Consultant - Safety Critical Software Development
- Chief Technical Officer, DER, TCR (TSO Compliance Representative)

- OBAR (Outside Boeing Authorized Representative)
- Principal Systems Engineer
- Principal Engineer
- Software Safety Engineer
- Sr. Manager
- Airworthiness Engineer FAA DER (Software and Engines)
- Technical Manager Software, Product Development Quality Engineer
- Sr. Engineering Specialist
- FAA DER
- Technical Manager
- CTO
- Computer Software Engineer
- Director, Systems & Software
- Technical Fellow - Military Certification
- SW Developer
- Systems Software Engineer
- Consulting DER
- Project Administrator & SW & CEH UM of an ODA
- Sr. Design Quality Engineer
- Principal Engineer
- Software Safety/Certification Engineer
- Engineering Manager
- Flight Controls Safety and Cert Engineer
- CEO

Question 2:

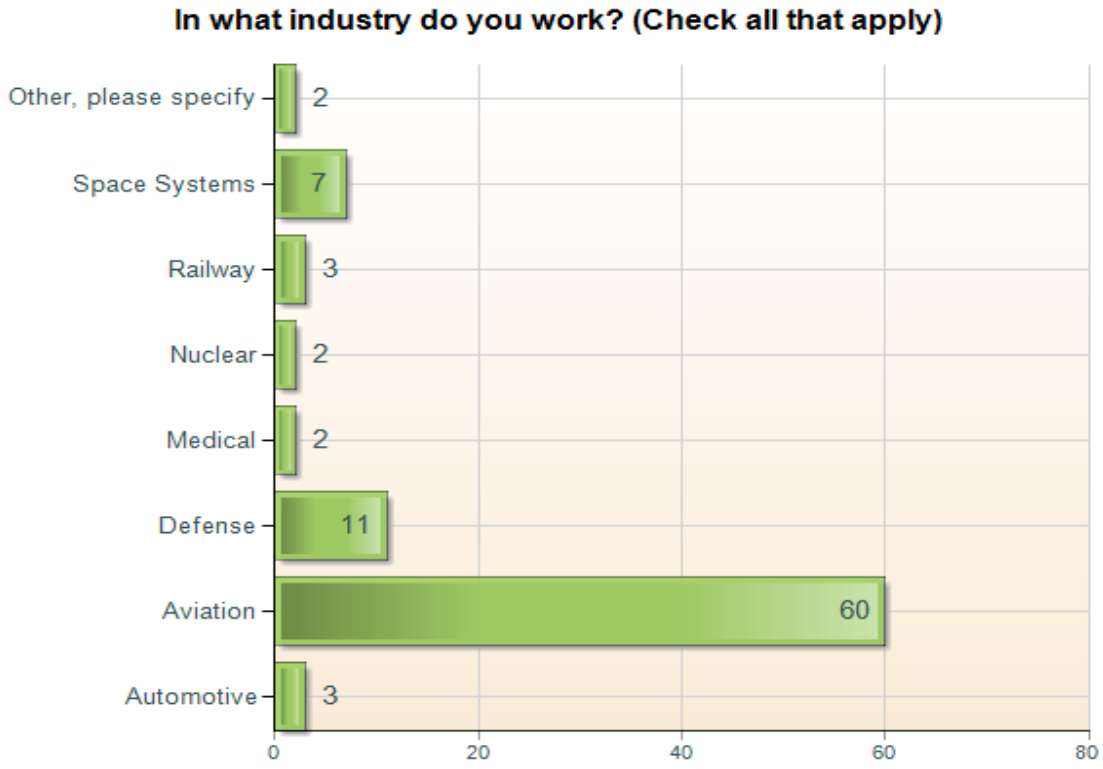


Figure A-1. Participants' industry

Responses for "Other":

- Power generation - green enterprises
- Industrial (process industry)

Question 3:

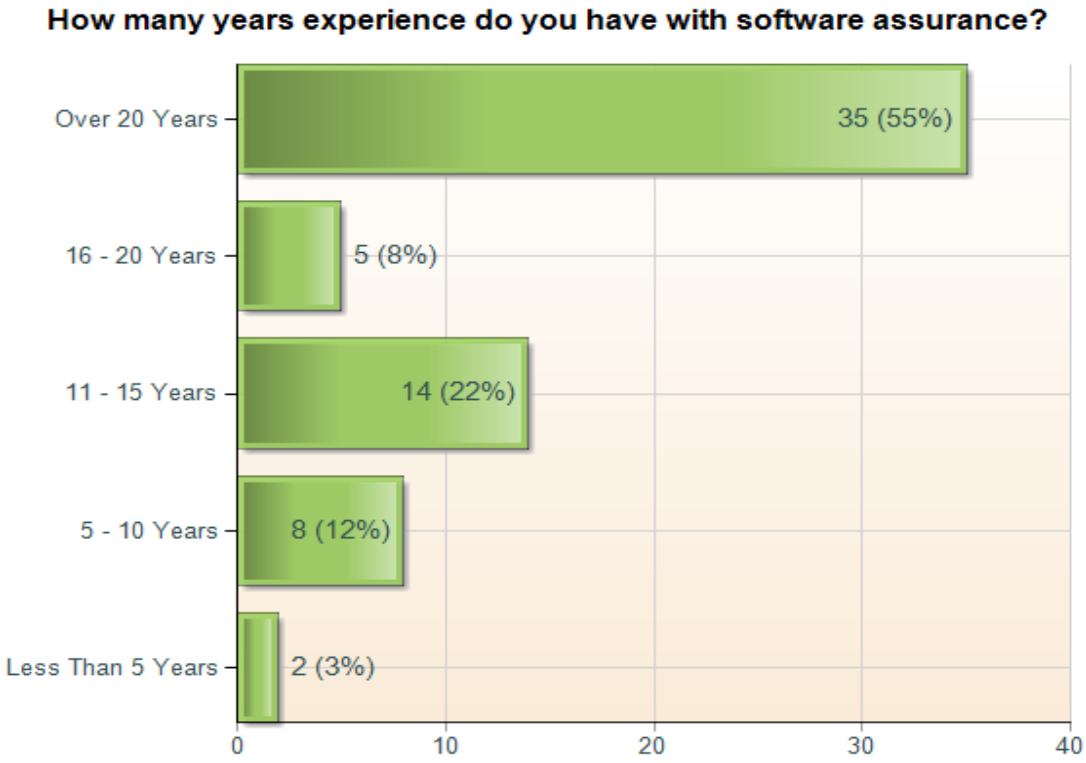


Figure A-2. Years of experience

Question 4:

How are your job duties related to your company's software assurance process? (Check all that apply.)

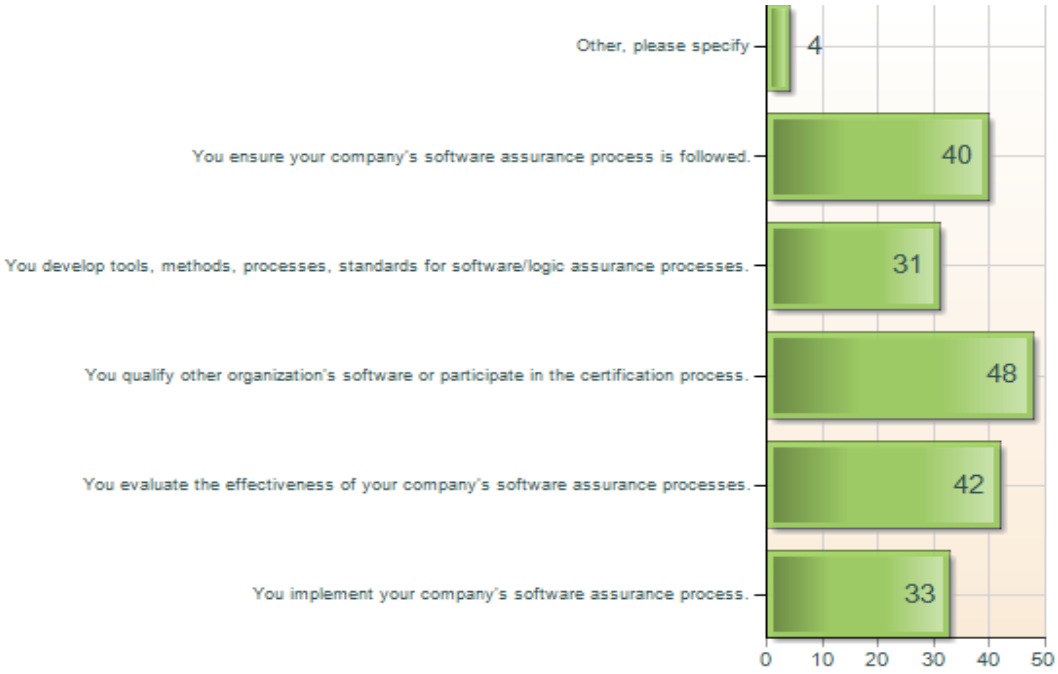


Figure A-3. Relationship between job duties and software assurance process

Responses for "Other":

- Manage Certified Software Development Groups.
- Approve/Recommend software lifecycle data artifacts on behalf of the FAA.
- Responsible for software aspects of certification.
- Develop new process and policy.

Question 5:

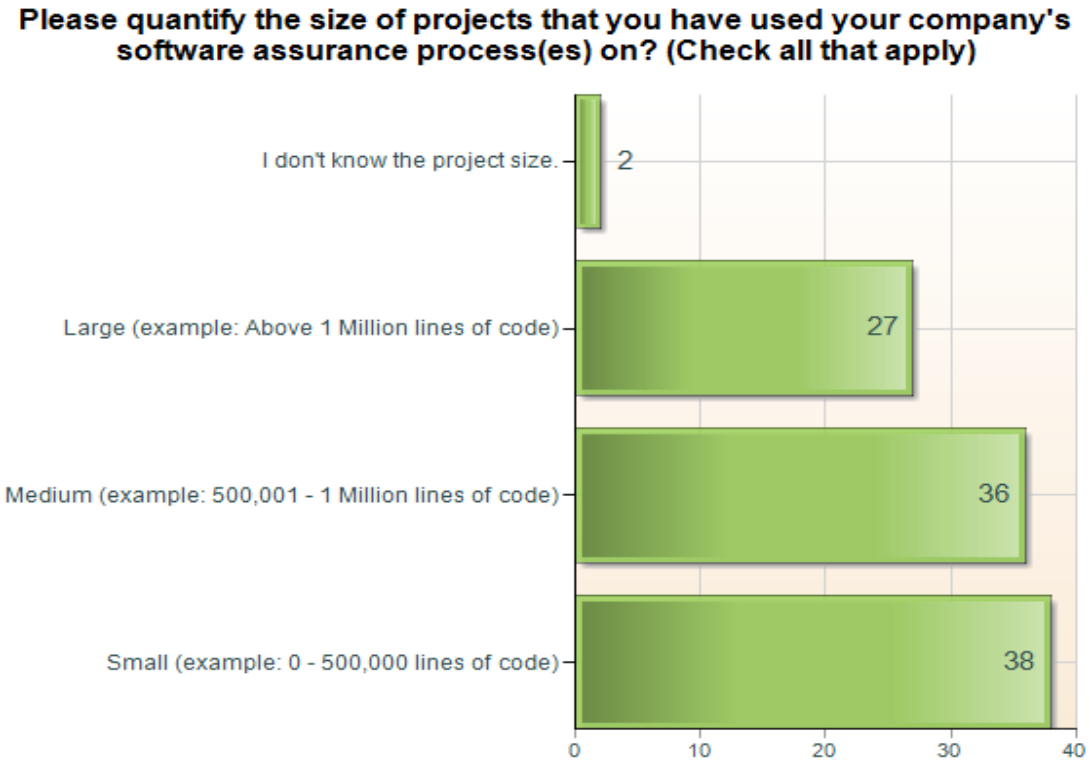


Figure A-4. Size of projects

Question 6:

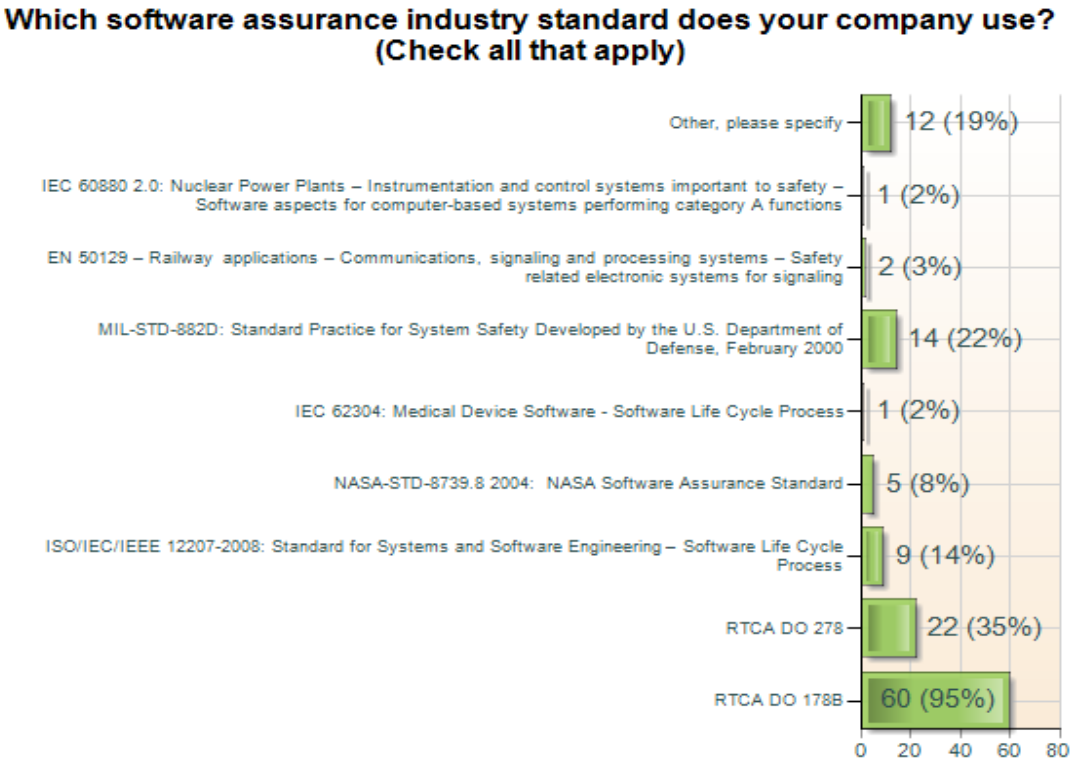


Figure A-5. Software assurance standards used

Responses for "Other":

- MIL HDBK 516b: Department of Defense Handbook: Airworthiness Certification Criteria
- Also use standards required under contract with customer
- MISRA
- AMCOM Regulation 385-17: US Army Aviation Missile Command Software System Safety Policy
- AC's and TSO's
- AS91000: Aerospace Policies Procedures Manual
- AS9115 if required by contract: Quality Management Systems-Requirements for Aviation, Space and Defense Organizations
- IEC-61508: Functional safety of electrical/electronic/programmable electronic safety-related systems
- ISO-26262: Road Vehicles – Functional safety – Part 1
- Capability Maturity Model Integration (CMMI)
- IEC 61508

- CMMI PPQA
- DO-200A: Standards for Processing Aeronautical Data
- AS9100B/C-AS9006

Question 7:

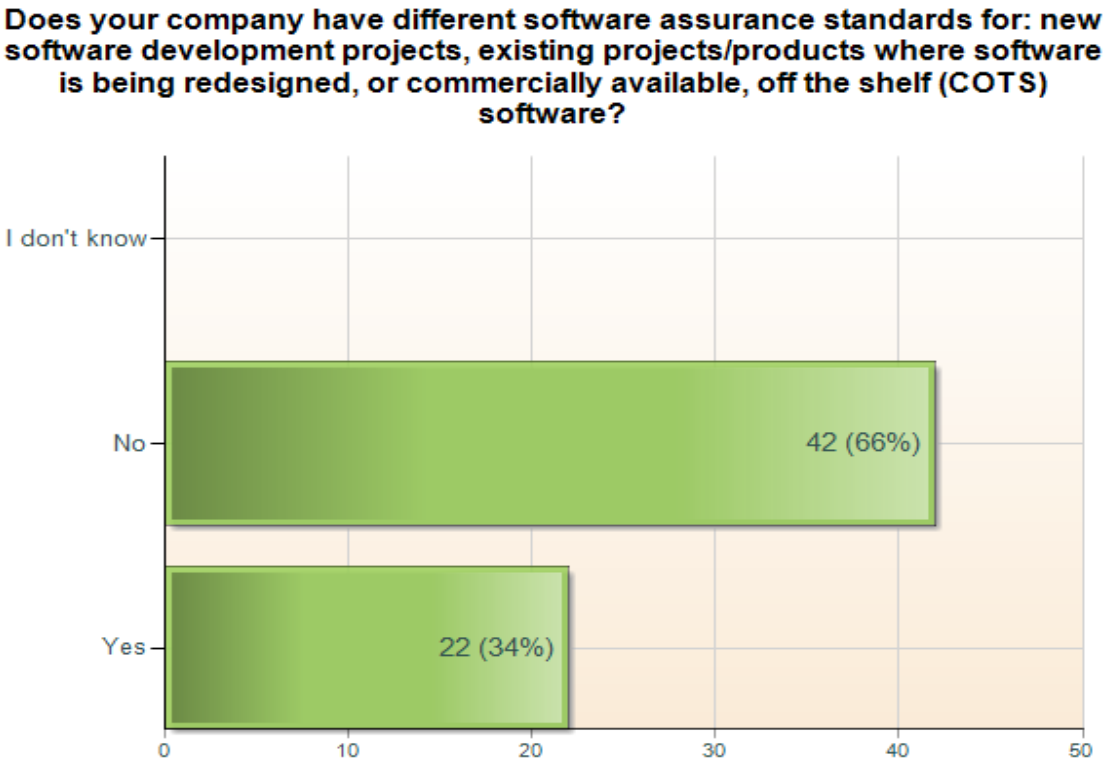


Figure A-6. Different software assurance processes for new vs. existing projects

Question 8:

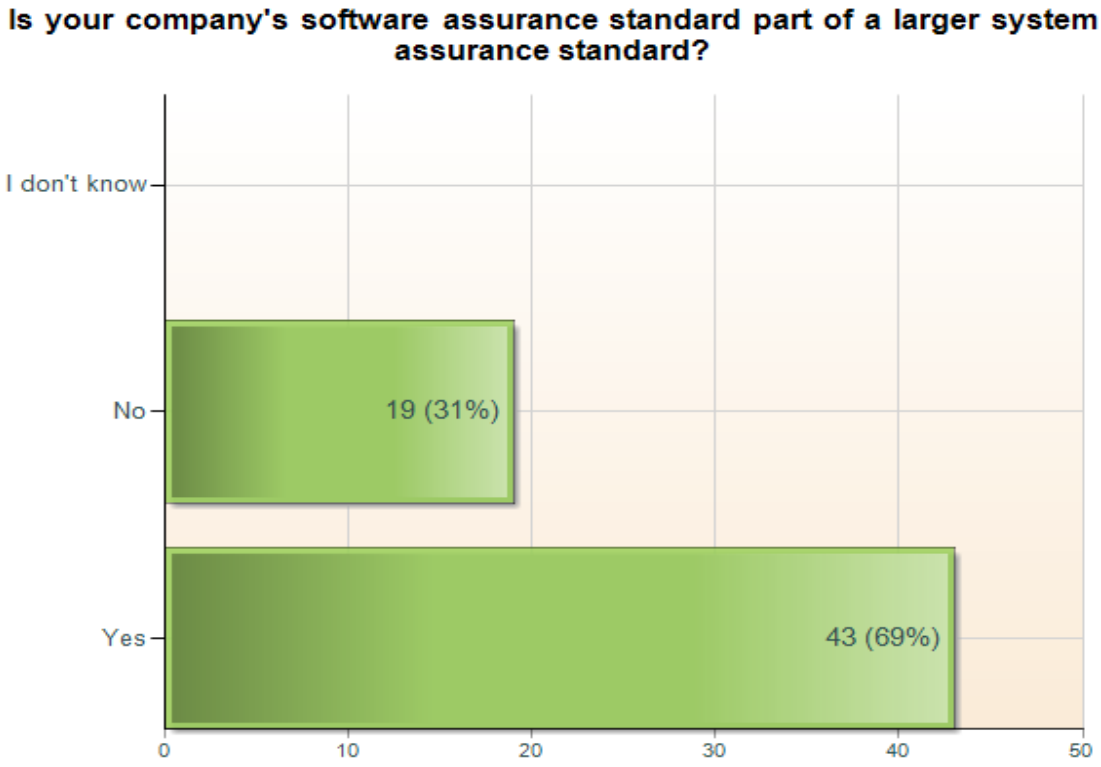


Figure A-7. Software assurance standard part of system assurance standard

16 Comments:

1. The nuclear software is part of our Nuclear Quality Assurance Manual, which complies with ASME NQA-1:2008/2009.
2. The level of system assurance varies on a project-by-project basis.
3. ARP-4754A (Certification Considerations for Highly-Integrated Or Complex Aircraft Systems) and DO-297 (Integrated Modular Avionics Development Guidance and Certification Considerations).
4. Aircraft performance qualification.
5. Internal company standards for product assurance drive the use of software assurance standards.
6. The system process calls out the software assurance standards.
7. Software Safety works closely with the System Safety function to satisfy our customer needs and expectations.
8. Software process is part of the system process that incorporates elements of SAE ARP4754.

9. Software assurance standard is designed at the sub-system functions (smallest unit) in which are the configuration items that build the element(s). These element(s) are subsequently made up the segment(s) level (highest level).
10. FAA engine certification.
11. The DOD/MIL standards used by the defense sector of the company differ from the commercial DO-178B.
12. About 30% of my clients have a commercial/military standard which they try to tailor into a DO-178B compliant one. The rest are dedicated to a DO-178B environment, hopefully with a single standard with DAL Level dependencies.
13. SAE ARP-4754.
14. Company System Assurance Standard, based on ARP 4754.
15. Somewhat, but currently mainly using DO-178B, overall.
16. Corporate Quality Management System, per AS9100B and 14CFR TSO regulations.

Question 9:

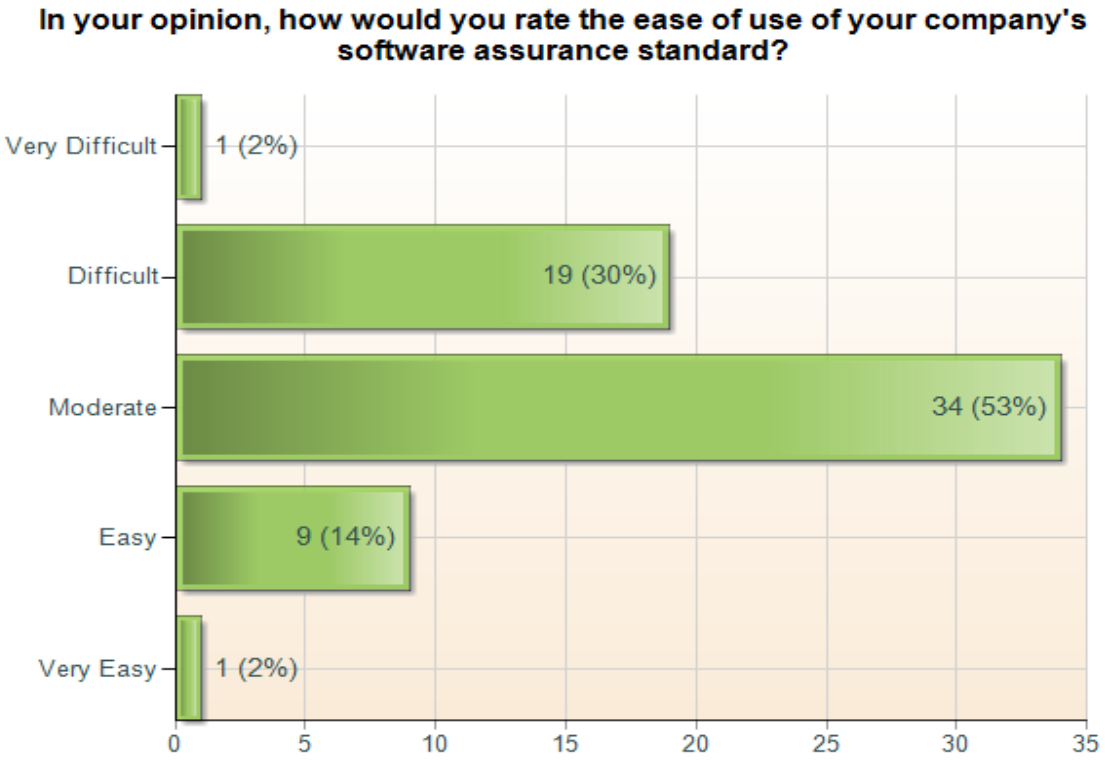


Figure A-8. Ease of use of your company's software assurance standard

Question 10:

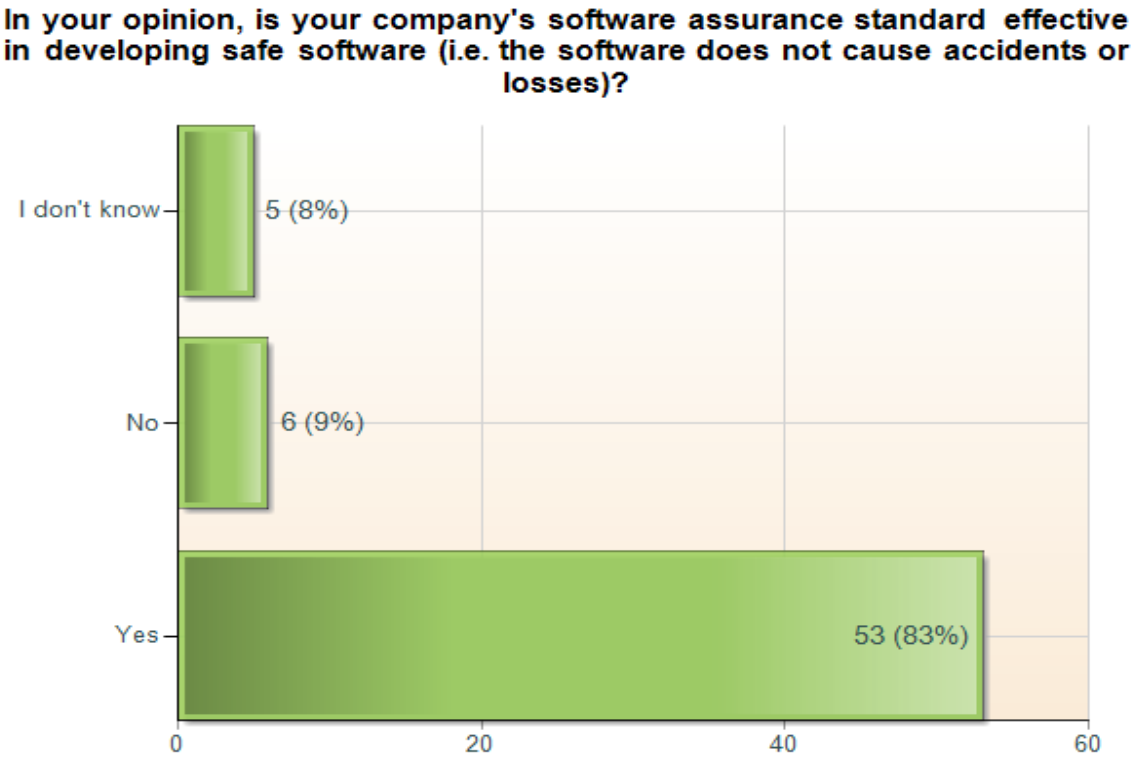


Figure A-9. Effectiveness of software assurance standard

Question 11: What, if any, other software assurance standards have you used? Please describe their effectiveness, ease of use, and limitations.

25 Comments:

1. IEC 60880 Ed 2 (Nuclear power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions) - no easier to use than the IEEE standards as modified by the Nuclear Regulatory Commission.
2. No other standards, but have used additional techniques, such as digital signatures of both objective evidence and software loads.
3. The FAA needs to require the following for Flight Critical Software: Mandatory use of "Abstract Data Types" require high resolution, time-stamped vehicle wide software warning and message logs. Use of Safe programming Languages - No C, No Visual Programming Languages used to auto generate code.
4. DOD-2167/A (Defense Systems Software Development superseded by MIL-STD-498).
5. None.
6. 2167A / 2168 - Moderate to difficult 498 – Moderate.
7. So much is now developed out of house; we have a lot of defined processes for insuring that they too follow our processes.
8. Person with lots experience using right tools.
9. None.
10. DoD-STD-2167. Overly prescriptive and not focused on development of safe software.
11. We use various standards, e.g., modeling and coding standards, to augment DO-178B and 278
12. DOD-STD-2167A, MIL-STD-498, Defense Systems Software Development: Somewhat similar to DO-178B in onerousness.
13. Safety Directed Development - used while at Honeywell in the 90's as an alternative to DO-178B. A limitation is that it requires very knowledgeable, experienced systems engineers to analyze the software, but it more directly addresses safety than DO-178B/C. Although developed by Honeywell, they have released it publicly.
14. We assess software development based on tailoring of the MIL-STD in many cases. Close and through examination is applied to the software to effect consistency with 178B (e.g., similar to AC 20-171).
15. 1. Why doesn't this survey allow me to go to previous pages? 2. Why does this survey refer to "software assurance standards" when the questions are really being asked of the development & integral processes overall? 3. My company uses the Personal Software Processes and the Team Software Process and delivers zero defects.
16. DO178.
17. We also use internal standards that are tailored for the intended software criticality.
18. IEC61508, Easier to comply with but less acceptance bodies with expertise.

19. DO-178A and individual project specific standards. DO-178A was adequate for ensuring safe software; however, it was augmented by our own company standard. The project-specific standards were not used to ensure a safe product; safety wasn't the primary issue for these products. The primary purpose of the homegrown standards was to promote a quality product.
20. None.
21. Already discussed – effect, ease of use, limits are all application environment dependent.
22. None.
23. RTCA/DO-178, RTCA/DO-178A: requires too much interpretation to be effective. DOD-STD-1679 not effective. DOD-STD-2167A not effective. MIL-STD-498 non-effective.
24. STANAG 4404 (Software IAW NATO Standardization Agreement) - difficult for some people to understand and apply to current software development.
25. Other standards may have been used prior to DO178B, but we have been primarily DO178 since its release.

Question 12:

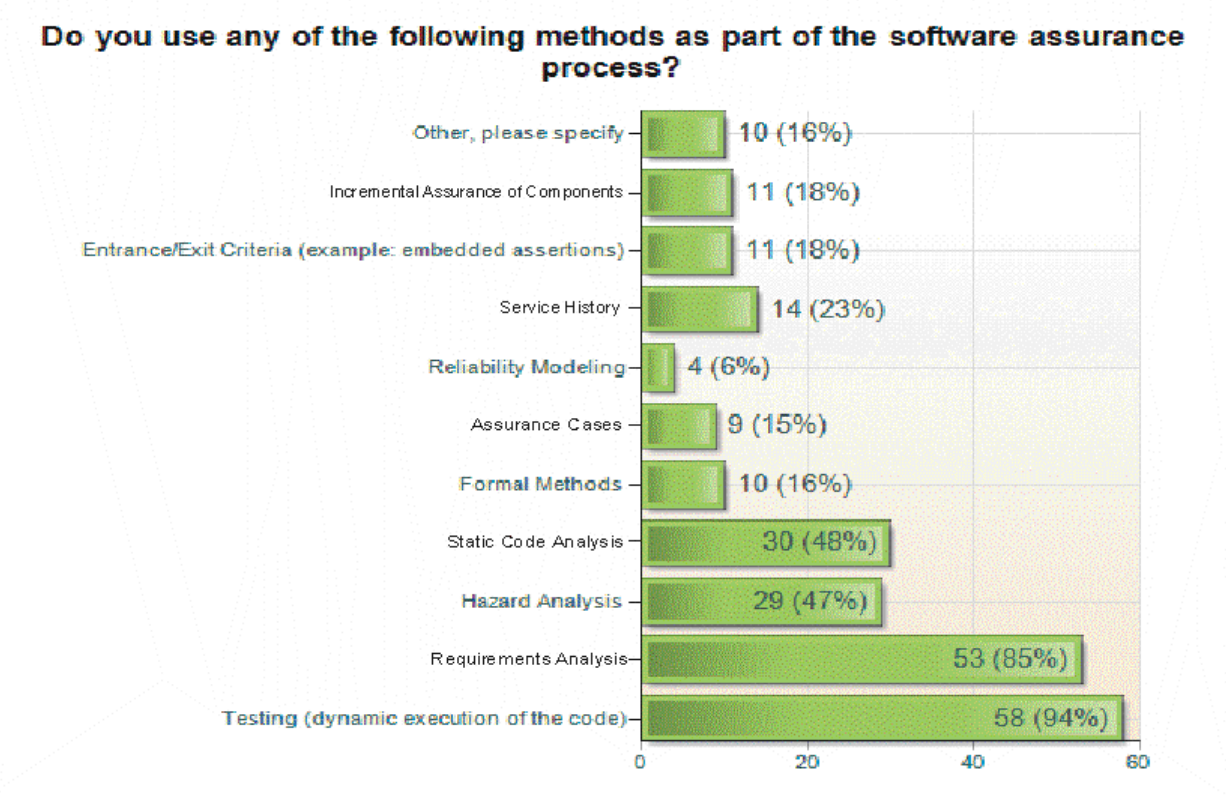


Figure A-10. Methods used for software assurance

10 Responses to "Other":

1. Need Algorithms reviews; Critical Race section reviews.
2. You're kidding, right? Only allowing 3? Zillions of others.
3. Peer reviews.
4. Process auditing.
5. Qualified development and verification tools.
6. FAA SOI audits.
7. Review of all applicable SW documentation for a project.
8. Simulation.
9. Life cycle entry/Exit criteria audits.
10. Formal reviews process.

Question 13:

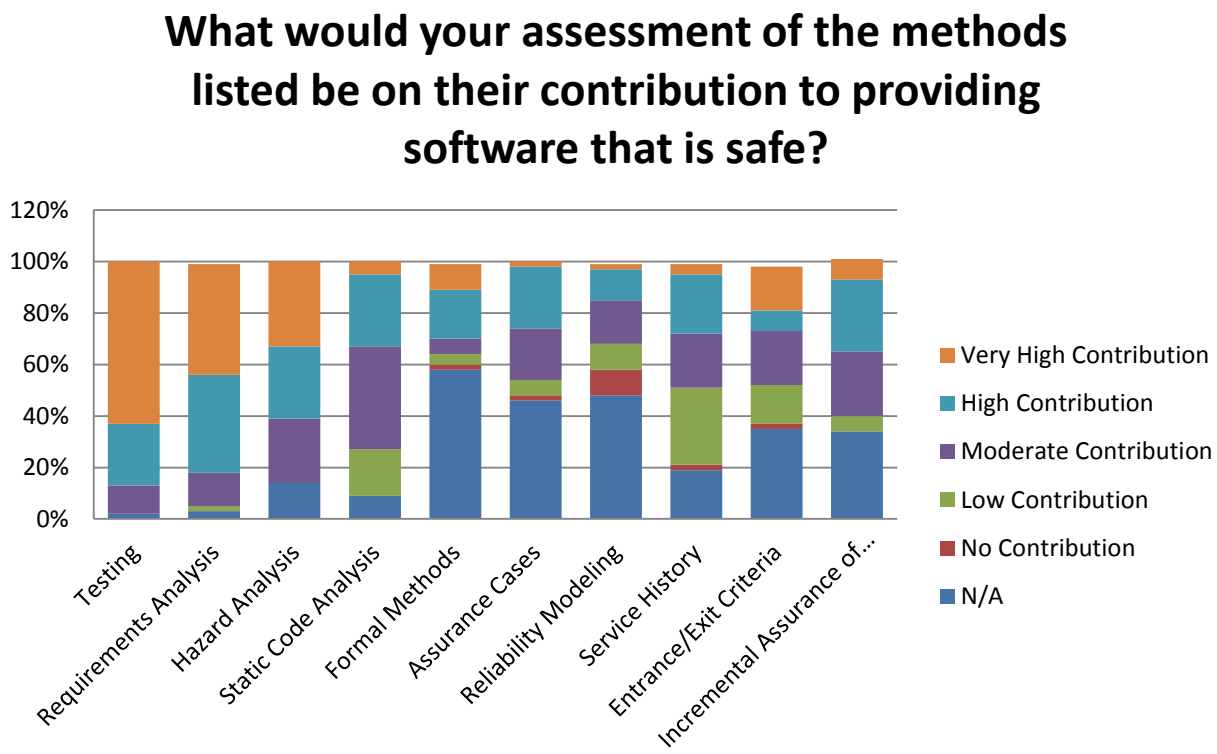


Figure A-11. Various assurance methods' contribution to safe software

Question 14:

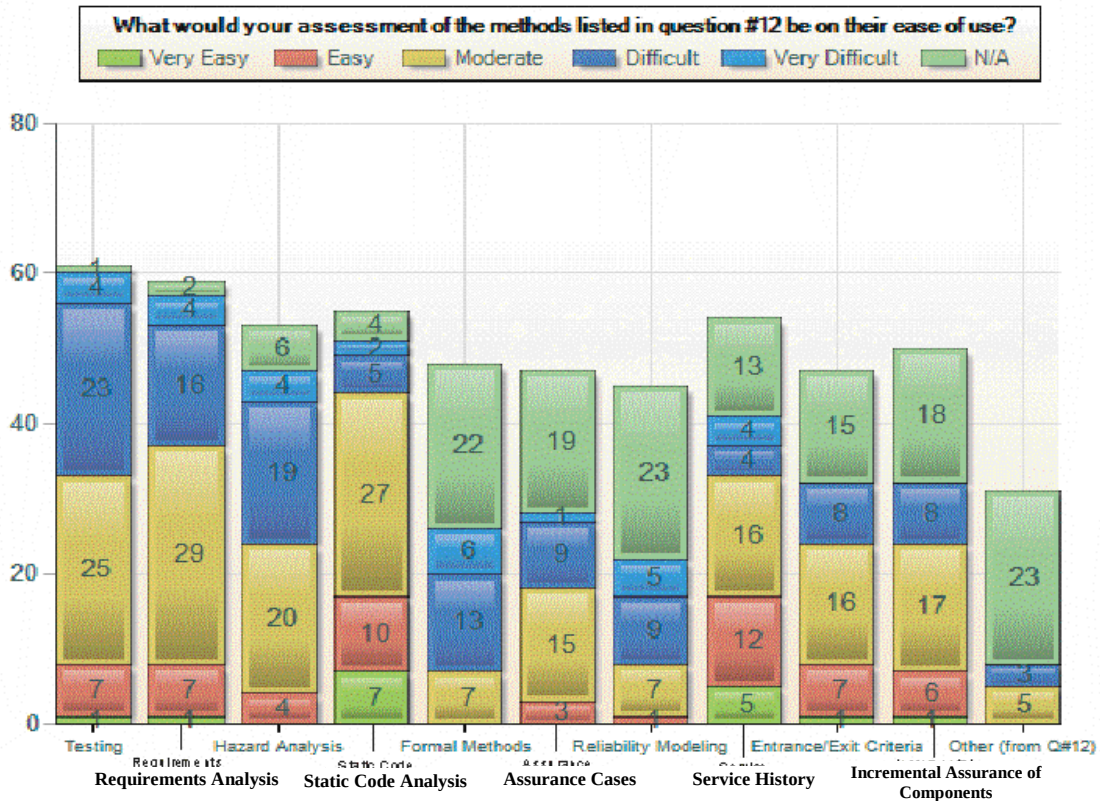


Figure A-12. Various assurance methods' ease of use

Question 15: Are there additional methods you would include in the software assurance process? Please list below.

25 Comments:

1. Code review.
2. System level requirements reviews. Requirements checklists prioritize requirements based on safety criticality eliminated software TSOs—These are dangerous.
3. If we could insure good requirements we would do it. No solution known at this time.
4. MC/DC testing 100%.
5. Exhaustive testing.
6. Structural Coverage Analysis, Data and Control Coupling Analysis.
7. Tribal knowledge of what to look for.
8. Include testability in the design requirements.
9. No.
10. IVV—independent organizations evaluating correctness and completion of artifacts and approaches.
11. None.
12. In my experience, the very best way to provide assurance is to assure that the developer's process has followed the RTCA/DO-178C guidelines. This is without question and proven. Any other process will be evaluated against those guidelines.
13. Requirements-based structural coverage testing, especially MC/DC testing, adds virtually no value for the expense incurred to conduct it.
14. CMMI.
15. Peer reviews—of requirements before design before code.
16. No.
17. Process on handling suppliers Requirements Verification Traceability Matrix. Process on Critical Detail Design and Test Readiness Review entry/exit criteria.
18. Configuration Management.
19. Assurance of system level operation and safety response during low-level testing.
20. Develop certified airborne software database of errors which have caused safety issues, and remedies.
21. Cohesive teaming with safety, system and software, testing personnel. Adopt formal model-based system engineering that provides consistent and well-defined interfaces between tools used to capture/analyze/simulate requirements, design to implementation of the software.
22. Thinking the problem from another perspective. I find more real issues in a requirement/design/code (or even test case when it's done) review where the designer "teaches" the material to one or more competent evaluators. Changing the perspective allows the designer to see his own lapses of logic or flow.
23. Coverage Analysis, as a measure of the effectiveness of Requirements and testing.
24. None.
25. NA.

Question 16: Do you have any recommendations for improving or streamlining your company's software assurance process? Please describe below.

1. Since I wrote the nuclear process, no.
2. Eliminated Software TSOs—These are dangerous Require Abstract Data Types for safety-critical code. Require FAA/NTSB vehicle wide software error/warning logging of messages with high-resolution timestamps on a universal clock. Require the use of safe programming languages. Require software certification for engineers writing safety critical code. Require engineers attain a PE. Require software engineers have a computer science bachelor's degree from an accredited university.
3. Train people in software engineering and test methodologies. Most U.S.-trained engineers are totally unprepared for safety critical systems/software development and testing. It will often take 5+ years to get an engineer to the point that they are truly useful in developing, verifying, and validating safety critical systems because there is very little formal training provided in universities or within the industry.
4. The processes are fine as long as properly trained and qualified people are following them.
5. Need to focus the testing on high-level requirements. I have not seen that much gained from low-level testing and structural coverage, compared to high-level testing.
6. Focus more on Design Assurance instead of Process Assurance.
7. Having common processes and tools across the various products would ensure consistency and provide some streamlining. Unfortunately, this is not practical or possible because of legacy vs. new or different platforms.
8. My recommendation to all company. Please don't do shortcut on testing.
9. The #1 issue is the quality of requirements that go into the software process. The industry in general has demonstrated an ability to faithfully translate the requirements they are given into an executable piece of software. The industry continues to increase requirements on the process of translating requirements into software, while ignoring the predominant problem. Pounding on a single nail will not keep the roof down, especially when that nail is already secure. I would suggest it is better to focus research on system definition, modeling, analysis, and validation.
10. Greater emphasis on formal methods. We are in a catch-22 in that my company would like to see regulators begin to embrace FM and I believe the regulators will begin to see FM as a valuable tool in safety assurance if companies like mine begin using them.
11. Improved configuration management, training for software quality assurance personnel, and more corporate emphasis on software assurance.
12. Automation to the maximum extent possible.
13. Yes develop a reusable framework for I/O because of the little idiosyncrasies that are discovered in the hardware.
14. There are many "good practices" that contribute significantly (and sometimes even more) to the overall quality of the SW.

15. No.
16. DO-178B is very academic in nature. It's like the authors wanted to include everything ever taught in computer science classes. Being required by cert authorities to follow every word in DO-178B does not allow for streamlining.
17. RESEARCH.
18. None.
19. None.
20. Much more effort needs to be devoted to systems requirements analysis.
21. More focus on operational scenarios and system testing and less emphasis on trivial defects.
22. Generally, better tools for gathering all the pieces for review.
23. More concise/specific and less wording checklists.
24. Peer Reviewing process to properly disposition the action items and waiver/deviations.
25. Project management configuration/management Design guidelines Process, SWQA Acceptance/Certification.
26. Focus on high-level testing with low-level coverage analysis.
27. Better and consistent integration of tools, methods, and practices across programs. Without this things are not standard or necessarily known.
28. Prevent cutting corners on assurance process.
29. Similar to Q15.
30. My two cents is that we are shooting at a moving target. Thirty years ago we WANTED patches at the object code level because we didn't trust the assembler/compilers. Now very few coding errors propagate to the final product. But design errors, especially errors of concept or algorithm, are the most common problem. And they are the most difficult to fix. The complexity is really challenging us at the system level and it has long become next to impossible to get a human mind around a major software build. A friend of mine actually went mentally disabled while trying to understand the redundancy management of the quadraplex Space Shuttle Orbiter. We fixed it by adding a fifth system, single string, which could be irrevocably switched into place by the crew (a real switch labeled “Backup”). Its existence was solely to protect against the quadraplex redundancy management which no one trusted.
31. Automation of the information-gathering and traceability process. The administrative components should not raise the costs of doing real certification work.
32. None at this time.
33. Spend more time analyzing requirements traceability to Functional Failure Paths and architectural design elements that reduce negative effects to the system. Keep requirements at a higher level and don't include low-level design items that increase cost and not safety.
34. More focus on software verification, less focus on software conformity.
35. No.

36. The company strives to improve the overall software-assurance process, but cert-authorities are continuously adding process requirements, for example, updating of FAA Order 8110.49, and migrating to DO-178C.

Question 17:

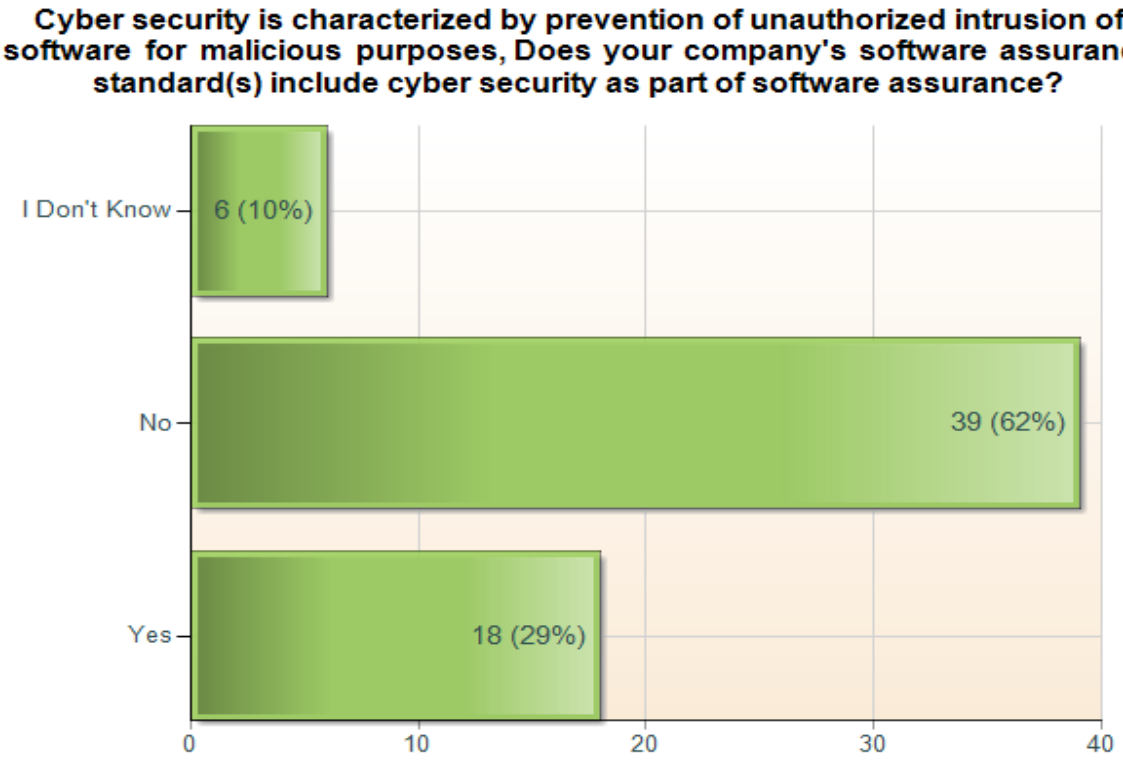


Figure A-13. Cyber security

Question 18: If you answered “Yes” to question #17, what methods do you use to determine if the software is secure?

20 Comments:

1. As outlined in Nuclear Regulatory Commission Regulatory Guide 5.71.
2. Static analysis, testing, secure configuration management, digital signatures.
3. Systems analysis, requirements analysis, design analysis, vulnerability analysis, testing.
4. I'm not sure this is a type of cyber security, but any corruption of the embedded airborne software would be detected by a program memory CRC (or Checksum) check.
5. Using CM to do a JOB.
6. Again, cyber security focused solely on software is misplaced. We work hard via architectural means to ensure the software is not the first line of defense. Ask how we make the system secure, not how we keep the software secure.
7. ISO 27001.
8. CRC is a safe method of assuring no intrusion.
9. We don't look at it as a software assurance problem so much as a system assurance problem.
10. Requirements Analysis, Testing.
11. Obscurity—Thus far, embedded controller software has been implemented using unique languages and/or limit external protocol interfaces with unique fault-detection requirements (e.g., ARINC 429 data bus).
12. Controlled access to source files, controlled build processes, controlled load processes in the manufacturing process, power-up runtime checks embedded in the software.
13. Final software is checked before release for viruses.
14. Formal methods.
15. Software development process: software part ions, software filtering implementation, unloadable software/executables.
16. CRC checks, hardware pin selectable configurations, design disallowing user modification, use of permissions for limited access.
17. I defer to another expert.
18. I am teaching the need for it, but most are hiding in the belief that their systems are “isolated.” That isolation is rapidly disappearing. This is a major stretch of the assurance requirements in a new and counter direction to classical system/software safety. Isolation could be effective for sensors and front-ends, but somewhere the fully “network” integrated cockpit is going to have to deal with lots of threats ... especially as fly-by-wire spreads.
19. CRC checks, bit parity checks, intrusion detection, watch dog timers, information assurance requirements, authentication, firewalls.
20. Develop use cases that cover security issues and include those in the overall system analysis.

Question 19:

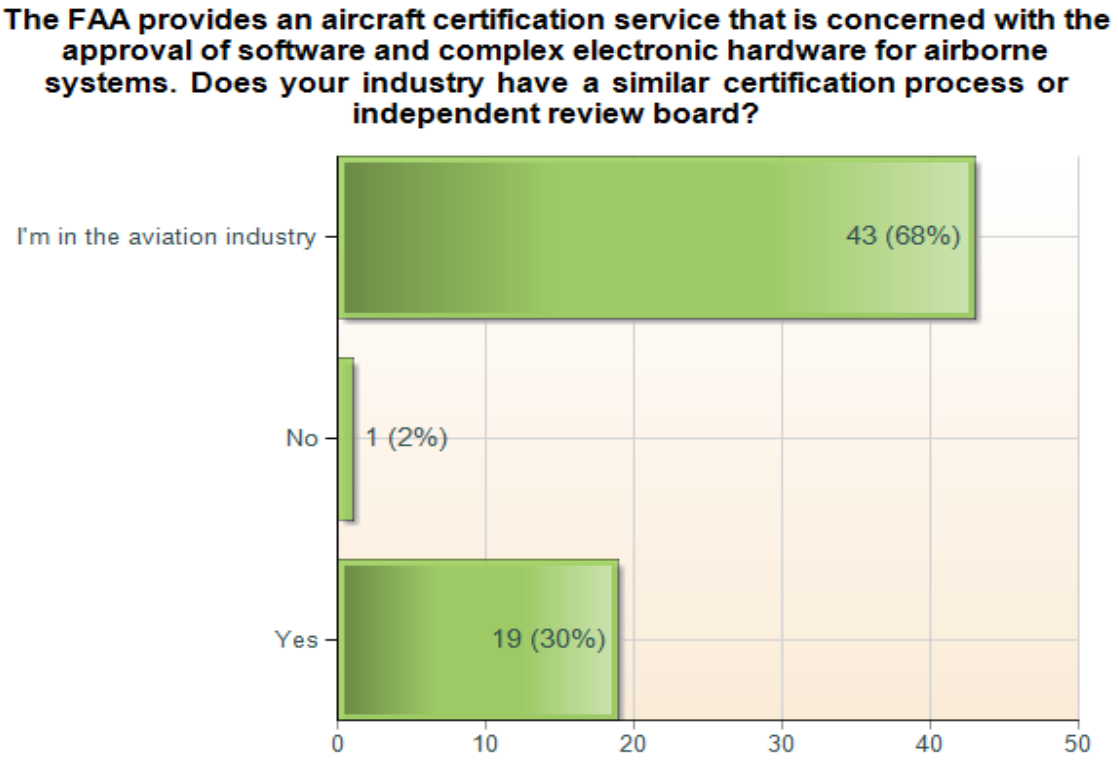


Figure A-14. Certification process other than FAA

Question 20:

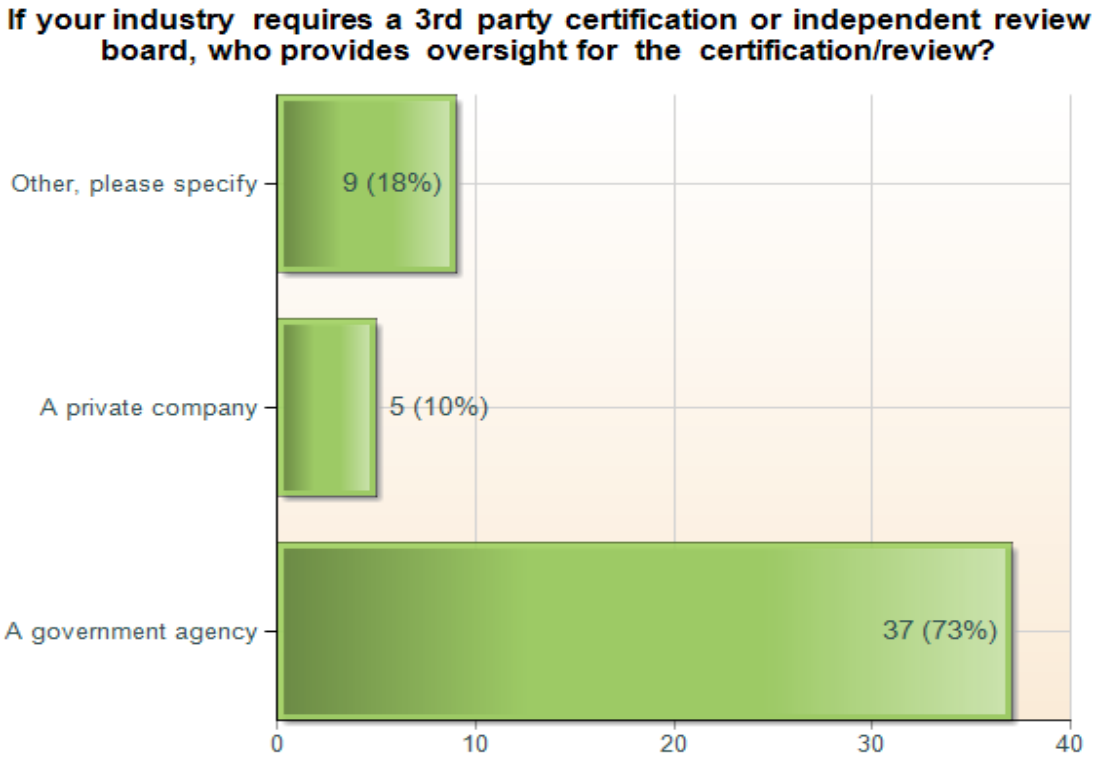


Figure A-15. Oversight of independent certification process

Response to "Other":

1. We do the oversight.
2. Not required, could not "uncheck" this box.
3. Not required.
4. Independent engineers as designated within company.
5. FFA, EASA, internal or external DER's.
6. I don't know same as questions 19.
7. Independent DER.
8. Company certification Administrated Authorized Representative.
9. We are an ODA under FAA.

Question 21: In your opinion, does having an independent 3rd party certification of the system or independent review board ensure the software is safe (i.e., the software does not cause accidents or losses)? Please explain your reasoning.

51 Responses:

1. If the third party does a poor job of assessment or is solely concerned with the article passing their review, then there is no value in the third party review. If the third party does a sufficiently bad job, there may be negative value in their review.
2. Not necessarily. Based on the 3rd party, the contribution to safe software can vary significantly. It is very much dependent on the individual certification authority.
3. No. The third party audits cause a lot of work in documentation preferences, but rarely result in a change to the software design itself. The software engineers and company representatives know how to design safe software and internal audits would be sufficient.
4. It keeps folks honest, though often the 3rd party can be rather clueless.
5. Yes. Through the use of DERs, fairly thorough review of our data helps ensure the process was followed and therefore that the product is safe.
6. Not particularly, if all the review is to ensure the process is followed.
7. Yes. Helps ensure that internal standards are documented and followed. Lack of this oversight would likely result in loss of lessons learned.
8. Generally the 3rd party gets over-concerned with following process vs. technical issues.
9. It reduces the risk.
10. An independent 3rd party review board only provides value or helps to ensure safe products if the members of the board are technically knowledgeable in SW/AEH and the actual system being certified. I have experienced numerous situations where the 3rd party team is focused on non-technical process aspects or pure documentation issues. Although I understand process compliance is important, fixating on less critical aspects, instead of the technical items, actually distracts development teams and could be a cause for less robust development and verification. In my humble opinion, if the 3rd party independent board member(s) have not actually performed SW/AEH development or verification on airborne SW programs (for several years), they should not be allowed to evaluate it.
11. Do complete SOI audit for their SW.
12. In general I believe a third party; specifically the FAA, is a good idea. I think the industry would be less safe left on our own; not by intent, but more because of unchecked consumption of our own bath water. Some regulatory oversight is needed. How that is executed can be debated in more words than fits in this box. I do appreciate the many forces that govern the rules and execution of them. It is a tough job.
13. It could. Depends on how the review is conducted.
14. Not so much. 3rd party paper reviews are merely following checklist steps without the insight or expertise of a coherent design team composed of flight crews, systems designers, AND software subject matter experts.
15. It does, in the sense that companies will follow the process only if they know they will or might get audited by DERs, FAA, or other certification authorities.
16. No comment at this time.
17. No.

18. No. It is driven by a culture that drives people to understand that what I am doing has the potential to do harm, and we must avoid it for the benefit of the end user. In fact, we ourselves may be the end user.
19. No. If you have an adequate internal process, then this should be much more beneficial and more focused on content rather than process.
20. Not necessarily, teams adapt their software safety practices to please the 3rd party representative's desires and not use their technical knowledge to produce safer software.
21. Yes, as it is independent with great guidelines and objectives.
22. Yes. The need to provide the evidence that the product is safe as well as performing its intended function means that work essential to ensuring safety needs to be done. Without that oversight, that work would not get done for cost reasons.
23. No, because the reviewers nitpick rather than provide useful improvements. The software seldom changes as a result of cert authority participation. They mostly impact the documentation in what are often perceived as worthless ways.
24. NO, it's not perfect, but it does put an independent set of eyes on the artifacts that may point out some overlooked parts or inappropriate assumptions in the development.
25. Yes. Thorough independent review.
26. No. It ensures that the process was followed, which is what ensures the software is safe.
27. No. Their understanding and evaluation are limited by what is presented to them.
28. No. Our processes ensure that. Audits just verify what we have done.
29. Yes, the government agency requirements act as a minimum standard against which compliance can be measured. Lack of an external standard leads to process drift over time, or management pressure to reduce process time/cost through process changes.
30. In my opinion, a 3rd Party Independent Review Board provides another perspective for validating the safe operation of software. One that is not biased by company or agency directives and can give an objective opinion.
31. Independent organizations in my experience don't have the technical expertise to really monitor that the technical work is being done in a safe manner. Their main contribution should be to ensure that the organization producing the software has integrity. I am doubtful this can really be done except with very regular interaction, more than just the prescribed reviews and audits.
32. No. It helps. Our third party review is to AS9100, and most auditors are not familiar with SW. If they were, I would say yes.
33. It helps, but doesn't ensure the software is safe.
34. No. The Product Development Quality Assurance department is independent, although internal, and it guarantees objectiveness.
35. It would be considered affective for an independent certification team/individual to be part of the review board, but the team/individual should to be technical and engaged closely with the design team to ensure the software is being developed and conformed to spec/orders/regulations.
36. To certify that all measures have taken place before use in a safety environment.
37. Independence makes you think harder and be more critical about “assurance” vs. “having a good feeling.”
38. Yes, it ensures a relatively consistent and rigorous practice. Without a 3rd party, expedient decisions are common.
39. 3rd party certifications are not reliable.

40. An independent DER ensures maximum objectivity.
41. Ideally, the independent certification organization has expertise on safety aspects, whereas the system folks are focused on system design.
42. Yes. 1) By prepping for review, you take that "other perspective" view of your own work and see the "oops" or "I wonder if that is an oops." 2) Review flushes out misunderstood or non-functional assurance processes. 3) If best practice teaching is part of the review, company can benefit from consolidated experience of the 3rd party. 4) It is amazing to me how repeatable many of the process errors are...mostly human failures, which are to be expected, but which must be found.
43. Software is neither safe nor unsafe. Systems are the place to look for safety. Software can contribute to hazards, but it must be evaluated in the context of a system and function.
44. It prevents the bar from being lowered to an unacceptable level.
45. It is an extra set of eyes on the project that can provide a safer product. However the size of the project and hazard assessment may be used to determine involvement.
46. Maybe. It all depends on the approach of the 3rd party certification. Those looking for requirements traceability and verification miss the mark. Experience of know issues pays tenfold when knowing what to look for.
47. I do not think that an independent 3rd party certification delivers what it is intended to deliver. In situations I am familiar with, the designers and architects went through significant effort but accomplished minimal improvements to the software-driven platform. Customer perception of safety was in several cases reduced after certification driven changes.
48. I would say it is a significant part, but not all of the assurance process.
49. No. Independent people in the developing organization who understand the process and the products will do the best job.
50. Not applicable.
51. The 3rd party oversight ensures that the software development included due diligence in all aspects of the development: e.g., software testers help review software requirements and code; developers help review requirements and test cases; and that the right decisions are made with respect to software safety.

Question 22:

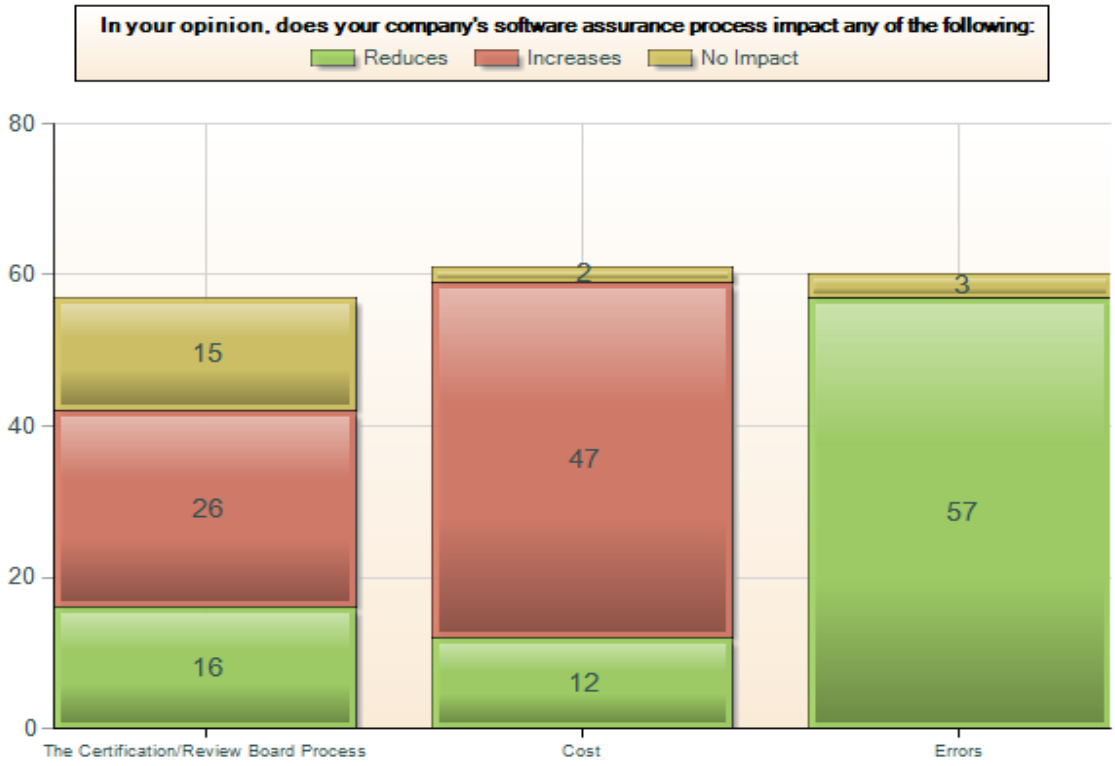


Figure A-16. Impact of software assurance process

Question 23: Do you have any additional comments about software assurance and potential alternative approaches, other than DO-178B or DO-278?

33 Responses:

1. I don't believe the DO-178b process reduces software errors or problems. I would like to see a large, comprehensive, clinical study which verifies whether or not the DO-178b process has an impact on software quality and how and where the impacts occur.
2. If studied in the context of safety engineering and software engineering, DO-178B and DO-278 can be extremely effective in terms of yield when combined with a good systems safety process. Failure to have a good, solid systems and safety process will negate most if not all software process initiatives. Safety and process assurance are holistic and rarely treated as such.
3. The process (DO-178B) is sufficient. Those that argue that 178B is not sufficient and that we need to go farther (i.e., driving things like more dissimilarity) are not looking at the real problems that are generating the issues. The issue that fallout from "178B" are when unqualified, untrained individuals are involved with inadequate technical oversight. No matter how extensive or over-burdensome the process you impose...if inexperienced folks are at the helm...they can produce a product full of errors.
4. Better guidance related to COTS, service history and previously developed software.
5. Not enough emphasis on robust requirements. Generally the problem we find lies in the requirements.
6. I am happy to see applicant to comply with DO-178C and other order.
7. Focus on system assurance.
8. Software assurance will never provide safety beyond the level of design assurance applied to the software requirements. For system or end-user improvements to software assurance, additional efforts should be placed on the development process assurance for system requirements, as those requirements define the software requirements.
9. I recommend looking at Safety Directed Development, just to understand an alternative that has been used.
10. Did not have time to fill in all material.
11. I question the value of MCDC testing.
12. It is not about the standard, it is how the standard is implemented.
13. Safety starts at the system level–ARP4754A would be a good starting point.
14. No.
15. I think the standards shouldn't be changed or updated to match inappropriate development methods—the methods should be updated to meet the criteria and environment of the environment dictated by the application and safety impact.
16. We will have to go into CC (common criteria in the future).
17. Don't use other approaches. Just RTCA/DO-178C, as tailored for specific projects.
18. In the end, it is sometimes comforting to have a process defined and followed, but it is the product that matters above all else.
19. DO-178B and airborne software development has significantly reduced errors to the point where safety issues almost always trace back to bad systems requirements. I have seen time and time again, the problem of the systems requirements say build a square wheel

and DO-178B forces you to build a perfectly square wheel. Everybody knows it, but unless the systems requirements get changed there's nothing that can be done about it.
There is a lot of overlap between DO-178B, CMMI, and AS9115. Finding a way to take advantage of this would reduce cost/overhead and let companies focus more on the goal of safe, correct SW product.

21. DO-178B is very prescriptive. Most other software assurance standards are less prescriptive and more evidence based.
22. No. I believe those two standards would be enough if applied consistently and correctly.
23. None.
24. Security is missing in current development methodology.
25. No.
26. We are in a dilemma where human understanding is being stretched by the mere size of software and computers/code are not yet smart enough to deal with it for us. I also note a NASA symposium remark made back in the 1970's: (roughly) “All of the world's computers aggregated into one would not have the equivalent neural count of a cockroach. However, if Moore's law holds, by 2030 we will have individual computers with effective neural counts beyond ours. Turning those on may be the last significant act of our species.”
27. We have used safety cases (GSN) to help with some of the analysis. It provides a good way of structuring some of the safety arguments. As an example, we found it very useful to structure our robust partitioning analysis.
28. None.
29. The assurance process needs to be part of the engineering development. Not a department trying to force its will on engineering. When engineering believes and owns the process, greater safety and fewer errors happen because they are responsible and not another organization playing policemen.
30. Software assurance per DO-178B(C) includes many overlapping checks and gates. Most organizations that use DO-178B experience a 1-year project cycle or more. In my estimation, what could be accomplished in a month using a process similar to extreme programming would be as safe as DO-178B pedigree, but delivered faster and the implementation would be closer to the original intent of the requirements.
31. DO-178B is great for the technical aspects for a program. However, if a company's culture does not have a strong integrated management process (such as in the CMMI Model), I have seen detrimental effects to the execution of a program. Programs that I have seen using CMMI modeled processes (non-commercial, not using DO178B) have executed strongly on maintaining schedules, cost, and meeting customer commitments; however, the execution of the technical processes are weak without the structure provided by DO-178B. I'd like to see implementation of an integrated process of strong management procedures with strong technical procedures.
32. I see little value in some of the SOI Review required artifacts/activities. A lot of projects are treating them as a checklist item to check off, without really improving the quality or safety of a product.
33. Some of our non-airborne projects follow Agile development methods. In my opinion, that involves a lot of churn and duplication of effort as records are not kept of what has already been decided or accomplished. I do not recommend Agile for safety-critical software development.

Question 23:

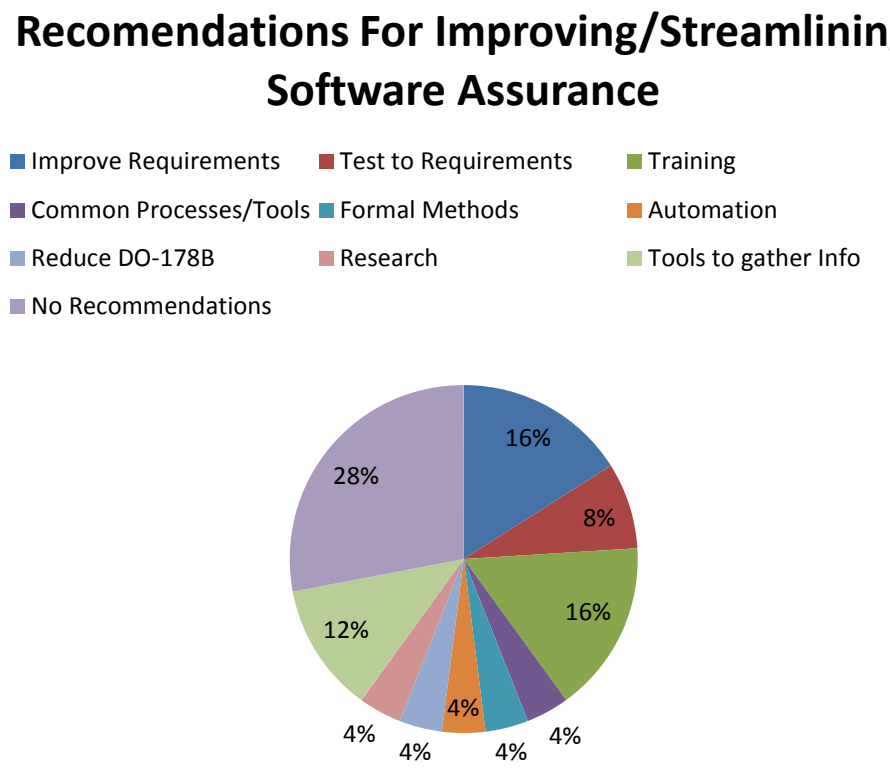


Figure A-17. Recommendations for improving/streamlining software assurance

APPENDIX B—SYSTEMS THEORETIC PROCESS ANALYSIS

In phase 3 of this study, a Systems Theoretic Process Analysis (STPA) was performed on a Flight Guidance System (FGS). The goal of this analysis was to take the results of the STPA analysis and compare them to a fault tree analysis (FTA) to determine if the STPA analysis results would enhance the safety analysis.

The first step in analyzing the FGS was to identify system-level hazards and review/identify safety constraints. For this analysis, the hazard (aircraft drops below or climbs above the pre-selected altitude [PSA]) was used because it was also analyzed in an FTA performed by the same authors of the NASA/CR-2003-212426, Flight Guidance System Requirements Specification. The results of the STPA analysis were compared to the FTA to demonstrate the benefits of using the STPA analysis when software, hardware, commercial off-the-shelf (COTS), and systems analyses are required.

The second step in the STPA analysis is to describe the system control structure by building a control structure diagram, as shown in figure B-1.

Each box represents a controller of the system as defined by one of the four conditions:

1. The controller must have a goal or goals.
2. The controller must be able to affect the state of the system.
3. The controller must be (or contain) a model of the system.
4. The controller must be able to ascertain the state of the system [B.1].

A controller of the system could be humans (such as the pilots and air traffic controllers), software (such as the FGS or COTS), or hardware (such as sensors on the aircraft). Once the controllers of the system are identified, the interactions between the controllers are added to the diagram. The interactions could include commands, data exchanges, and feedback. The goal of an STPA analysis is to ensure the system is controlled in such a way so as not to cause a hazard. To accomplish this goal, the interactions between the controllers are examined. Potential control actions that could lead to aircraft dropping below the PSA are identified. This is accomplished by determining if 1) a required control action between controllers of the system is not provided, 2) an incorrect or unsafe control action is provided, 3) a potentially correct or inadequate control action is provided too late or too early (i.e., at the wrong time), or 4) a correct control action is stopped too soon.

The components of the system that exert control are highlighted in the boxes of figure B-1 and are described below:

- Flight Control Panel (FCP)–This panel is the primary device with which the flight crew interacts with the FGS. The crew can change vertical and lateral modes via this panel; turn the flight director (FD) on and off; engage and disengage the autopilot (AP); and transfer control between the left and right FGSs. The Synchronization (SYNC) and AP disengage can also be activated on the yokes and sent to the FCP, then routed through to the FGS. The pilots can also activate the GA switch and AP disengage on the throttles via the FCP.
- Autopilot (AP)–The AP commands the control surfaces based on the pitch and roll commands generated by the FGS.
- Navigation Radios–The FGS receives navigation information from several sources, including Very High Frequency Omni-Directional Range (VOR) for lateral navigation (NAV), Localizer for precision lateral approach, and Glideslope for precision vertical approach.
- Air Data System (ADS)–The ADS provides information about the aircraft state sensed from the surrounding air mass, such as the pressure altitude and indicated airspeed (IAS).
- Flight Management System (FMS)–The FMS's primary function is to manage the flight plan. The FMS contains flight data (waypoints, airways, airports, and runways) that are used by the FGS to provide guidance commands along a flight plan.
- Flight Director (FD): This panel displays the pitch and roll guidance commands to the pilot and copilot on the primary flight display (PFD).
- Hazard: Aircraft drops below the PSA.

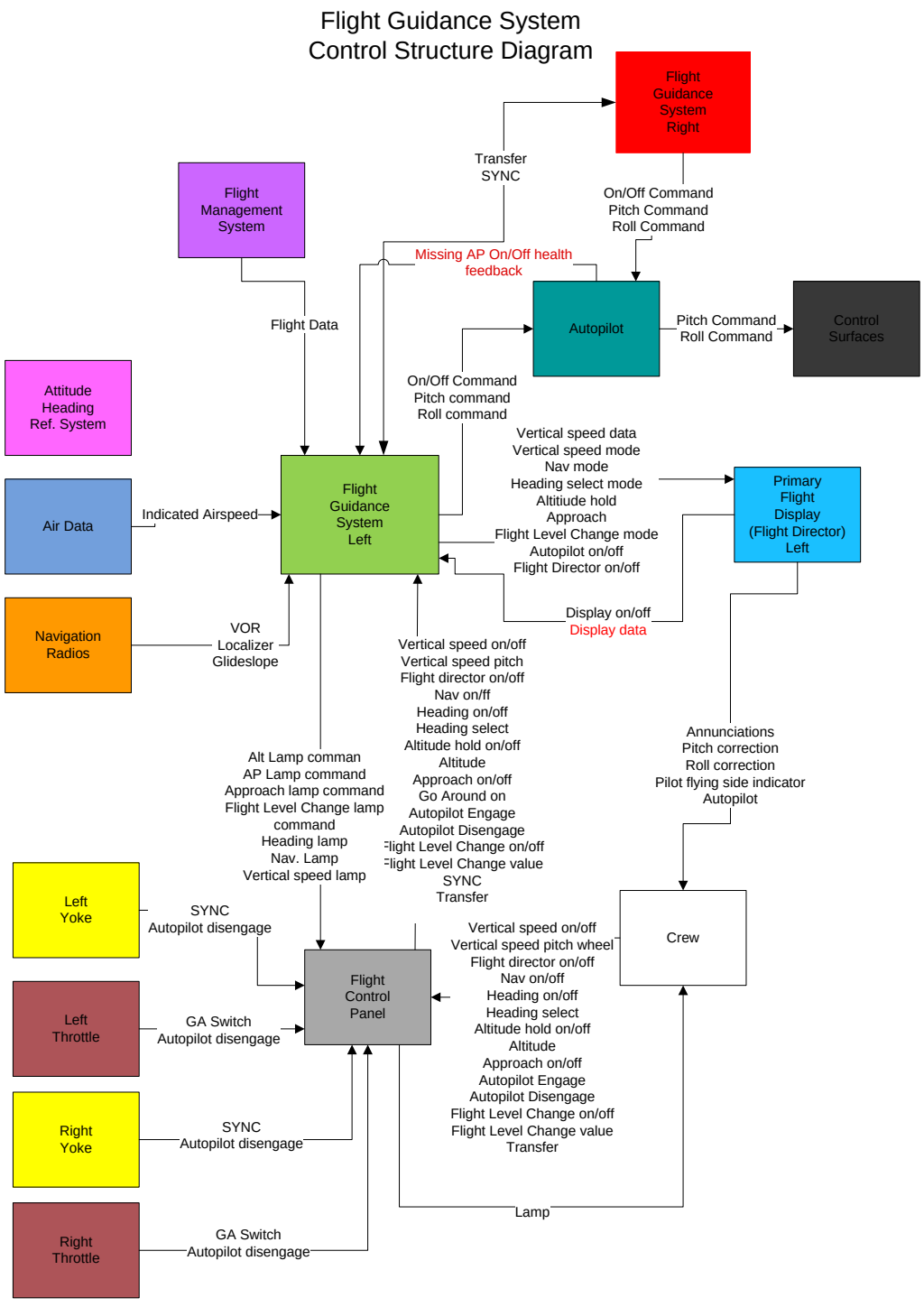


Figure B-1. FGS control structure diagram

B.1 FGS LEFT TO AP: AP ON COMMAND INADEQUATE CONTROL ACTION

B.1.1 UNSAFE CONTROL ACTION: INADVERTENT AP ON COMMAND

AP On command is triggered inadvertently (see figure B-2).

Some possible causes:

- AP On command sent from wrong side
- Mode confusion

Result: If the AP On command is triggered inadvertently, the AP may attempt to fly the plane to a previous waypoint or preselected altitude (PSA). It may also result in mode confusion and the pilot may continue to attempt to fly the plane. There are currently no requirements for whether the pilot, the AP, or both commands will be acted on.

Mitigation: Requirements are needed to specify prioritization between the pilot and the AP. These requirements should include whether to leave the AP on if the throttle or yoke are active. Requirements are also needed to specify validity checks on both the AP commands and any flight data coming from the FMS.

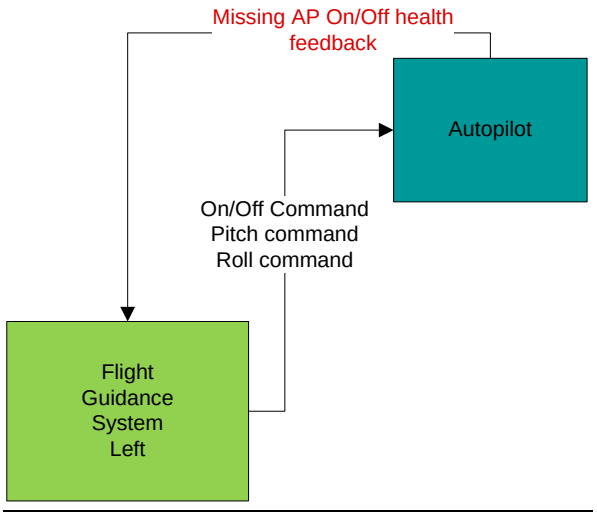


Figure B-2. FGS left interactions with AP

B.1.2 UNSAFE CONTROL ACTION: AP ON COMMAND IS NOT PROVIDED/MISSING

FGS does not send the AP On command.

Some possible causes:

- FGS is in the process of transferring control between sides (left to right or right to left) when AP On command is sent.
- Mode confusion.
- System is in a mode where the AP is disengaged such as Go Around (GA) mode.
- FGS loses power.
- AP On command sent from FGS side not in control.

Result: Pilot quits commanding controlled surfaces, resulting in unknown aircraft behavior and failure to hold the PSA.

Mitigation: Requirements specify feedback to pilot as to which FGS is in control. Requirement needs to specify what happens during transitioning. Feedback to pilot could help to mitigate this scenario. Requirements specify an AP lamp, but it is not based on a health and status message from the AP. Requirements are needed to specify what happens if the FGS loses power.

B.1.3 UNSAFE CONTROL ACTION: FGS COMMANDS AP TO TURN ON AT THE WRONG TIME

FGS command to turn on the AP arrives at wrong time (e.g., command is late).

Some possible causes:

- FGS sends On command after reset if the AP was on prior to reset.
- FGS completes Pilot Flying Transfer or another task and then sends AP On.

Result: Mode confusion concerning the status of the AP leads to incorrect decisions.

Mitigation: Requirements are needed to specify what happens to all the modes when the FGS is reset, powered down, or loses power. Requirements are needed regarding the system status before it issues any commands. Requirements are needed specifying system response time to pilot commands.

B.1.4 AP ON COMMAND IS STOPPED TOO SOON

The AP On command is interrupted or stopped before completion.

Some possible causes: This scenario is looking for ways in which a continuous data stream or discrete command could be interrupted. From the requirements, it is unclear if this command is discrete or whether the signal is sent continuously.

Result: If the AP On command can be stopped too soon, the AP may not maintain altitude, and because of potential mode confusion, pilots might also not be commanding the aircraft.

Mitigation: Requirements are needed to specify what to do if a partial command is sent or received. Requirements must be in place that specify what to do if an Off command arrives before the system is finished implementing the On command. At the very least, feedback to the pilot must reflect the actual state of the system. Requirements for feedback from the AP to the FGS are needed.

B.2 FGS TO AP: AP OFF COMMAND INADEQUATE CONTROL ACTION

B.2.1 UNSAFE CONTROL ACTION: INADVERTENT AP OFF COMMAND SENT TO AP

AP Off command is triggered inadvertently.

Some possible causes:

- AP Off command is sent from off-side FGS.
- FGS inadvertently enters a mode in which AP is disengaged (e.g., GA mode).
- Pilot transfer results in AP being turned off.

Result: AP does not issue any controls and does not maintain the pre-selected altitude.

Mitigation: Requirements are needed to specify how commands from the off-side FGS are handled. Requirements also need to be considered for limiting when the Go Around mode can be triggered. Requirements are needed to specify how the pilot transfer will occur and if any other modes or components will be affected.

B.2.2 UNSAFE CONTROL ACTION: AP OFF COMMAND IS NOT PROVIDED/MISSING

Some possible causes:

- FGS is in the process of transferring control between sides (left to right or right to left) when AP Off command is sent.
- Mode confusion.
- FGS loses power.
- AP Off command sent from FGS side not in control.

Result: Pilot and AP both attempt to command the controlled surfaces, resulting in unknown aircraft behavior and failure to hold the PSA. There are currently no requirements for whether the pilot, the AP, or both commands will be acted on.

Mitigation: Requirements are needed to specify prioritization between the pilot and the AP. These requirements should include whether to leave the AP on if the throttle or yoke are active. Requirements do specify feedback to the pilot regarding which FGS is in control, but requirements need to specify what happens during transitioning. Feedback to the pilot could help

to mitigate this scenario. Requirements specify an AP lamp, but it is not based on a health and status message from the AP. Requirements are also needed to specify what happens if the FGS loses power or is restarted.

B.2.3 UNSAFE CONTROL ACTION: AP OFF COMMAND IS SENT AT THE WRONG TIME

AP Off is sent at the wrong time (e.g., too late).

Some possible causes:

- Inadvertent activation of the GA mode
- Processing delay

Results: AP continues to control aircraft when it should be off or turns off unexpectedly and does not maintain the pre-selected altitude when the pilot expects it to be maintained.

Mitigation: Although there are requirements that specify the triggering of GA mode, it is unclear what the FGS software does with the GA command. More details need to be specified as to how the software handles the GA input command from the FCP. Requirements specify an AP lamp, but it is not based on a health and status message from the AP. The AP lamp should be based on feedback from the AP rather than the requested state of the AP.

B.2.4 UNSAFE CONTROL ACTION: AP OFF COMMAND IS STOPPED TOO SOON

The AP off command is interrupted or stopped before completion.

Possible cause:

- This scenario is looking for ways in which a continuous data stream or discrete command could be interrupted. From the requirements, it is unclear if this command is discrete or whether the signal is sent continuously.

Result: If the AP Off command can be stopped too soon, the AP may not maintain altitude, roll, or both. Pilots, because of potential mode confusion, might not be commanding the aircraft.

Mitigation: Requirements are needed to specify what to do if a partial command is sent or received. Requirements must be in place specifying what to do if an On command arrives before the system is finished implementing the Off command. At the least, feedback to the pilot must reflect the actual state of the system. Requirements for feedback from the AP to the FGS are needed.

B.3 FGS TO AP: PITCH COMMAND INADEQUATE CONTROL ACTION

B.3.1 UNSAFE CONTROL ACTION: PITCH COMMAND SENT INADVERTENTLY

FGS pitch command erroneously directs AP to an altitude other than PSA.

Some possible causes:

- During independent mode, pitch command from non-pilot flying side is erroneously processed.
- During pilot transfer, a command is processed before the transfer is complete.
- The FGS software sends erroneous pitch value.

Result: The FGS commands the AP to pitch up or down, resulting in the plane not being at the correct altitude.

Mitigation: Requirements are needed to specify the timing of the altitude data input to the FGS and how the data are checked for corruption, hacking, etc. Requirements are needed to specify the pilot transfer, particularly when the sides are out of sync. Requirements are needed to specify how the pitch values are checked for correctness.

B.3.2 UNSAFE CONTROL ACTION: PITCH COMMAND NOT PROVIDED/MISSING

Some possible causes:

- FGS is powered down.
- FGS had an inadvertent mode change (see Crew – FGS interactions).
- FGS internal model of AP indicates AP is off.

Results: If aircraft is required to climb to a new PSA or maintain the current PSA, the AP will not command surfaces—and aircraft may drop below PSA.

Mitigation: Power down, restart, and initialization requirements are needed for the FGS and AP. Feedback from the AP to the FGS needs to be defined. In addition, the requirements for leaving a mode need to be examined to determine whether the dangers of leaving a mode inadvertently outweigh the dangers of being stuck in that mode.

B.3.3 UNSAFE CONTROL ACTION: FGS SENDS PITCH COMMAND TOO EARLY, TOO LATE, OR OUT OF SEQUENCE

Some possible causes:

- FGS uses obsolete flight data.
- Pitch command is delayed because of the processing of another command.
- Obsolete pitch command is sent after pilot transfer.

Result: FGS pitch up/down command is sent after aircraft reaches new PSA.

Mitigation: Requirements are needed to specify when flight data and commands are obsolete.

B.3.4 UNSAFE CONTROL ACTION: FGS PITCH COMMAND STOPPED TOO SOON

Result: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously.

Mitigation: Requirements are needed specifying how the system should handle partial commands or commands that come in while another command is being processed. Feedback is needed from the AP to the FGS regarding the status of the AP.

B.4 FGS TO AP: ROLL COMMAND

The roll command may indirectly affect the altitude. There are currently no requirements correcting for pitch during a roll maneuver. It is unknown if this would be handled by the FGS or the AP. This command will need to be re-examined as the requirements mature.

B.5 AP TO FGS: HEALTH AND STATUS FEEDBACK

At this point, health and status feedback from the AP to the FGS is needed but not provided. It is unknown what form that feedback will take. Some common forms would be a query from the FGS to the AP, a continuous or discrete signal from the AP when it is on, a continuous or discrete signal from the AP indicating whether it is on or off, and regular messages from the AP to the FGS stating what the AP is doing. The following analysis of the result should provide additional guidance as the requirements are written.

B.5.1 UNSAFE CONTROL ACTION: INADVERTENT HEALTH AND STATUS FEEDBACK

Results: If provided when not expected, the feedback might interfere with the processing of the FGS.

Mitigation: Establish clear priorities for when to process the AP health and status feedback.

B.5.2 UNSAFE CONTROL ACTION: HEALTH AND STATUS FEEDBACK NOT PROVIDED/MISSING

Results: The FGS will not know the status of AP.

Mitigation: Clear requirements should be provided of what the FGS should do if feedback is missing and how long it has to be missing before acting. Included should be requirements for when and how to notify the pilots.

B.5.3 UNSAFE CONTROL ACTION: HEALTH AND STATUS FEEDBACK IS SENT AT THE WRONG TIME

Result: If the feedback is late, the FGS may use obsolete data.

Mitigation: Depending on the type of feedback used, the FGS should have a method to determine if the data is relevant or not. The requirements should include what to do if the data being sent by the AP are incorrect.

B.5.4 UNSAFE CONTROL ACTION: HEALTH AND STATUS FEEDBACK STOPPED TOO SOON

Results: The FGS will not have accurate information on the status of the AP.

Mitigation: Clear requirements should be provided of what the FGS should do if feedback is missing and how long it has to be missing before acting. Included should be requirements for when and how to notify the pilots.

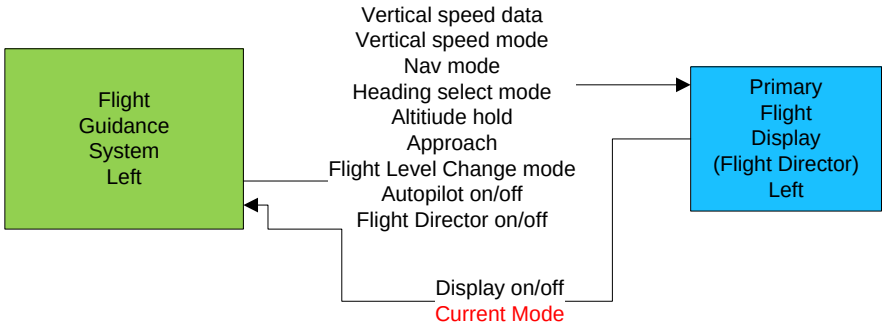


Figure B-3. FGS left interactions with PFD

B.6 FGS TO PFD: VERTICAL SPEED DATA INADEQUATE CONTROL ACTION

B.6.1 UNSAFE CONTROL ACTION: FGS INADVERTENTLY SENDS VERTICAL SPEED DATA

Some possible causes:

- Incorrect vertical speed (VS) data are sent.
- VS data are sent at the wrong rate.

Result: Incorrect situational awareness leads to errors in climb or descent rates. Maintaining these incorrect rates can lead to dropping below the pre-selected altitude. In addition, handling the overload of VS data may cause other data to be delayed.

Mitigation: All commands and data should have expected rates and obsolescence defined. Components receiving data need to have clear requirements on how to handle data received at the wrong time.

B.6.2 UNSAFE CONTROL ACTION: FGS DOES NOT SEND VS DATA OR THEY ARE MISSING.

Some possible causes:

- FGS loses power.
- Flight director loses power.
- FGS thinks flight director is "Off."

Result: The pilot has no guidance regarding rate of climb; must rely on visual cues, and may not level off at the PSA.

Mitigation: The requirements should specify how the system behaves during power loss, restart, and initialization. The requirements should specify how the software processes the flight director's on/off commands. It should specify what prioritization the flight director's on/off command should have, thus reducing the possibility of the command being permanently delayed.

B.6.3 UNSAFE CONTROL ACTION: FGS SENDS VS DATA AT THE WRONG TIME

Some possible causes:

- Prioritization issues result in delay in calculating or transmitting VS data.
- FGS does not have VS data to transmit.

Result: When the data should have arrived, there are only old data or no data to provide guidance for the pilot. When the data do arrive and display, they are already obsolete.

Mitigation: The requirements should specify which commands have priority and the required response time for each command. Data to the pilots must have requirements defining when the data are obsolete and what to display to the pilots.

B.6.4 UNSAFE CONTROL ACTION: FGS STOPS SENDING VS DATA TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify the type of signal/command the VS data send to the flight director and what will be done to prevent partially delivered messages.

B.7 FGS TO PFD: VS MODE INADEQUATE CONTROL ACTION

B.7.1 UNSAFE CONTROL ACTION: FGS INADVERTENTLY SENDS VS MODE FEEDBACK

Some possible causes:

- Select and deselect for VS are the same, so the system leaves VS mode immediately.
- Offside FGS sends VS mode feedback to onside FD.

Result: The aircraft does not hold the PSA because the VS mode is turned off when the pilot thinks it is turned on.

Mitigation: The triggering conditions for selecting and deselecting the VS mode must be mutually exclusive in the requirements. Because the flight directors can receive data from either FGS, the requirements should consider how to mark data displayed from the other side, particularly during independent mode.

B.7.2 UNSAFE CONTROL ACTION: FGS DOES NOT SEND VS MODE FEEDBACK OR IT IS MISSING

Some possible causes:

- FGS thinks both flight displays are off and AP is disengaged.
- FGS thinks it is in an overspeed condition.
- FGS thinks the system is transferring to the non-pilot flying side.

Result: The flight display may continue to show the previous mode or may not show any mode. The pilot experiences mode confusion. How he will react depends on visual clues available and former mode.

Mitigation: Flight director must be capable of marking data as obsolete. Response times to pilot commands must be defined. If the lamps on the control panel are illuminated independently of the flight director annunciations, this could help to mitigate mechanical causes.

B.7.3 UNSAFE CONTROL ACTION: FGS SENDS VS MODE FEEDBACK AT THE WRONG TIME

Some possible causes:

- System is processing another command, and feedback to FD is delayed.
- Pilot transfers results in obsolete data being sent to flight director.

Result: The flight display may continue to show the previous mode or may not show any mode. The pilot experiences mode confusion, and how he will react depends on visual clues available and former mode. When the obsolete mode data are displayed, mode confusion could result in

the pilot inadvertently turning on the VS mode when he intends for it to be off (on and off are the same button).

Mitigation: The triggering conditions for selecting and deselecting the VS mode must be mutually exclusive in the requirements. Because the flight directors can receive data from either FGS, the requirements should consider how to mark data displayed from the other side, particularly during independent mode. Data should have obsolescence defined and the flight directors must be able to mark data as obsolete.

B.7.4 UNSAFE CONTROL ACTION: FGS STOPS SENDING VS MODE FEEDBACK TOO SOON.

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways for which a continuous data stream or command could be interrupted. From the requirements, it is unclear whether this command is a one-time command or if the signal is sent continuously. The requirements should specify the type of signal/command the VS data send to the flight director and what will be done to prevent partially delivered messages.

B.8 FGS TO PFD: FLIGHT LEVEL CHANGE MODE INADEQUATE CONTROL ACTION

B.8.1 UNSAFE CONTROL ACTION: FGS INADVERTENTLY SENDS FLIGHT LEVEL CHANGE MODE FEEDBACK

Some possible causes:

- Select and deselect for flight level change are the same, so the system leaves flight level change mode immediately.
- Offside FGS sends flight level change mode feedback to onside FD.

Result: The aircraft does not hold the PSA because the flight level change mode is turned off and the pilot is subject to mode confusion.

Mitigation: The triggering conditions for selecting and deselecting the flight level change mode must be mutually exclusive in the requirements. Because the flight directors can receive data from either FGS, the requirements should consider how to mark data displayed from the other side, particularly during independent mode.

B.8.2 UNSAFE CONTROL ACTION: FGS DOES NOT SEND FLIGHT LEVEL CHANGE MODE FEEDBACK OR IT IS MISSING

Some possible causes:

- FGS thinks both flight displays are off and AP is disengaged.
- FGS thinks it is in an overspeed condition.
- FGS thinks the system is transferring to the non-pilot flying side.

Result: The flight display may continue to show the previous mode or may not show any mode. The pilot experiences mode confusion, and how he will react depends on the visual clues available and the former mode.

Mitigation: FD must be capable of marking data as obsolete. Response times to pilot commands must be defined. If the lamps on the control panel are illuminated independently of the flight director annunciations, this could help to mitigate mechanical causes.

B.8.3 UNSAFE CONTROL ACTION: FGS SENDS FLIGHT LEVEL CHANGE MODE FEEDBACK AT THE WRONG TIME

Some possible causes:

- System is processing another command and feedback to FD is delayed.
- Pilot transfer results in obsolete data being sent to FD.

Result: The flight display may continue to show the previous mode or may not show any mode. The pilot experiences mode confusion and how he will react depends on visual clues available and former mode. When the obsolete mode data are displayed, mode confusion could result in the pilot inadvertently turning on the FLC mode when he intends for it to be off (on and off are the same button).

Mitigation: The triggering conditions for selecting and deselecting the flight level change mode must be mutually exclusive in the requirements. Because the FDs can receive data from either FGS, the requirements should consider how to mark data displayed from the other side, particularly during independent mode. Data should have obsolescence defined and the FDs must be able to mark data as obsolete.

B.8.4 UNSAFE CONTROL ACTION: FGS STOPS SENDING ‘FLIGHT LEVEL CHANGE MODE FEEDBACK TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify the type of

signal/command the VS data sends to the FD and what will be done to prevent partially delivered messages.

B.9 FGS TO PFD: ALTITUDE HOLD MODE FEEDBACK INADEQUATE CONTROL ACTION

B.9.1 UNSAFE CONTROL ACTION: ALTITUDE HOLD MODE FEEDBACK ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends altitude hold (ALT) mode feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.9.2 UNSAFE CONTROL ACTION: ALT MODE FEEDBACK NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends ALT mode feedback to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.9.3 UNSAFE CONTROL ACTION: ALT MODE FEEDBACK ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to the wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.9.4 UNSAFE CONTROL ACTION: ALT MODE FEEDBACK STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or if the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.10. FGS TO PFD: APPROACH MODE FEEDBACK INADEQUATE CONTROL ACTION

B.10.1 UNSAFE CONTROL ACTION: APPROACH MODE FEEDBACK ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends incorrect mode feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.10.2 UNSAFE CONTROL ACTION: APPROACH MODE FEEDBACK NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends mode feedback to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.10.3 UNSAFE CONTROL ACTION: APPROACH MODE FEEDBACK ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.10.4 UNSAFE CONTROL ACTION: APPROACH MODE FEEDBACK STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.11. FGS TO PFD: NAV CHANGE MODE INADEQUATE CONTROL ACTION

B.11.1 UNSAFE CONTROL ACTION: NAV CHANGE MODE FEEDBACK ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends incorrect mode feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.11.2 UNSAFE CONTROL ACTION: NAV CHANGE MODE FEEDBACK NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends mode feedback to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.11.3 UNSAFE CONTROL ACTION: NAV CHANGE MODE FEEDBACK ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.11.4 UNSAFE CONTROL ACTION: NAV CHANGE MODE FEEDBACK STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is to look for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.12 FGS TO PFD: HDG MODE FEEDBACK INADEQUATE CONTROL ACTION

B.12.1 UNSAFE CONTROL ACTION: HDG MODE FEEDBACK ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends incorrect mode feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.12.2 UNSAFE CONTROL ACTION: HDG MODE FEEDBACK NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends mode feedback to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.12.3 UNSAFE CONTROL ACTION: HDG MODE FEEDBACK ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.12.4 UNSAFE CONTROL ACTION: HDG MODE FEEDBACK STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.13 FGS TO PFD: AP ON/OFF FEEDBACK INADEQUATE CONTROL ACTION

B.13.1 UNSAFE CONTROL ACTION: AP ON/OFF FEEDBACK ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends incorrect feedback.
- FGS does not know actual state of AP.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined. Feedback from AP is needed.

B.13.2 UNSAFE CONTROL ACTION: AP ON/OFF FEEDBACK NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends feedback to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.13.3 UNSAFE CONTROL ACTION: AP ON/OFF FEEDBACK ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.13.4 UNSAFE CONTROL ACTION: AP ON/OFF FEEDBACK STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.14 FGS TO PFD: FLIGHT DIRECTOR ON/OFF COMMAND INADEQUATE CONTROL ACTION

B.14.1 UNSAFE CONTROL ACTION: FLIGHT DIRECTOR ON/OFF COMMAND ARRIVES INADVERTENTLY

Some possible causes:

- System is transitioning from one mode to another and sends incorrect feedback.
- Offside FGS sends incorrect command.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.14.2 UNSAFE CONTROL ACTION: FLIGHT DIRECTOR ON/OFF COMMAND NOT PROVIDED/MISSING

Some possible causes:

- System is transitioning from one mode to another and does not send feedback.
- FGS sends command to wrong display.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to pilot commands need to be defined.

B.14.3 UNSAFE CONTROL ACTION: FLIGHT DIRECTOR ON/OFF COMMAND ARRIVES AT WRONG TIME

Some possible causes:

- System is transitioning from one mode to another and sends feedback late.
- Pilot Flying Transfer delays feedback.

Results: Pilot mode confusion could result in flying to wrong altitude.

Mitigation: Requirements for response time to Pilot commands need to be defined.

B.14.4 UNSAFE CONTROL ACTION: FLIGHT DIRECTOR ON/OFF COMMAND STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify what will be done to prevent partially delivered messages.

B.15 CREW TO FGS: VS ON/OFF COMMAND INADEQUATE CONTROL ACTION

The following sections refer to figure B-4 for all the inadequate control actions, possible causes, results, and mitigations.

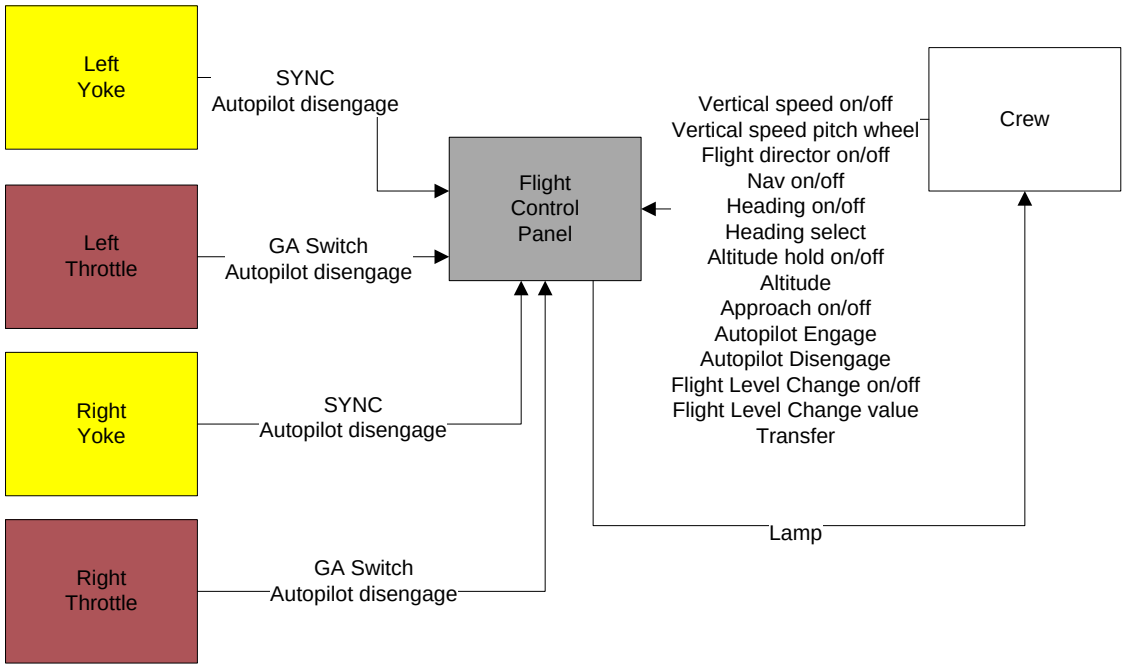


Figure B-4. FGS left interactions with crew via flight control panel, yokes throttle

B.15.1 UNSAFE CONTROL ACTION: FGS RECEIVES INADVERTENT VS ON/OFF COMMAND

Some possible causes:

- Because the VS Switch toggles VS Mode on and off, mode confusion is one way it could be turned on if the pilot does not have adequate feedback about the state of the mode.
- In the current requirements, there are two macros that can both be true (Select_VS and Deselect_VS), which means that the Variable VS could bounce back and forth between Cleared and Selected.

Results: Aircraft drops below PSA because VS Mode is turned off, and VS is no longer maintained or is turned on with incorrect values.

Mitigation: Select_VS and Deselect_VS need to be mutually exclusive in the specification. Because of the potential for confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The VS light currently appears to be based on the actual state rather than on the requested state; however, the VS is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.15.2 UNSAFE CONTROL ACTION: VS ON/OFF COMMAND IS NOT PROVIDED OR MISSING

Some possible causes:

- Mode confusion as to whether or not the aircraft is in VS Mode.
- Pilot selects VS Mode, but system does not register because we are in an overspeed condition or it is immediately deselected because Modes is Off, pilot flying is being transferred, or non-basic Vertical Mode is activated.

Result: System remains in whatever mode it was previously in or reverts to the basic Vertical Flight Mode (Pitch Mode). Because Pitch Mode holds the aircraft at a fixed pitch angle and the aircraft may have been pitched up or down when Pitch Mode was selected, the aircraft may travel down (below the PSA) rather than maintain level or move up.

Mitigation: Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The VS light currently appears to be based on the actual state rather than on the requested state; however, the VS is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.15.3 UNSAFE CONTROL ACTION: VS ON/OFF COMMAND RECEIVED AT WRONG TIME

Some possible causes:

- Pilot mode confusion
- Pilot error
- Processing delay

Result: Aircraft drops below PSA because VS Mode is turned on with incorrect values. Turning it on at the wrong time could result in the plane constantly climbing or descending, depending on the current state of aircraft and previous mode. Turning it off at the wrong time could result in the aircraft dropping below PSA because VS Mode is turned off and VS is no longer maintained. In addition, there are two macros that can both be true (Select_VS and Deselect_VS), which means that the Variable VS could bounce back and forth between Cleared and Selected.

Mitigation: Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The VS light currently appears to be based on the actual rather than the requested state; however, the VS is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design. Requirements need to specify response times to pilot inputs.

B.15.4 UNSAFE CONTROL ACTION: VS ON/OFF COMMAND IS STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify the type of signal/command the VS data send to the flight director and what will be done to prevent partially delivered messages.

B.16 CREW TO FGS: VS PITCH WHEEL INADEQUATE CONTROL ACTION

B.16.1 UNSAFE CONTROL ACTION: INADVERTENT VS PITCH WHEEL SIGNAL RECEIVED

Results: If the system is in Flight Level Change mode, ALT mode, Altitude Select, or Vertical Go Around (VGA), it will exit to Pitch Mode. It will also change the pitch to a random amount—either positive or negative.

Mitigation: Requirements need to specify how intended Pitch Wheel motions can be separated from random motions. At the least, changes in mode should be highlighted in such a way as to catch the pilot's attention, such as blinking until confirmed.

B.16.2 UNSAFE CONTROL ACTION: VS PITCH WHEEL SIGNAL IS NOT PROVIDED OR MISSING

Some possible causes:

- System is in VAPPR
- Mode confusion
- Mechanical error

Results: In most cases, the pilot will be able to react quickly enough. However, if there is no clear way for the pilot to tell how much he has moved the Pitch Wheel, he might move it again, resulting in an unexpected dive or climb.

Mitigation: Requirements are needed to either limit the amount of Pitch Wheel change allowed in a given time or indicate to the pilot that the wheel has been moved.

B.16.3 UNSAFE CONTROL ACTION: VS PITCH WHEEL SIGNAL IS SENT AT THE WRONG TIME

Some possible causes:

- Delay in signal processing, such as during SYNC, Transfer, and Restart.
- Feedback and directions to pilot are obsolete.

Results: Aircraft does not follow intended route, including a potential drop below pre-selected altitude.

Mitigation: Specify maximum response times to pilot inputs and obsolescence for all data and feedback.

B.16.4 UNSAFE CONTROL ACTION: VS PITCH WHEEL SIGNAL IS STOPPED TOO SOON

Some possible causes:

- FCP or FGS begins processing signal before pilot finishes turning wheel.
- Lack of situational awareness leads to pilot not turning wheel enough.

Results: Pitch will not increase or decrease by the desired amount.

Mitigation: Specify maximum response times to pilot inputs and obsolescence for all data and feedback. Provide feedback for all pilot inputs.

B.17 CREW TO FGS: FLIGHT LEVEL CHANGE MODE ON/OFF COMMAND INADEQUATE CONTROL ACTION

B.17.1 UNSAFE CONTROL ACTION: INADVERTENT FLIGHT LEVEL CHANGE MODE ON/OFF COMMAND SENT TO FGS

Some possible causes:

- Pilot mode confusion as to whether Flight Level Change is already selected or deselected.
- Pilot inadvertently presses Flight Level Change switch.
- Pilot inadvertently activates non-basic Vertical mode, activates Pilot Flying Transfer, rotates the VS Pitch Wheel, or turns Modes off.

Because the Flight Level Change Switch toggles Flight Level Change Mode on and off, aircraft could drop below selected altitude if it is selected inadvertently—resulting in it being turned off when the pilot thinks he is turning it on or the reverse. In addition, there are two macros that can both be true (Select_FLC and Deselect_FLC), which means that the Variable FLC could bounce back and forth between Cleared and Selected.

Results: Turning the Flight Level Change Mode off unexpectedly could result in the airspeed decreasing, resulting in decreased lift and the aircraft dropping below the PSA. It would then default into either VS mode or ALT mode and it is unclear what values it would use in those modes. Turning it on unexpectedly could result in airspeed falling off if no airspeed has been selected or if only an obsolete value is available. It could also result in prematurely exiting ALT mode or Altitude Select (ALTSEL) mode. It cannot activate during vertical approach mode (VAPPR) unless there is an overspeed condition.

Mitigation: Select_FLC and Deselect_FLC need to be mutually exclusive in the specification. Because of the potential for confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The FLC light currently appears to be based on the actual rather than the requested state; however, the FLC is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design and will require a review of the design.

B.17.2 UNSAFE CONTROL ACTION: FLIGHT LEVEL CHANGE MODE ON/OFF COMMAND IS NOT PROVIDED/MISSING

Some possible causes:

- Pilot mode confusion as to whether Flight Level Change is already selected or deselected.
- Pilot presses Flight Level Change Switch to select Flight Level Change Mode, but system does not register because we are in VAPPR.
- Pilot presses Flight Level Change Switch to deselect Flight Level Change Mode, but system does not register because we are in overspeed condition.

Results: System may remain in whatever mode it was in previously or revert to the basic Vertical Flight Mode (Pitch Mode). Because Pitch Mode holds the aircraft at a fixed pitch angle and the aircraft may have been pitched up or down when Pitch Mode is selected, the aircraft may travel down (below PSA) rather than maintain level or move up. If it remains off, airspeed will not be maintained and the aircraft may fall below the PSA because of a lack of lift.

Mitigation: Flight Level Change Switch is intended to toggle Flight Level Change Mode on and off. The macros Select_FLC and Deselect_FLC are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The FLC light currently appears to be based on actual rather than requested; however the FLC is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.17.3 UNSAFE CONTROL ACTION: FLIGHT LEVEL CHANGE MODE ON/OFF COMMAND IS SENT AT THE WRONG TIME

One possible cause is that a delay occurs in pressing or processing the button, perhaps because of mode confusion.

Result: Turning the Flight Level Change Mode off at the wrong time could result in the airspeed decreasing, resulting in decreased lift and the aircraft dropping below the PSA. It would then default into either VS mode or ALT mode, and it is unclear what values it would use in those modes. Turning it on at the wrong time could result in airspeed falling off if no airspeed has been selected or if only an obsolete value is available. It could also result in prematurely exiting ALT

mode or Altitude Select (ALTSEL) mode. It cannot activate during VAPPR unless there is an overspeed condition.

Mitigation: Select_FLC and Deselect_FLC should be revised to eliminate non-determinism. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The FLC light currently appears to be based on the actual state rather than the requested state; however, the FLC is internal to the FGS and although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.17.4 UNSAFE CONTROL ACTION: FLIGHT LEVEL CHANGE MODE ON/OFF COMMAND IS STOPPED TOO SOON

This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command. No scenarios were found for this inadequate control action to occur. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.18 CREW TO FGS: ALT ON/OFF COMMAND INADEQUATE CONTROL ACTION

B.18.1 UNSAFE CONTROL ACTION: INADVERTENT ALT ON/OFF COMMAND SENT TO FGS

Some possible causes:

- Pilot does not realize ALT is already selected or deselected.
- Pilot inadvertently presses ALT Switch.
- Pilot inadvertently activates non-basic Vertical mode (VS mode, Flight Level Change mode, ALT mode, Altitude Select mode, VAPPR, or VAPPR mode), activates Pilot Flying Transfer, rotates the VS Pitch Wheel, or turns modes off.

Because the ALT switch toggles ALT mode on and off, aircraft could drop below selected altitude if it is selected inadvertently, resulting in it being turned off instead of on or on instead of off. In addition, there are two macros that can both be true (Select_ALT and Deselect_ALT), which means that the Variable ALT could bounce back and forth between Cleared and Selected.

Results: Turning the ALT mode off unexpectedly will result in dropping below the current altitude, which may be the PSA. Turning it on unexpectedly could result in holding an incorrect altitude if no altitude has been selected, if only an obsolete value is available, or if the aircraft has not yet reached the PSA (technically it will not drop below, but it will remain below). It could also result in prematurely exiting another mode. It cannot activate during VAPPR.

Mitigation: Select_ALT and Deselect_ALT need to be mutually exclusive in the specification. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this

reflects the actual state rather than the requested state. The Alt light currently appears to be based on the actual state rather than the requested state; however, the ALT is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.18.2 UNSAFE CONTROL ACTION: ALT ON/OFF COMMAND IS NOT PROVIDED/MISSING

Some possible causes:

- Pilot presses ALT switch to select ALT mode, but system does not register because aircraft is in VAPPR.
- ALT switch to deselect ALT mode does not register.

Result: If the system is in another mode, it will remain in that mode. This may or may not result in dropping below the PSA. If the system is in VAPPR, the aircraft will travel down (below PSA) rather than maintain level or move up. If the previous mode was ALT, then the aircraft will not transition to VS and will remain at current altitude rather than transitioning to a new altitude.
Mitigation: ALT switch is intended to toggle ALT mode on and off. The macros Select_ALT and Deselect_ALT are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback, indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The Alt light currently appears to be based on the actual state rather than the requested state; however, the ALT is internal to the FGS and although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.18.3 UNSAFE CONTROL ACTION: ALT ON/OFF COMMAND IS SENT AT THE WRONG TIME

Some possible causes:

- Pilot mode confusion
- Pilot error
- Pilot bumps ALTSEL_Target_Altitude

Result: Because the ALT switch toggles ALT mode on and off, aircraft could drop below selected altitude if it is selected at the wrong time, resulting in the aircraft not responding in a timely manner and in mode confusion. If the pilot presses the switch a second time, the aircraft will be back in the mode in which it started rather than in the intended state. Turning it on or leaving it on at the wrong time could result in the plane holding an incorrect altitude. Turning it off or leaving it off could result in failure to hold the PSA. In addition, there are two macros that can both be true (Select_ALT and Deselect_ALT), which means that the Variable ALT could bounce back and forth between Cleared and Selected.

Mitigation: Select_ALT and Deselect_ALT should be revised to eliminate non-determinism. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The ALT light currently appears to be based on the actual state rather than the requested state; however, the ALT is internal to the FGS and although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design. Signal-processing requirements should give minimum and maximum system response time to pilot input.

B.18.4 UNSAFE CONTROL ACTION: ALT ON/OFF COMMAND IS STOPPED TOO SOON

This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command. No scenarios were found for this inadequate control action to occur. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.19 CREW TO FGS: APPROACH ON/OFF COMMAND INADEQUATE CONTROL ACTION

Because we are looking at the hazard "Aircraft drops below Pre-Selected Altitude," this analysis focuses on entering and leaving the VAPPR. The Approach Switch also turns the Lateral Approach Mode on or off.

B.19.1 UNSAFE CONTROL ACTION: INADVERTENT VERTICAL APPROACH ON/OFF COMMAND SENT TO FGS

Some possible causes:

- Mode confusion, perhaps contributed to by the Vertical Approach button toggling the VAPPR on and off depending on current status.
- Pilot inadvertently presses Vertical Approach Switch.
- Lateral Approach mode is deactivated, Selected Navigation source is changed, Selected Navigation frequency is changed, pilot inadvertently activates Pilot Flying Transfer or turns Modes off.
- In addition, there are two macros that can both be true (Select_VAPPR and Deselect_VAPPR), which means that the Variable VAPPR could bounce back and forth between Cleared and Selected.

Results: Turning the VAPPR on unexpectedly will result in prematurely exiting another mode and could result in dropping below the current altitude, which may be the PSA. Turning it off unexpectedly could result in a GA at an unsafe altitude. It cannot activate during an overspeed condition.

Mitigation: Select_VAPPR and Deselect_VAPPR need to be mutually exclusive in the specification. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects actual rather than requested state. The APPR light currently appears to be based on actual rather than requested; however the APPR is internal to the FGS; although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design and will require a review of the design.

B.19.2 UNSAFE CONTROL ACTION: VERTICAL APPROACH ON/OFF COMMAND IS NOT PROVIDED OR MISSING

Some possible causes:

- Pilot presses Vertical Approach Switch to select VAPPR, but system does not register because button pressed is on inactive side.
- Pilot presses Vertical Approach Switch to select VAPPR, but system does not register because we are in an overspeed condition.
- Vertical Approach Switch fails.

Result: The system will remain in whatever mode it was in before VAPPR should have registered. Depending on the mode, this could result in the aircraft descending faster than it should or not descending when it should. If it is already in VAPPR mode, it will continue descent. The aircraft may travel down (below PSA) rather than maintain level or move up.

Mitigation: Vertical Approach Switch is intended to toggle VAPPR on and off. The macros Select_VAPPR and Deselect_VAPPR are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The APPR light currently appears to be based on the actual state rather than the requested state; however; the APPR is internal to the FGS and although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design. The Transfer Switch is also a toggle, so there must be clear indications to the pilot of which side is in control, even during conditions when the flight guidance commands are not able to be displayed on the active side.

B.19.3 UNSAFE CONTROL ACTION: VERTICAL APPROACH ON/OFF COMMAND RECEIVED AT WRONG TIME

Some possible causes:

- Processing delay.
- Mode confusion with respect to state of VAPPR or Pilot Flying.
- Pilot error.
- See section 8.19.2 for ways to exit VAPPR unexpectedly.

Result: Depending on the state of the modes, the system will enter VAPPR Mode when it should not, leaving the current necessary state and beginning a descent into nowhere, or the system will remain in VAPPR whether or not this is the mode the aircraft needs to be in. The aircraft may travel down (below PSA) rather than maintain level or move up.

Mitigation: Vertical Approach Switch is intended to toggle VAPPR on and off. The macros Select_VAPPR and Deselect_VAPPR are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The APPR light currently appears to be based on the actual state rather than the requested state; however, the APPR is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design. Signal-processing requirements should give minimum and maximum system response time to pilot input.

B.19.4 UNSAFE CONTROL ACTION: VERTICAL APPROACH ON/OFF COMMAND IS STOPPED TOO SOON

This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command. No scenarios were found for this inadequate control action to occur. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.20 CREW TO FGS: GA ON/OFF COMMAND INADEQUATE CONTROL ACTION

Because we are looking at the hazard "Aircraft drops below Pre-Selected Altitude," this analysis focuses on entering and leaving the VAPPR Mode. The GA Switch also turns the Lateral Approach Mode on or off.

B.20.1 UNSAFE CONTROL ACTION: INADVERTENT VGA ON/OFF COMMAND SENT TO FGS

Some possible causes:

- Mode confusion.
- Pilot inadvertently presses GA Switch.
- There are two macros that can both be true (Select_VGA and Deselect_VGA), which means that the Variable VGA could bounce back and forth between Cleared and Selected.

Results: Turning the VAPPR Mode on unexpectedly could result in a VAPPR at an unsafe altitude. Turning it on could also result in prematurely exiting another mode. It cannot activate during an overspeed condition. Turning it off inadvertently would result in the system entering either ALT or VS mode, potentially with bad data.

Mitigation: Select_VGA and Deselect_VGA need to be mutually exclusive in the specification. The pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The GA light currently appears to be based on the actual state rather than the requested state; however, the GA is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design. Requirements are needed specifying when data can be used and when they should be considered obsolete.

B.20.2 UNSAFE CONTROL ACTION: VGA ON/OFF COMMAND IS NOT PROVIDED OR MISSING

Some possible causes:

- Pilot presses GA Switch to select Vertical GA Mode, but system does not register because button pressed is on inactive side.
- Pilot presses GA Switch to select Vertical GA Mode, but system does not register because aircraft is in an overspeed condition.
- GA Switch fails.
- VAPPR is not activated because the Deselect VGA is true. Deselect VGA is true when non-Basic Lateral mode is activated, non-Basic Vertical mode is activated, pilot presses SYNC Switch, pilot rotates VS Pitch Wheel, or pilot inadvertently activates Pilot Flying Transfer or turns Modes off.

Result: Depending on the state of the modes, the system will remain in VAPPR Mode when it should not, resulting in the continuation of a hazardous descent.

Mitigation: The macros Select_VGA and Deselect_VGA are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The GA light currently appears to be based on the actual state rather than the requested state; however, the GA is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—thus requiring a review of the design.

B.20.3 UNSAFE CONTROL ACTION: VGA ON/OFF COMMAND RECEIVED AT WRONG TIME

Some possible causes:

- System delay
- Pilot error

Result: Depending on the state of the modes, system will enter VGA Mode when it should not, resulting in leaving the current optimal state and beginning an unnecessary GA. Because this was not intended, the current settings could be incorrect, resulting in an unsafe GA level being specified to the pilot.

Mitigation: The macros Select_VGA and Deselect_VGA are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The GA light currently appears to be based on the actual state rather than the requested state; however, the GA is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes

in the design—thus requiring a review of the design. Signal-processing requirements should give minimum and maximum system response time to pilot input.

B.20.4 UNSAFE CONTROL ACTION: VGA ON/OFF COMMAND IS STOPPED TOO SOON

This scenario looks at mitigating ways in which a command, continuous or discrete, could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command, which decreases the chance of it being stopped too soon. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.21 CREW TO FGS: AP ENGAGE COMMAND INADEQUATE CONTROL ACTION

B.21.1 UNSAFE CONTROL ACTION: INADVERTENT AP ENGAGE COMMAND TO FGS

Some possible causes:

- AP Engage button pushed inadvertently.
- Mode confusion (e.g. regarding pilot flying).

Result: AP may execute obsolete pitch and roll commands.

Mitigation: A two-step activation process (e.g., button press and hold for three seconds) or limit course correction should be considered. Requirements limiting course correction immediately after the AP is turned on might also be used to mitigate this hazard. At a minimum, requirements are needed specifying when flight data and commands are obsolete. Although there is an AP On lamp for the panel, it is currently based on whether the FGS received a command to turn the AP on, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback.

B.21.2 UNSAFE CONTROL ACTION: AP ENGAGE COMMAND IS NOT PROVIDED/MISSING

Some possible causes:

- Mode confusion.
- Mechanical failure.
- AP engage command is not registered because aircraft is in GA mode.

Result: Neither pilot nor AP send commands to controlled surfaces.

Mitigation: The AP lamp command is based on whether the FGS thinks the AP should be turned on rather than the actual state of the AP. Basing the lamp on feedback from the AP could help mitigate this scenario.

B.21.3 UNSAFE CONTROL ACTION: AP ENGAGE COMMAND IS SENT AT WRONG TIME

Some possible causes:

- Processing delayed.
- Pilot Transfer or system reset results in obsolete commands being processed.

Results: AP fails to command controlled surfaces when the pilot commands and starts commanding at another unexpected time.

Mitigation: The AP lamp command is based on whether the FGS thinks the AP should be turned on rather than the actual state of the AP. Basing the lamp on feedback from the AP could help mitigate this scenario.

B.21.4 UNSAFE CONTROL ACTION: AP ENGAGE COMMAND IS STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking at mitigating ways in which a command, continuous or discrete, could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify the type of signal/command the AP Engage sends to the flight director and what will be done to prevent partially delivered messages.

B.22 CREW TO FGS: AP DISENGAGE COMMAND INADEQUATE CONTROL ACTION

B.22.1 UNSAFE CONTROL ACTION: INADVERTENT AP DISENGAGE COMMAND SENT TO AP

Some possible causes:

- AP Disengage button is pushed inadvertently.
- Mode confusion (e.g. regarding pilot flying).
- GA mode button is pressed inadvertently.

Result: Neither the AP nor the pilot is commanding controlled surfaces.

Mitigation: Although there is an AP On lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback. Requirements are also needed that will prevent both the AP and the pilot from commanding controlled surfaces.

B.22.2 UNSAFE CONTROL ACTION: AP DISENGAGE COMMAND IS NOT PROVIDED/MISSING

Some possible causes:

- Mode confusion.
- Mechanical failure.
- Button pressed on inactive side.
- Pilot Flying Transfer is in process.
- Both AP Engage and AP disengage are pressed on the active side.

Result: Both AP and pilot are commanding controlled surfaces.

Mitigation: Although there is an AP On lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback. AP Engage and AP Disengage need to be prioritized. Requirements are needed that will prevent both the AP and the pilot from commanding controlled surfaces. Requirements are also needed on the AP to define behavior while transferring, during initialization, and during restart.

B.22.3 UNSAFE CONTROL ACTION: AP DISENGAGE COMMAND IS SENT AT THE WRONG TIME

Some possible causes:

- Another command is processed before AP Disengage.
- System is restarting.
- AP Engage is pushed instead of AP Disengage button (the buttons are next to each other).

Result: AP remains on when it should be off, and pilot and AP both attempt to command controlled surfaces. Then, AP unexpectedly is turned off while the aircraft is still flying and descends below the PSA.

Mitigation: Although there is an AP On lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback. AP Engage and AP Disengage need to be prioritized. Requirements are needed that will prevent both the AP and the pilot from commanding controlled surfaces. Requirements are also needed on the AP to define behavior while transferring, during initialization, and during restart.

B.22.4 UNSAFE CONTROL ACTION: AP DISENGAGE COMMAND IS STOPPED TOO SOON

Result: The requirements do not specify what would happen to a partially delivered command.

Mitigation: This scenario is looking at mitigating ways in which a command, continuous or discrete, could be interrupted. From the requirements, it is unclear if this command is a one-time command or whether the signal is sent continuously. The requirements should specify the type of signal/command the VS data send to the flight director and what will be done to prevent partially delivered messages.

B.23 CREW TO FGS: SYNC INADEQUATE CONTROL ACTION

As development progresses and more information on SYNC becomes available, this analysis should be re-examined.

B.23.1 UNSAFE CONTROL ACTION: INADVERTENT SYNC COMMAND RECEIVED

Results: The SYNC switch is at the top of the prioritization list. Only commands to turn the AP on or off can be processed if the SYNC button is registered. Therefore, it could result in any of the "Command Not Provided" results. If the button gets stuck in the depressed state, it could result in the pilots being unable to change any of the flight modes. At the current stage of development, it is not clear what the intended results of pressing the SYNC button are; however, it is assumed that it would usually be used during independent modes, such as GA and Approach. If so, inadvertent commanding of SYNC might result in inaccurate data being copied into both sides of the FGS.

Possible Mitigation plan: Ensure that there is clear notification to pilots of which side is in command. This could prove more of an issue if one of the Flight Displays is turned off or not working. In addition, requirements should be considered to evaluate the correctness and reasonableness of this input.

B.23.2 UNSAFE CONTROL ACTION: SYNC COMMAND IS NOT PROVIDED OR MISSING

Results: Data are not transferred to the inactive side. A transfer in Pilot Flying after this might result in a sudden course correction.

Possible mitigation plan: Determine a margin of difference that is acceptable. Ensure clear notification of the pilots if the two sides differ by more than that acceptable margin. Implement limits on course-correction commands sent to the AP.

B.23.3 UNSAFE CONTROL ACTION: SYNC COMMAND IS SENT AT THE WRONG TIME

Results: If the SYNC message is delayed until after a Pilot Transfer occurs, the wrong data will be transferred. Result will be similar to inadvertent or not provided.

Possible mitigation plan: Determine a margin of difference that is acceptable. Ensure clear notification of the pilots if the two sides differ by more than that acceptable margin. Implement limits on course correction commands sent to the AP. Ensure there is clear notification to pilots of which side is in command. This could prove more of an issue if one of the Flight Displays is turned off or not working. Requirements should be considered to evaluate the correctness and reasonableness of this input. Signal-processing requirements should give minimum and maximum system response time to pilot input.

B.23.4 UNSAFE CONTROL ACTION: SYNC COMMAND IS STOPPED TOO SOON

This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command. No scenarios were found for this inadequate control action to occur. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.24 CREW TO FGS: PILOT TRANSFER COMMAND INADEQUATE CONTROL ACTION

As development progresses and more information on Pilot Transfer becomes available, this analysis should be re-examined.

B.24.1 UNSAFE CONTROL ACTION: INADVERTENT PILOT TRANSFER COMMAND RECEIVED

Results: The Pilot Transfer switch is low on the prioritization list. However, it could still cause the following events to be ignored: VAPPR Capture Condition, LAPPR Capture Condition, NAV Capture Condition, ALTSEL Capture Condition, and ALTSEL Track Condition. In addition, transferring inadvertently could result in intended commands being ignored and other unintended commands being acted on. If the switch somehow remains stuck, it could result in a rapidly fluctuating change of which side is in command, pilot commands being ignored, and erratic behavior.

Possible mitigation plan: Ensure that there is clear notification to pilots of which side is in command. Care must be taken that this reflects the actual state rather than the requested state. The Pilot Flying arrow currently appears to be based on the actual state rather than the requested state; however, the Pilot Flying is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design. In addition, requirements should be considered to evaluate the correctness and reasonableness of the input.

B.24.2 UNSAFE CONTROL ACTION: PILOT TRANSFER COMMAND IS NOT PROVIDED OR MISSING

Results: Intended transfer does not occur and commands from the intended non-pilot flying side are acted on.

Possible mitigation plan: Ensure that there is clear notification to pilots of which side is in command. Care must be taken that this reflects the actual state rather than the requested state. The Pilot Flying arrow currently appears to be based on the actual state rather than the requested state; however, the Pilot Flying is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design.

B.24.3 UNSAFE CONTROL ACTION: PILOT TRANSFER COMMAND IS SENT AT THE WRONG TIME

Results: Intended transfer does not occur when intended, and commands continue to be processed from the originally active side. If the transfer occurred because of an error on this side, the results could be hazardous. When the signal is finally recorded, mode confusion may result as the system acts on a transfer, which is no longer intended.

Possible mitigation plan: Requirements should be considered to evaluate the correctness and reasonableness of this input. Signal-processing requirements should give minimum and maximum system response time to pilot input. Ensure that there is clear notification to pilots of which side is in command. Care must be taken that this reflects the actual state rather than the requested state. The Pilot Flying arrow currently appears to be based on the actual state rather than the requested state; however, the Pilot Flying is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design.

B.24.4 UNSAFE CONTROL ACTION: PILOT TRANSFER COMMAND IS STOPPED TOO SOON

This scenario is looking for ways in which a continuous data stream or command could be interrupted. From the requirements, this command is a one-time button push command and the signal is not sending a continuous command. No scenarios were found for this inadequate control action to occur. Although there is no way in the requirements that this could occur, care should be taken in the design to prevent partially delivered messages.

B.25 CREW TO FGS: NAV ON/OFF COMMAND INADEQUATE CONTROL ACTION

B.26 CREW TO FGS: HDG COMMAND INADEQUATE

The heading command may indirectly affect the altitude. There are currently no requirements correcting for pitch during a roll maneuver. It is unknown if this would be handled by the FGS or the AP. This command will need to be re-examined as the requirements mature.

B.27 CREW TO FGS: FLIGHT DIRECTOR ON/OFF COMMAND INADEQUATE CONTROL ACTION

The specification did not provide enough information on the Flight Director On/Off command to perform the analysis on the interaction between Crew and FGS.

B.28 FMS TO FGS: FLIGHT DATA INADEQUATE CONTROL ACTION

It is unclear at this time what the Flight Data include. This analysis is based on it being pre-selected waypoints and flight levels. As the requirements mature, this analysis will need to be revisited (see figure B-5).

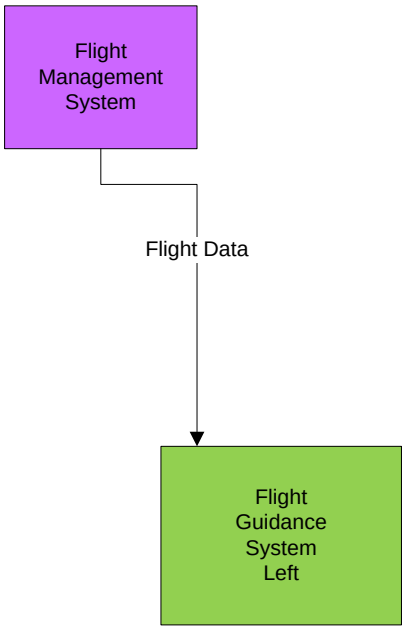


Figure B-5. FGS left interactions with FMS

B.28.1 UNSAFE CONTROL ACTION: INADVERTENT FLIGHT DATA RECEIVED

Some possible causes:

- FGS restart.
- Requirements for the FMS are not part of this analysis.

Results: FGS sends incorrect commands to pilot and AP, resulting in the aircraft flying to an unintended altitude.

Possible mitigation plan: If it is intended that the FGS receive flight data from the FMS during flight, all data received should be time stamped. The pilot's role in approving course corrections

subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined.

B.28.2 UNSAFE CONTROL ACTION: FLIGHT DATA ARE NOT PROVIDED OR ARE MISSING

Some possible causes:

- FMS is unavailable.
- FGS restart.

Results: FGS does not send commands or sends incorrect commands to the pilot and AP, resulting in the aircraft flying to an unintended altitude.

Possible mitigation plan: If it is intended that the FGS receive flight data from the FMS, all data received should be time stamped. The pilot's role in approving course corrections subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined. In addition, the behavior of the FGS when other systems are unavailable or not responding should be defined.

B.28.3 UNSAFE CONTROL ACTION: FLIGHT DATA ARE SENT AT THE WRONG TIME

Some possible causes:

- FGS restart.
- Requirements for the FMS are not part of this analysis.

Results: FGS sends incorrect commands to pilot and AP, resulting in the aircraft flying to an unintended altitude.

Possible mitigation plan: If it is intended that the FGS receive flight data from the FMS, all data received should be time stamped. The pilot's role in approving course corrections subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined.

B.28.4 UNSAFE CONTROL ACTION: FLIGHT DATA ARE STOPPED TOO SOON

Some possible causes:

- FMS is unavailable.
- FGS restart.

Results: FGS does not send commands or sends incorrect commands to pilot and AP, resulting in the aircraft flying to an unintended altitude.

Possible mitigation plan: If it is intended that the FGS receive flight data from the FMS and that all data received should be time stamped. The pilot's role in approving course corrections

subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined. In addition, the behavior of the FGS when other systems are unavailable or not responding should be defined. How the system will respond to partial messages also needs to be defined.

B.29 AIR DATA TO FGS: IAS INADEQUATE CONTROL ACTION

B.29.1 UNSAFE CONTROL ACTION: INADVERTENT IAS DATA RECEIVED

Results: If provided when not expected, the feedback might interfere with the processing of the FGS (see figure B-6).

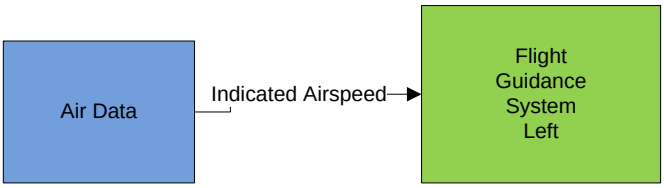


Figure B-6. FGS left interactions with air data

Possible mitigation plan: Requirements need to provide clear priorities of when to process IAS. The requirements should state how long, for example, an overspeed condition can exist before it is registered.

B.29.2 UNSAFE CONTROL ACTION: IAS DATA ARE NOT PROVIDED OR ARE MISSING

Results: If an overspeed condition exists, it will not be registered and the system may proceed into modes normally restricted during overspeed conditions. These modes include VS mode, VAPPR, and VAPPR. It could also fail to enter Flight Level Change mode or exit Flight Level Change mode prematurely.

Possible mitigation plan: Requirements are needed to ensure an overspeed condition will be registered.

B.29.3 UNSAFE CONTROL ACTION: IAS DATA ARE SENT AT THE WRONG TIME

Results: An overspeed condition may not be registered in a timely manner or be registered after the problem has been corrected. Pilots may adjust the throttle to compensate.

Mitigation: Requirements must be provided to determine if the IAS is reasonable, and to address when data become obsolete and how the FGS should respond.

B.29.4 UNSAFE CONTROL ACTION: IAS DATA ARE STOPPED TOO SOON

Results: It is unclear based on the specification whether the system will expect continuous or discrete IAS data, the frequency with which it will receive the data, or what it would mean to the system for the data to stop too soon.

Possible mitigation plan: Requirements need to be added that handle situations for which the IAS becomes obsolete.

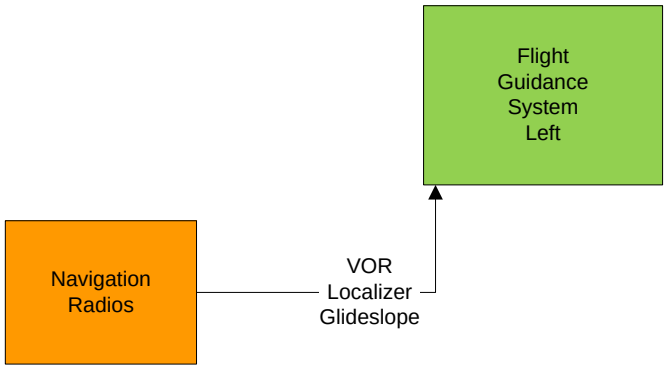


Figure B-7. FGS left interactions with navigation radios

B.30 NAVIGATION RADIOS TO FGS: NAVIGATION RADIO DATA (VOR/LOCALIZER/GLIDESLOPE) INADEQUATE CONTROL ACTION

B.30.1 UNSAFE CONTROL ACTION: INADVERTENT NAVIGATION RADIO DATA RECEIVED

Results: Selected navigation source or frequency could be marked as changed, resulting in the system leaving VAPPR.

Mitigation: Requirements must be provided to determine if the navigation radio data are reasonable. Care must be taken during the latter stages of approach to ensure that clearing Vertical Approach is intended and reasonable while not creating an unnecessary delay in acting during an emergency.

B.30.2 UNSAFE CONTROL ACTION: NAVIGATION RADIO DATA ARE NOT PROVIDED OR MISSING

Results: It is undefined, but the system may continue to act on obsolete data.

Possible mitigation plan: Additional requirements are needed to address when data become obsolete and how the FGS should respond.

B.30.3 UNSAFE CONTROL ACTION: NAVIGATION RADIO DATA ARE SENT AT THE WRONG TIME

Results: Receiving the navigation radio data too early could lead to mode confusion and result in the pilot changing to Approach Mode too early. The consequences of this dynamic could include dropping below the PSA. Receiving the navigation radio data late could lead to using data from the wrong airport or having to perform a GA.

Possible mitigation plan: Requirements must be provided to determine if the navigation radio data are reasonable. Care must be taken during the latter stages of approach to ensure that clearing Vertical Approach is intended and reasonable while not creating an unnecessary delay in acting during an emergency. Additional requirements are needed to address when data become obsolete and how the FGS should respond.

B.30.4 UNSAFE CONTROL ACTION: NAVIGATION RADIO DATA ARE STOPPED TOO SOON

Results: It is unclear, based on the specification, whether the system will expect continuous or discrete navigation radio data, the frequency with which it will receive the data, or what it would mean to the system for the data to stop too soon.

Possible mitigation plan: Requirements must be provided to determine if the navigation radio data are reasonable. Care must be taken during the latter stages of approach to ensure that clearing Vertical Approach is intended and reasonable, but does not create an unnecessary delay in acting during an emergency. Additional requirements are needed to address when data become obsolete and how the FGS should respond.

The findings for step 3 of the process of listing inadequate control actions are detailed in sections B.1, B.3, and B.6–B.30. B.31 is a summary of all the findings.

B.31 SUMMARY FINDINGS OF THE STPA ANALYSIS OF FGS

FGS: Requirements are needed to specify how commands from the off-side FGS are handled. Requirements also needed to limit when the GA mode can be triggered in the FGS software. Requirements are needed to specify how the Pilot Transfer will occur and if any other modes or components will be affected in the FGS software. Requirements are needed to specify the FGS prioritization between the pilot and the AP commands. These requirements should include whether to leave the AP on if the throttle or yoke are active. Requirements do specify feedback to pilot as to which FGS is in control, but requirement needs to specify what happens during transitioning. Feedback to pilot could help to mitigate this scenario. Requirements specify an AP lamp, but it is not based on a Health and Status message from the AP. The AP lamp should be based on feedback from the AP rather than the requested state of the AP. Requirements are also needed to specify what happens if the FGS loses power or is restarted. Requirements are need to specify how the pitch values are checked for correctness.

Although there are requirements that specify the triggering of GA mode, it is unclear what the FGS software does with the GA command. More details need to be specified as to how the software handles the GA input command from the FCP.

Vertical Speed (VS) Data: All commands and data should have expected rates and obsolescence defined. Components receiving data need to have clear requirements on how to handle data received at the wrong time. The requirements should specify how the system behaves during power loss, restart, and initialization. The requirements should specify how the software processes the flight director on/off commands in regard to the VS data. The requirements should specify what prioritization the Flight Director (FD) on/off command should have, reducing the possibility of the command being permanently delayed.

Vertical Speed (VS): The VS Switch is intended to toggle VS on and off. Select_VS and Deselect_VS need to be mutually exclusive in the specification. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The VS light currently appears to be based on the actual state rather than the requested state; however, the VS is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design. Requirements need to specify response times to pilot inputs.

Pitch Wheel: Requirements need to specify how intended Pitch Wheel motions can be separated from inadvertent motions. At a minimum, changes in mode should be highlighted in such a way as to catch the pilot's attention, such as blinking until confirmed. Requirements are needed to either limit the amount of Pitch Wheel change allowed in a given time or at least to make sure the pilot can tell that the wheel has been moved. Specify maximum response times to pilot inputs and obsolescence for all data and feedback. Provide feedback for all pilot inputs.

Flight Level Change Mode: Flight Level Change Switch is intended to toggle Flight Level Change Mode on and off. The macros Select_FLC and Deselect_FLC are not mutually exclusive and need to be corrected because both could be true at the same time. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The FLC light currently appears to be based on the actual state rather than the requested state; however, the FLC is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design.

ALT: ALT switch is intended to toggle ALT mode on and off. The macros Select_ALT and Deselect_ALT are not mutually exclusive and need to be corrected because both could be true at the same time. Pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The Alt light currently appears to be based on the actual state rather than the requested state; however, the ALT is internal to the FGS and although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design.

Signal-processing requirements should give minimum and maximum system response time to pilot input.

Vertical approach mode (VAPPR): Vertical Approach Switch is intended to toggle VAPPR on and off. The macros Select_VAPPR and Deselect_VAPPR are not mutually exclusive and need to be corrected because both could be true at the same time. Because of the potential of confusion with the same input doing two different things, it is crucial that the pilot be provided with feedback indicating the current state. Care must be taken that this reflects the actual state rather than the requested state. The APPR light currently appears to be based on the actual state rather than the requested state; however, the APPR is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design. Signal-processing requirements should give minimum and maximum system response time to pilot input.

Transfer: Requirements should be considered to evaluate the correctness and reasonableness of the transfer input. Signal-processing requirements should give minimum and maximum system response time to pilot input. Ensure that there is clear notification to pilots of which side is in command. Care must be taken that this reflects the actual state rather than the requested state. The Pilot Flying arrow currently appears to be based on the actual state rather than the requested state; however, the Pilot Flying is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design.

Vertical Go Around (VGA): Select_VGA and Deselect_VGA need to be mutually exclusive in the specification. The pilot must be provided with feedback indicating the current state. Care must be taken that this reflects the actual rather than the requested state. The GA light currently appears to be based on the actual state rather than the requested state; however the GA light is internal to the FGS and, although it appears that the intent is to display the actual state, the specification does not make this clear enough to prevent mistakes in the design—requiring a review of the design. Requirements are needed specifying when data can be used and when they should be considered obsolete. Signal-processing requirements should give minimum and maximum system response time to pilot input.

Autopilot (AP): Although there is an AP On lamp for the panel, it is currently based on whether the FGS received a command to turn the AP On, not on the actual state of the AP. Feedback from the AP to the FGS is needed and the lamp should be based on this feedback. AP Engage and AP Disengage need to be prioritized. Requirements are needed that will prevent both the AP and the pilot from commanding controlled surfaces. Requirements are also needed for the AP to define behavior while transferring, during initialization, and during re-start.

SYNC: Determine a margin of difference that is acceptable; ensure clear notification to the pilots if the data displayed on the FDs differ by more than that acceptable margin. Implement limits on course correction commands sent to the AP. Ensure that there is clear notification to pilots of which side is in command. This could prove more of an issue if one of the Flight Displays is turned off or not working. Requirements should be considered to evaluate the correctness and

reasonableness of this input. Signal-processing requirements should give minimum and maximum system response time to pilot input.

Indicated Airspeed (IAS): Requirements need to provide clear priorities of when to process IAS. The requirements should state how long, for example, an overspeed condition can exist before it is registered. Requirements are needed to ensure an overspeed condition will be registered. Requirements must be provided to determine if the IAS is reasonable and to address when data become obsolete and how the FGS should respond. Requirements need to be added that handle the IAS becoming obsolete.

Flight Management System (FMS): It is unclear at this time what information is sent from the FMS to the FGS. This analysis assumes the flight data include pre-selected waypoints and flight levels. As the requirements mature, this analysis will need to be revisited. If it is intended that the FGS receive flight data from the FMS during flight; all data received should be time stamped. The pilot's role in approving course corrections subsequent to receipt of data should be analyzed. Power off, restart, and initialization need to be defined. How the system will respond to partial messages also needs to be defined.

All Input and Output Commands: Requirements are needed to specify what to do if a partial command is sent or received. Requirements must be in place that specify what to do if an AP On command arrives before the system is finished implementing the AP Off command. At a minimum, feedback to the pilot must reflect the actual state of the system.

Feedback from AP: At this point, Health and Status Feedback from the AP to the FGS is needed but not provided. It is unknown what form that feedback will take. Some common forms would be a query from the FGS to the AP, a continuous or discrete signal from the AP when it is on, a continuous or discrete signal from the AP indicating whether it is on or off, and regular messages from the AP to the FGS stating what the AP is doing.

B.32. REFERENCES

B.1. Staats, M., Gay, G., Whalen, M.W., and Heimdahl, M.P., "On the Danger of Coverage Directed Test Case Generation," in Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE), Tallinn, Estonia, April 2012.