



CRC Press  
Taylor & Francis Group

# DEVELOPING SAFETY-CRITICAL SOFTWARE

A Practical Guide for Aviation Software and DO-178C Compliance

LEANNA RIERSON



# Developing Safety-Critical Software

# Developing Safety-Critical Software

**A Practical Guide for Aviation Software and DO-178C Compliance**

**LEANNA RIERSON**



---

CRC Press is an imprint of the  
Taylor & Francis Group, an **informa** business

MATLAB<sup>®</sup> and Simulink<sup>®</sup> are trademarks of The MathWorks, Inc. and are used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB<sup>®</sup> and Simulink<sup>®</sup> software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB<sup>®</sup> and Simulink<sup>®</sup> software.

CRC Press  
Taylor & Francis Group  
6000 Broken Sound Parkway NW, Suite 300  
Boca Raton, FL 33487-2742

*CRC Press is an imprint of Taylor & Francis Group, an Informa business*

No claim to original U.S. Government works

Version Date: 20121016

International Standard Book Number: 978-1-4398-1368-3 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access [www.copyright.com](http://www.copyright.com) (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

**Visit the Taylor & Francis Web site at**  
**<http://www.taylorandfrancis.com>**

**and the CRC Press Web site at**  
**<http://www.crcpress.com>**

*I dedicate this book in memory of Cary Spitzer, who believed in its importance and my ability to write it, and to my grandmother, Charlotte Richardson, who prayed daily for my work. Both Cary and Grandma Richardson passed away in the fall of 2011 as I was finishing the first draft of this book. I miss them both immensely and hope they would be pleased with the text that they helped inspire.*

# Contents

## Part I Introduction

### 1 Introduction and Overview

#### Acronyms

#### 1.1 Defining Safety-Critical Software

#### 1.2 Importance of Safety Focus

#### 1.3 Book Purpose and Important Caveats

#### 1.4 Book Overview

#### References

## Part II Context of Safety-Critical Software Development

### 2 Software in the Context of the System

#### Acronyms

#### 2.1 Overview of System Development

#### 2.2 System Requirements

##### 2.2.1 Importance of System Requirements

##### 2.2.2 Types of System Requirements

##### 2.2.3 Characteristics of Good Requirements

##### 2.2.4 System Requirements Considerations

##### 2.2.4.1 Integrity and Availability Considerations

##### 2.2.4.2 Other System Requirements Considerations

##### 2.2.5 Requirements Assumptions

##### 2.2.6 Allocation to Items

- 2.3 System Requirements Validation and Verification
  - 2.3.1 Requirements Validation
  - 2.3.2 Implementation Verification
  - 2.3.3 Validation and Verification Recommendations
- 2.4 Best Practices for Systems Engineers
- 2.5 Software's Relationship to the System
- References
- 3 Software in the Context of the System Safety Assessment
  - Acronyms
  - 3.1 Overview of the Aircraft and System Safety Assessment Process
    - 3.1.1 Safety Program Plan
    - 3.1.2 Functional Hazard Assessment
    - 3.1.3 System Functional Hazard Assessment
    - 3.1.4 Preliminary Aircraft Safety Assessment
    - 3.1.5 Preliminary System Safety Assessment
    - 3.1.6 Common Cause Analysis
    - 3.1.7 Aircraft and System Safety Assessments
  - 3.2 Development Assurance
    - 3.2.1 Development Assurance Levels
  - 3.3 How Does Software Fit into the Safety Process?
    - 3.3.1 Software's Uniqueness
    - 3.3.2 Software Development Assurance
    - 3.3.3 Other Views
    - 3.3.4 Some Suggestions for Addressing Software in the System Safety Process
  - References

## Part III Developing Safety-Critical Software Using DO-178C

- 4 Overview of DO-178C and Supporting Documents
  - Acronyms
  - 4.1 History of DO-178

- 4.2 DO-178C and DO-278A Core Documents
  - 4.2.1 DO-278A and DO-178C Differences
  - 4.2.2 Overview of the DO-178C Annex A Objectives Tables
- 4.3 DO-330: Software Tool Qualification Considerations
- 4.4 DO-178C Technology Supplements
  - 4.4.1 DO-331: Model-Based Development Supplement
  - 4.4.2 DO-332: Object-Oriented Technology Supplement
  - 4.4.3 DO-333: Formal Methods Supplement
- 4.5 DO-248C: Supporting Material
- References
- 5 Software Planning
  - Acronyms
  - 5.1 Introduction
  - 5.2 General Planning Recommendations
  - 5.3 Five Software Plans
    - 5.3.1 Plan for Software Aspects of Certification
    - 5.3.2 Software Development Plan
    - 5.3.3 Software Verification Plan
    - 5.3.4 Software Configuration Management Plan
    - 5.3.5 Software Quality Assurance Plan
  - 5.4 Three Development Standards
    - 5.4.1 Software Requirements Standards
    - 5.4.2 Software Design Standards
    - 5.4.3 Software Coding Standards
  - 5.5 Tool Qualification Planning
  - 5.6 Other Plans
    - 5.6.1 Project Management Plan
    - 5.6.2 Requirements Management Plan
    - 5.6.3 Test Plan
  - References
- 6 Software Requirements
  - Acronyms

- 6.1 Introduction
- 6.2 Defining Requirement
- 6.3 Importance of Good Software Requirements
  - 6.3.1 Reason 1: Requirements Are the Foundation for the Software Development
  - 6.3.2 Reason 2: Good Requirements Save Time and Money
  - 6.3.3 Reason 3: Good Requirements Are Essential to Safety
  - 6.3.4 Reason 4: Good Requirements Are Necessary to Meet the Customer Needs
  - 6.3.5 Reason 5: Good Requirements Are Important for Testing
- 6.4 The Software Requirements Engineer
- 6.5 Overview of Software Requirements Development
- 6.6 Gathering and Analyzing Input to the Software Requirements
  - 6.6.1 Requirements Gathering Activities
  - 6.6.2 Requirements Analyzing Activities
- 6.7 Writing the Software Requirements
  - 6.7.1 Task 1: Determine the Methodology
  - 6.7.2 Task 2: Determine the Software Requirements Document Layout
  - 6.7.3 Task 3: Divide Software Functionality into Subsystems and/or Features
  - 6.7.4 Task 4: Determine Requirements Priorities
  - 6.7.5 A Brief Detour (Not a Task): Slippery Slopes to Avoid
    - 6.7.5.1 Slippery Slope #1: Going to Design Too Quickly
    - 6.7.5.2 Slippery Slope #2: One Level of Requirements
    - 6.7.5.3 Slippery Slope #3: Going Straight to Code
  - 6.7.6 Task 5: Document the Requirements
    - 6.7.6.1 Document Functional Requirements
    - 6.7.6.2 Document Nonfunctional Requirements
    - 6.7.6.3 Document Interfaces
    - 6.7.6.4 Uniquely Identify Each Requirement

- 6.7.6.5 Document Rationale
- 6.7.6.6 Trace Requirements to Their Source
- 6.7.6.7 Identify Uncertainties and Assumptions
- 6.7.6.8 Start a Data Dictionary
- 6.7.6.9 Implement Characteristics of Good Requirements
- 6.7.7 Task 6: Provide Feedback on the System Requirements
- 6.8 Verifying (Reviewing) Requirements
  - 6.8.1 Peer Review Recommended Practices
- 6.9 Managing Requirements
  - 6.9.1 Basics of Requirements Management
  - 6.9.2 Requirements Management Tools
- 6.10 Requirements Prototyping
- 6.11 Traceability
  - 6.11.1 Importance and Benefits of Traceability
  - 6.11.2 Bidirectional Traceability
  - 6.11.3 DO-178C and Traceability
  - 6.11.4 Traceability Challenges
- References
- Recommended Readings
- 7 Software Design
  - Acronyms
  - 7.1 Overview of Software Design
    - 7.1.1 Software Architecture
    - 7.1.2 Software Low-Level Requirements
    - 7.1.3 Design Packaging
  - 7.2 Approaches to Design
    - 7.2.1 Structure-Based Design (Traditional)
    - 7.2.2 Object-Oriented Design
  - 7.3 Characteristics of Good Design
  - 7.4 Design Verification
  - References

## 8 Software Implementation: Coding and Integration

### Acronyms

### 8.1 Introduction

### 8.2 Coding

#### 8.2.1 Overview of DO-178C Coding Guidance

#### 8.2.2 Languages Used in Safety-Critical Software

##### 8.2.2.1 Assembly Language

##### 8.2.2.2 Ada

##### 8.2.2.3 C

#### 8.2.3 Choosing a Language and Compiler

#### 8.2.4 General Recommendations for Programming

#### 8.2.5 Special Code-Related Topics

##### 8.2.5.1 Coding Standards

##### 8.2.5.2 Compiler-Supplied Libraries

##### 8.2.5.3 Autocode Generators

### 8.3 Verifying the Source Code

### 8.4 Development Integration

#### 8.4.1 Build Process

#### 8.4.2 Load Process

### 8.5 Verifying the Development Integration

### References

### Recommended Reading

## 9 Software Verification

### Acronyms

### 9.1 Introduction

### 9.2 Importance of Verification

### 9.3 Independence and Verification

### 9.4 Reviews

#### 9.4.1 Software Planning Review

#### 9.4.2 Software Requirements, Design, and Code Reviews

#### 9.4.3 Test Data Reviews

#### 9.4.4 Review of Other Data Items

## 9.5 Analyses

### 9.5.1 Worst-Case Execution Time Analysis

### 9.5.2 Memory Margin Analysis

### 9.5.3 Link and Memory Map Analysis

### 9.5.4 Load Analysis

### 9.5.5 Interrupt Analysis

### 9.5.6 Math Analysis

### 9.5.7 Errors and Warnings Analysis

### 9.5.8 Partitioning Analysis

## 9.6 Software Testing

### 9.6.1 Purpose of Software Testing

### 9.6.2 Overview of DO-178C's Software Testing Guidance

#### 9.6.2.1 Requirements-Based Test Methods

#### 9.6.2.2 Normal and Robustness Tests

### 9.6.3 Survey of Testing Strategies

#### 9.6.3.1 Equivalence Class Partitioning

#### 9.6.3.2 Boundary Value Testing

#### 9.6.3.3 State Transition Testing

#### 9.6.3.4 Decision Table Testing

#### 9.6.3.5 Integration Testing

#### 9.6.3.6 Performance Testing

#### 9.6.3.7 Other Strategies

#### 9.6.3.8 Complexity Measurements

#### 9.6.3.9 Summary and Characteristics of a Good Test

### 9.6.4 Test Planning

### 9.6.5 Test Development

#### 9.6.5.1 Test Cases

#### 9.6.5.2 Test Procedures

#### 9.6.5.3 DO-178C Requirements

#### 9.6.5.4 Low-Level Requirements Testing versus Unit Testing

#### 9.6.5.5 Handling Requirements That Cannot Be Tested

- 9.6.5.6 Obtaining Credit for Multiple Levels of Testing
- 9.6.5.7 Testing Additional Levels of Requirements
- 9.6.6 Test Execution
  - 9.6.6.1 Performing Dry Runs
  - 9.6.6.2 Reviewing Test Cases and Procedures
  - 9.6.6.3 Using Target Computer versus Emulator or Simulator
  - 9.6.6.4 Documenting the Verification Environment
  - 9.6.6.5 Test Readiness Review
  - 9.6.6.6 Running Tests for Certification Credit
- 9.6.7 Test Reporting
- 9.6.8 Test Traceability
- 9.6.9 Regression Testing
- 9.6.10 Testability
- 9.6.11 Automation in the Verification Processes
- 9.7 Verification of Verification
  - 9.7.1 Review of Test Procedures
  - 9.7.2 Review of Test Results
  - 9.7.3 Requirements Coverage Analysis
  - 9.7.4 Structural Coverage Analysis
    - 9.7.4.1 Statement Coverage (DO-178C Table A-7 Objective 7)
    - 9.7.4.2 Decision Coverage (DO-178C Table A-7 Objective 6)
    - 9.7.4.3 Modified Condition/Decision Coverage (DO-178C Table A-7 Objective 5)
    - 9.7.4.4 Additional Code Verification (DO-178C Table A-7 Objective 9)
    - 9.7.4.5 Data Coupling and Control Coupling Analyses (DO-178C Table A-7 Objective 8)
    - 9.7.4.6 Addressing Structural Coverage Gaps
    - 9.7.4.7 Final Thoughts on Structural Coverage Analysis
- 9.8 Problem Reporting

## 9.9 Recommendations for the Verification Processes

### References

### Recommended Readings

## 10 Software Configuration Management

### Acronyms

### 10.1 Introduction

#### 10.1.2 Why Is Software Configuration Management Needed?

#### 10.1.3 Who Is Responsible for Implementing Software Configuration Management?

#### 10.1.4 What Does Software Configuration Management Involve?

### 10.2 SCM Activities

#### 10.2.1 Configuration Identification

#### 10.2.2 Baselines

#### 10.2.3 Traceability

#### 10.2.4 Problem Reporting

##### 10.2.4.1 Problem Report Management with Multiple Stakeholders

##### 10.2.4.2 Managing Open/Deferred Problem Reports

#### 10.2.5 Change Control and Review

#### 10.2.6 Configuration Status Accounting

#### 10.2.7 Release

#### 10.2.8 Archival and Retrieval

#### 10.2.9 Data Control Categories

#### 10.2.10 Load Control

#### 10.2.11 Software Life Cycle Environment Control

### 10.3 Special SCM Skills

### 10.4 SCM Data

#### 10.4.1 SCM Plan

#### 10.4.2 Problem Reports

- 10.4.3 Software Life Cycle Environment Configuration Index
- 10.4.4 Software Configuration Index
- 10.4.5 SCM Records
- 10.5 SCM Pitfalls
- 10.6 Change Impact Analysis
- References
- 11 Software Quality Assurance
  - Acronyms
  - 11.1 Introduction: Software Quality and Software Quality Assurance (SQA)
    - 11.1.1 Defining Software Quality
    - 11.1.2 Characteristics of High-Quality Software
    - 11.1.3 Software Quality Assurance
    - 11.1.4 Examples of Common Quality Process and Product Issues
  - 11.2 Characteristics of Effective and Ineffective SQA
    - 11.2.1 Effective SQA
    - 11.2.2 Ineffective SQA
  - 11.3 SQA Activities
  - References
- 12 Certification Liaison
  - Acronyms
  - 12.1 What Is Certification Liaison?
  - 12.2 Communicating with the Certification Authorities
    - 12.2.1 Best Practices for Coordinating with Certification Authorities
  - 12.3 Software Accomplishment Summary
  - 12.4 Stage of Involvement (SOI) Audits
    - 12.4.1 Overview of SOI Audits
    - 12.4.2 Overview of the Software Job Aid
    - 12.4.3 Using the Software Job Aid

- 12.4.4 General Recommendations for the Auditor
- 12.4.5 General Recommendations for the Auditee (the Applicant/Developer)
- 12.4.6 SOI Review Specifics
  - 12.4.6.1 SOI 1 Entry Criteria, Expectations, and Preparation Recommendations
  - 12.4.6.2 SOI 2 Entry Criteria, Expectations, and Preparation Recommendations
  - 12.4.6.3 SOI 3 Entry Criteria, Expectations, and Preparation Recommendations
  - 12.4.6.4 SOI 4 Entry Criteria, Expectations, and Preparation Recommendations
- 12.5 Software Maturity Prior to Certification Flight Tests
- References

## Part IV Tool Qualification and DO-178C Supplements

### 13 DO-330 and Software Tool Qualification

#### Acronyms

#### 13.1 Introduction

#### 13.2 Determining Tool Qualification Need and Level (DO-178C Section 12.2)

#### 13.3 Qualifying a Tool (DO-330 Overview)

##### 13.3.1 Need for DO-330

##### 13.3.2 DO-330 Tool Qualification Process

#### 13.4 Special Tool Qualification Topics

##### 13.4.1 FAA Order 8110.49

##### 13.4.2 Tool Determinism

##### 13.4.3 Additional Tool Qualification Considerations

##### 13.4.4 Tool Qualification Pitfalls

##### 13.4.5 DO-330 and DO-178C Supplements

##### 13.4.6 Using DO-330 for Other Domains

#### References

- 14 DO-331 and Model-Based Development and Verification
  - Acronyms
  - 14.1 Introduction
  - 14.2 Potential Benefits of Model-Based Development and Verification
  - 14.3 Potential Risks of Model-Based Development and Verification
  - 14.4 Overview of DO-331
  - 14.5 Certification Authorities Recognition of DO-331
  - References
- 15 DO-332 and Object-Oriented Technology and Related Techniques
  - Acronyms
  - 15.1 Introduction to Object-Oriented Technology
  - 15.2 Use of OOT in Aviation
  - 15.3 OOT in Aviation Handbook
  - 15.4 FAA-Sponsored Research on OOT and Structural Coverage
  - 15.5 DO-332 Overview
    - 15.5.1 Planning
    - 15.5.2 Development
    - 15.5.3 Verification
    - 15.5.4 Vulnerabilities
    - 15.5.5 Type Safety
    - 15.5.6 Related Techniques
    - 15.5.7 Frequently Asked Questions
  - 15.6 OOT Recommendations
  - 15.7 Conclusion
  - References
  - Recommended Readings
- 16 DO-333 and Formal Methods
  - Acronyms
  - 16.1 Introduction to Formal Methods
  - 16.2 What Are Formal Methods?

- 16.3 Potential Benefits of Formal Methods
- 16.4 Challenges of Formal Methods
- 16.5 DO-333 Overview
  - 16.5.1 Purpose of DO-333
  - 16.5.2 DO-333 and DO-178C Compared
    - 16.5.2.1 Planning and Development
    - 16.5.2.2 Configuration Management, Quality Assurance, and Certification Liaison
    - 16.5.2.3 Verification
- 16.6 Other Resources
- References

## Part V Special Topics

- 17 Noncovered Code (Dead, Extraneous, and Deactivated Code)
  - Acronyms
  - 17.1 Introduction
  - 17.2 Extraneous and Dead Code
    - 17.2.1 Avoiding Late Discoveries of Extraneous and Dead Code
    - 17.2.2 Evaluating Extraneous or Dead Code
  - 17.3 Deactivated Code
    - 17.3.1 Planning
    - 17.3.2 Development
    - 17.3.3 Verification
  - References
- 18 Field-Loadable Software
  - Acronyms
  - 18.1 Introduction
  - 18.2 What Is Field-Loadable Software?
  - 18.3 Benefits of Field-Loadable Software
  - 18.4 Challenges of Field-Loadable Software
  - 18.5 Developing and Loading Field-Loadable Software

- 18.5.1 Developing the System to Be Field-Loadable
  - 18.5.2 Developing the Field-Loadable Software
  - 18.5.3 Loading the Field-Loadable Software
  - 18.5.4 Modifying the Field-Loadable Software
- 18.6 Summary
- References
- 19 User-Modifiable Software
  - Acronyms
  - 19.1 Introduction
  - 19.2 What Is User-Modifiable Software?
  - 19.3 Examples of UMS
  - 19.4 Designing the System for UMS
  - 19.5 Modifying and Maintaining UMS
  - References
- 20 Real-Time Operating Systems
  - Acronyms
  - 20.1 Introduction
  - 20.2 What Is an RTOS?
  - 20.3 Why Use an RTOS?
  - 20.4 RTOS Kernel and Its Supporting Software
    - 20.4.1 RTOS Kernel
    - 20.4.2 Application Program Interface
    - 20.4.3 Board Support Package
    - 20.4.4 Device Driver
    - 20.4.5 Support Libraries
  - 20.5 Characteristics of an RTOS Used in Safety-Critical Systems
    - 20.5.1 Deterministic
    - 20.5.2 Reliable Performance
    - 20.5.3 Compatible with the Hardware
    - 20.5.4 Compatible with the Environment
    - 20.5.5 Fault Tolerant
    - 20.5.6 Health Monitoring

- 20.5.7 Certifiable
- 20.5.8 Maintainable
- 20.5.9 Reusable
- 20.6 Features of an RTOS Used in Safety-Critical Systems
  - 20.6.1 Multitasking
  - 20.6.2 Guaranteed and Deterministic Schedulability
    - 20.6.2.1 Scheduling between Partitions
    - 20.6.2.2 Scheduling within Partitions
  - 20.6.3 Deterministic Intertask Communication
  - 20.6.4 Reliable Memory Management
  - 20.6.5 Interrupt Processing
  - 20.6.6 Hook Functions
  - 20.6.7 Robustness Checking
  - 20.6.8 File System
  - 20.6.9 Robust Partitioning
- 20.7 RTOS Issues to Consider
  - 20.7.1 Technical Issues to Consider
    - 20.7.1.1 Resource Contention
    - 20.7.1.2 Priority Inversion
    - 20.7.1.3 Memory Leaks
    - 20.7.1.4 Memory Fragmentation
    - 20.7.1.5 Intertask Interference
    - 20.7.1.6 Jitter
    - 20.7.1.7 Vulnerabilities
  - 20.7.2 Certification Issues to Consider
    - 20.7.2.1 Creating a Safe Subset
    - 20.7.2.2 User's Manual
    - 20.7.2.3 Reverse Engineering
    - 20.7.2.4 Deactivated Features
    - 20.7.2.5 Complexity
    - 20.7.2.6 Disconnect with the System
    - 20.7.2.7 Code Compliance Issues

- 20.7.2.8 Error Handling Issues
  - 20.7.2.9 Problem Reporting
  - 20.7.2.10 Partitioning Analysis
  - 20.7.2.11 Other Supporting Software
  - 20.7.2.12 Target Testing
  - 20.7.2.13 Modifications
- 20.8 Other RTOS-Related Topics
  - 20.8.1 ARINC 653 Overview
  - 20.8.2 Tool Support
  - 20.8.3 Open Source RTOSs
  - 20.8.4 Multicore Processors, Virtualization, and Hypervisors
  - 20.8.5 Security
  - 20.8.6 RTOS Selection Questions
- References
- 21 Software Partitioning
  - Acronyms
  - 21.1 Introduction to Partitioning
    - 21.1.1 Partitioning: A Subset of Protection
    - 21.1.2 DO-178C and Partitioning
    - 21.1.3 Robust Partitioning
  - 21.2 Shared Memory (Spatial Partitioning)
  - 21.3 Shared Central Processing Unit (Temporal Partitioning)
  - 21.4 Shared Input/Output
  - 21.5 Some Partitioning-Related Challenges
    - 21.5.1 Direct Memory Access
    - 21.5.2 Cache Memory
    - 21.5.3 Interrupts
    - 21.5.4 Interpartition Communication
  - 21.6 Recommendations for Partitioning
  - References
- 22 Configuration Data
  - Acronyms

- 22.1 Introduction
- 22.2 Terminology and Examples
- 22.3 Summary of DO-178C Guidance on Parameter Data
- 22.4 Recommendations
- References
- 23 Aeronautical Data
  - Acronyms
  - 23.1 Introduction
  - 23.2 DO-200A: Standards for Processing Aeronautical Data
  - 23.3 FAA Advisory Circular 20-153A
  - 23.4 Tools Used for Processing Aeronautical Data
  - 23.5 Other Industry Documents Related to Aeronautical Data
    - 23.5.1 DO-201A: Standards for Aeronautical Information
    - 23.5.2 DO-236B: Minimum Aviation System Performance Standards: Required Navigation Performance for Area Navigation
    - 23.5.3 DO-272C: User Requirements for Aerodrome Mapping Information
    - 23.5.4 DO-276A: User Requirements for Terrain and Obstacle Data
    - 23.5.5 DO-291B: Interchange Standards for Terrain, Obstacle, and Aerodrome Mapping Data
    - 23.5.6 ARINC 424: Standard, Navigation System Database
    - 23.5.7 ARINC 816-1: Embedded Interchange Format for Airport Mapping Database
  - References
- 24 Software Reuse
  - Acronyms
  - 24.1 Introduction
  - 24.2 Designing Reusable Components
  - 24.3 Reusing Previously Developed Software
    - 24.3.1 Evaluating PDS for Use in Civil Aviation Products

24.3.2 Reusing PDS That Was Not Developed Using DO-178[ ]

24.3.3 Additional Thoughts on COTS Software

24.4 Product Service History

24.4.1 Definition of Product Service History

24.4.2 Difficulties in Seeking Credit Using Product Service History

24.4.3 Factors to Consider When Claiming Credit Using Product Service History

References

25 Reverse Engineering

Acronyms

25.1 What Is Reverse Engineering?

25.2 Examples of Reverse Engineering

25.3 Issues to Be Addressed When Reverse Engineering

25.4 Recommendations for Reverse Engineering

References

26 Outsourcing and Offshoring Software Life Cycle Activities

Acronyms

26.1 Introduction

26.2 Reasons for Outsourcing

26.3 Challenges and Risks in Outsourcing

26.4 Recommendations to Overcome the Challenges and Risks

26.5 Summary

Appendix A: Example Transition Criteria

Appendix B: Real-Time Operating System Areas of Concern

Appendix C: Questions to Consider When Selecting a Real-Time Operating System for a Safety-Critical System

Appendix D: Software Service History Questions

Index

# Preface

It's with a great sense of responsibility and complete humility that I present this book. After two years of researching and writing, I feel I've only scratched the surface. I hope that what I share and have experienced will help you in your professional endeavors.

My passion is safety and making its realization practical. As early as high school, I wrote a column for the local newspaper on safety tips, extensively researching (as much as one can in rural America before the Internet): bathtub safety, tractor safety, and electrical safety, among others. In my senior year of high school, I competed in the state persuasive speech competition—the subject was seatbelt safety. My desire to promote safety led me to the Federal Aviation Administration (FAA) and is the primary motivation behind this book.

Software is only one link in the overall safety chain. Yet, it is an important link and one which grows more vital each year. As software becomes more prevalent and is used in more critical ways, its risks and contributions to safety increase. At the same time that the risk due to software is increasing, the business models of many companies seem to be shifting—more is expected with less, schedule becomes king, and software engineers are treated like pieces on a game board that can be randomly reorganized.

This book is intended to be a tool for those caught in the churn of the industry. Perhaps you are a systems engineer or manager, a software manager, a software engineer, a quality assurance engineer, or a student striving to learn from the experiences of others. You want to do an outstanding job but are overwhelmed with schedule and budget pressures. I

hope that this book, based on the last 20 years of my life in the aviation industry, will help you as you pursue quality and excellence.

# Acknowledgments

This book would not have been possible without the help of many colleagues and friends. The following reviewed portions of the text and provided valuable input: Patty (Bartels) Bath of AVISTA, Jim Chelini of VEROCEL, Andrew Elliott of Design Assurance, Kelly Hayhurst of NASA's Langley Research Center, Wendy Ljungren of GE Aviation, Jeff Knickerbocker of Sunrise Certification, George Romanski of VEROCEL, Andrew Kornecki of Embry-Riddle Aeronautical University, Fred Strathmann of Springton Technologies, Steven VanderLeest of DornerWorks, and Dan Veerhusen who retired from Rockwell Collins. Thanks to each of you for your time and your friendship. You challenged my thinking and made the text better.

This book would have never been conceivable without those who mentored, guided, and opened doors of opportunity for me early in my career. There are too many to name, but I'm especially indebted to the following:

- Milt Sills, former vice president of engineering at Cessna Aircraft Company. He mentored me and stoked my love of aviation. He passed away in 2003 but continues to have a daily influence on my life.
- Jim Peterson, retired FAA systems and equipment branch manager at the Wichita Aircraft Certification Office. Jim recruited me to the FAA and instilled in me a passion to stand up for what's right no matter what the cost.
- Elizabeth Erickson, retired FAA director of Aircraft Certification Services and currently with Erickson Aviation Services. Beth believed

in me and mentored me in challenging times at FAA headquarters in Washington, District of Columbia.

- Cheryl Dorsey, owner of Digital Flight. Cheryl recruited me into the ranks of software engineer at FAA headquarters. My background was avionics systems and electronic hardware, but Cheryl realized my potential and ignited my desire for safe software.
- Mike DeWalt, FAA's chief scientific technical advisor for aircraft computer software. Mike guided me when I was very young and inexperienced.

I am also grateful to RTCA, Inc. I've been privileged to serve on five committees—in leadership roles on three of them. RTCA has been very supportive of this book and generous to grant permission for references and quotes. You can learn more about RTCA, Inc. or purchase their documents by contacting them at

RTCA, Inc.

1150 18th Street NW Suite 910

Washington, DC 20036

Phone: (202) 833-9339

Fax: (202) 833-9434

Web: [www.rtca.org](http://www.rtca.org)

Thanks also to my dear friends Susan Corcoran and Denise Polansky who helped with the final stages of this effort, to the team who prayed for me through to the finish line, and to the staff at Taylor & Francis Group for their support and help. Nora Konopka, Jill Jurgensen, and Glenon Butler have been terrific.

MATLAB<sup>®</sup> and Simulink<sup>®</sup> are registered trademarks of The MathWorks, Inc. For product information, please contact:

The MathWorks, Inc.

3 Apple Hill Drive

Natick, MA, 01760-2098 USA

Tel: 508-647-7000

Fax: 508-647-7001

E-mail: [info@mathworks.com](mailto:info@mathworks.com)

Web: [www.mathworks.com](http://www.mathworks.com)

# Author



**Leanna Rierson** is an independent consultant in software, complex electronic hardware, and integrated module avionics (IMA) development for safety-critical systems, with emphasis on civil aviation. She has over 20 years of experience in the software and aviation industry. Rierson spent nine years as a software and avionics specialist at the United States Federal Aviation Administration (FAA)—five of those years were in the position of chief scientific and technical advisor for aircraft computer software. Rierson has published numerous papers on safety-critical software, integrated modular avionics, and aviation; led many national and international engineering teams and workshops; and developed courses, policies, handbooks, and guidance material for the FAA. She served as a

subgroup cochair and editorial team leader on the RTCA special committee that wrote DO-178C and six other related documents. Rierison has taught DO-178B, and now DO-178C, to hundreds of professionals. She is an FAA Designated Engineering Representative (DER) with Level A authority in the software and complex hardware technical areas. She has worked with numerous aircraft and avionics companies, including Boeing, Cessna, Learjet, Embraer, Rockwell Collins, GE Aviation, Honeywell, and numerous others. She is currently working part-time for the Rockwell Collins avionics certification team.

Rierison holds a master's degree in software engineering from Rochester Institute of Technology and a bachelor's degree in electrical engineering from Wichita State University. She also attended Ozark Christian College and received a master's degree from Johnson Bible College. She is currently pursuing a doctorate degree.

# Part I Introduction

Part I sets the foundation for the entire book. It includes an explanation of what safety-critical software is and why it is important in today's environment. An overview of the book and an explanation of the approach taken are also provided.

# 1 Introduction and Overview

## Acronym

|      |                                                  |
|------|--------------------------------------------------|
| DER  | Designated Engineering Representative            |
| EASA | European Aviation Safety Agency                  |
| FAA  | Federal Aviation Administration                  |
| IEEE | Institute of Electrical and Electronic Engineers |
| IMA  | Integrated Modular Avionics                      |
| NASA | National Aeronautics and Space Administration    |

## 1.1 Defining Safety-Critical Software

A general definition for *safety* is the “freedom from those conditions that can cause death, injury, illness, damage to or loss of equipment or property, or environmental harm” [1]. The definition of *safety-critical software* is more subjective. The Institute of Electrical and Electronic Engineers (IEEE) defines safety-critical software as: “software whose use in a system can result in unacceptable risk. Safety-critical software includes software whose operation or failure to operate can lead to a hazardous state, software intended to recover from hazardous states, and software intended to mitigate the severity of an accident” [2]. The *Software Safety Standard* published by the U.S. National Aeronautics and Space Administration

(NASA) identifies software as *safety-critical* if at least one of the following criteria is satisfied [3,4]:

1. It resides in a safety-critical system (as determined by a hazard analysis) AND at least one of the following:
  - Causes or contributes to a hazard.
  - Provides control or mitigation for hazards.
  - Controls safety-critical functions.
  - Processes safety-critical commands or data.
  - Detects and reports, or takes corrective action, if the system reaches a specific hazardous state.
  - Mitigates damage if a hazard occurs.
  - Resides on the same system (processor) as safety-critical software.
2. It processes data or analyzes trends that lead directly to safety decisions.
3. It provides full or partial verification or validation of safety-critical systems, including hardware or software systems.

From these definitions, it can be concluded that software by itself is neither safe nor unsafe; however, when it is part of a safety-critical system, it can cause or contribute to unsafe conditions. Such software is considered *safety critical* and is the primary theme of this book.

## 1.2 Importance of Safety Focus

In 1993, Ruth Wiener wrote in her book *Digital Woes: Why We Should Not Depend Upon Software*: “Software products—even programs of modest size—are among the most complex artifacts that humans produce, and software development projects are among our most complex undertakings. They soak up however much time or money, however many people we throw at them. The results are only modestly reliable. Even after the most thorough and rigorous testing some bugs remain. We can never test all threads through the system with all possible inputs” [5]. Since that time, society has become more and more reliant on software. We are past the

point of returning to purely analog systems. Therefore, we must make every effort to ensure that software-intensive systems are reliable and safe. The aviation industry has a good track record, but as complexity and criticality increase, care must be taken.

It is not unusual to hear statements similar to: “Software has not caused any major aircraft accidents, so why all the fuss?” The contribution of software to aircraft accidents is debatable, because the software is part of a larger system and most investigations focus on the system-level aspects. However, in other domains (e.g., nuclear, medical, and space) software errors have definitely led to loss of life or mission. The Ariane Five rocket explosion, the Therac-25 radiation overdoses, and a Patriot missile system shutdown during the Gulf War are some of the more well-known examples. The historical record for safety-critical software in civil aviation has been quite respectable. However, now is not the time to sit back and marvel at our past. The future is riskier for the following reasons:

- *Increased lines of code:* The number of lines of code used in safety-critical systems is increasing. For example, the lines of code from the Boeing 777 to the recently certified Boeing 787 have increased eightfold to tenfold, and future aircraft will have even more software.
- *Increased complexity:* The complexity of systems and software is increasing. For example, Integrated Modular Avionics (IMA) provides weight savings, easier installation, efficient maintenance, and reduced cost of change. However, IMA also increases the system’s complexity and results in functions previously isolated to physically different federated hardware being combined onto a single hardware platform with the associated loss of multiple functions should the hardware fail. This increased complexity makes it more difficult to thoroughly analyze safety impact and to prove that intended functionality is met without unintended consequences.
- *Increased criticality:* At the same time that size and complexity of software are increasing, so is the criticality. For example, flight control surface interfaces were almost exclusively mechanical 10 years ago. Now, many aircraft manufacturers are transitioning to fly-by-wire software to control the flight control surfaces, since the fly-by-wire

software includes algorithms to improve aircraft performance and stability.

- *Technology changes:* Electronics and software technology are changing at a rapid rate. It is challenging to ensure the maturity of a technology before it becomes obsolete. For example, safety domains require robust and proven microprocessors; however, because a new aircraft development takes around 5 years, the microprocessors are often nearly obsolete before the product is flight tested. Additionally, the changes in software technology make it challenging to hire programmers who know Assembly, C, or Ada (the most common languages used for airborne software). These real-time languages are not taught in many universities. Software developers are also getting further away from the actual machine code generated.
- *More with less:* Because of economic drivers and the pressure to be profitable, many (maybe even most) engineering organizations are being requested to do more with less. I've heard it described as follows: "First, they took away our secretaries; next, they took away our technical writers; and then, they took away our colleagues." Most good engineers are doing the work of what was previously done by two or more people. They are spread thin and exhausted. People are less effective after months of overtime. A colleague of mine puts it this way: "Overtime is for sprints, not for marathons." Yet, many engineers are working overtime for months or even years.
- *Increased outsourcing and offshoring:* Because of the market demands and the shortage of engineers, more and more safety-critical software is being outsourced and offshored. While not always an issue, oftentimes, outsourced and offshored teams do not have the systems domain knowledge and the safety background needed to effectively find and remove critical errors. In fact, without appropriate oversight, they may even inject errors.
- *Attrition of experienced engineers:* A number of engineers responsible for the current safety record are retiring. Without a rigorous training and mentoring program, younger engineers and managers do not understand why key decisions and practices were put into place. So they either don't follow them or they have them removed altogether. A colleague recently expressed his dismay after his team gave a presentation on speed brakes to their certification authority's new

systems engineer who is serving as the systems *specialist*. After the 2-hour presentation, the new engineer asked: “You aren’t talking about the wheels, are you?”\*

- *Lack of available training*: There are very few safety-focused degree programs. Additionally, there is little formal education available for system and software validation and verification.

Because of these and other risk drivers, it is essential to focus even more on safety than ever before.

## 1.3 Book Purpose and Important Caveats

I am truly honored and humbled that you have decided to read this book. The purpose of this text is to equip practicing engineers and managers with the information they need to develop safety-critical software for aviation. Over the last 20 years, I’ve had the privilege of working as an avionics developer, aircraft developer, certification authority, Federal Aviation Administration (FAA) Designated Engineering Representative (DER), consultant, and instructor. I’ve evaluated safety-critical software on dozens of systems, such as flight controls, IMA, landing gear, flaps, ice protection, nose wheel steering, flight management, battery management, displays, navigation, terrain awareness and warning, traffic collision and avoidance, real-time operating systems, and many more. I’ve worked with teams of 500 and teams of 3. It has been and continues to be an exciting adventure!

The variety of positions and systems has allowed me to experience and observe common issues, as well as effective solutions, for developing safetycritical software. This book is written using these experiences to help practicing aircraft systems engineers, avionics and electronic systems engineers, software managers, software development and verification engineers, certification authorities and their designees, quality assurance engineers, and others who have a desire to implement and assure safe software.

As a practical guide for developing and verifying safety-critical software, the text provides concrete guidelines and recommendations based on real-life projects. However, it is important to note the following:

- The information herein represents one person's view. It is written based on personal experiences and observations, as well as research and interactions with some of the brightest and most committed people in the world. Every effort has been made to present accurate and complete advice; however, every day I learn new things that clarify and expand my thinking. Throughout the book terms like *typically*, *usually*, *normally*, *most of the time*, *many times*, *oftentimes*, etc. are used. These generalizations are made based on the numerous projects I've been involved in; however, there are many projects and billions of lines of code that I have not seen. Your experience may differ and I welcome that insight. If you wish to clarify or debate a topic, ask a question, or share your thoughts, feel free to send an email to both of the following addresses: [LRierson1@aol.com](mailto:LRierson1@aol.com) and [Digital\\_Safety@sbcglobal.net](mailto:Digital_Safety@sbcglobal.net).
- I have used a personal and somewhat informal tone. Having taught DO-178B (and now DO-178C) to hundreds of students, this book is intended to be an interaction between me (the instructor) and you (the engineer). Since I do not know your background, I have attempted to write in such a way that the text is useful regardless of your experience. I have integrated *war stories* and some occasional humor, as I do in the classroom. At the same time, professionalism has been a constant goal.
- Because this book focuses on safety-critical software, the text is written with higher criticality software (such as DO-178C levels A and B) in mind. For lower levels of criticality, some of the activities may not be required.
- While the focus of this book is on aviation software, DO-178C compliance, and aircraft certification, many of the concepts and best practices apply to other safety-critical or mission-critical domains, such as medical, nuclear, military, automotive, and space.
- Because of my background as a certification authority and as a DER, reading this book may at times be a little like crawling into the mind of a certification authority. (Don't be afraid.) Both FAA and the European Aviation Safety Agency (EASA) policy and guidance materials are discussed; however, the FAA guidance is primarily used unless there is a significant difference. While the content of this book is intended to explain and be consistent with the certification authorities' policy and

guidance as they exist at the time of this writing, this book does not constitute certification authority policy or guidance. Please consult with your local authority for the policy and guidance applicable to your specific project.

- Having traveled the globe and interacted with engineers on six continents, I have made every effort to present an international perspective on safety-critical software and certification. However, because the majority of my work has taken place in the United States and on FAA certified projects, that perspective is presented.
- Several RTCA documents, including DO-178C, are referenced throughout this document. It has been exciting to serve in leadership roles on three RTCA committees and to have a key role in developing the referenced documents. As noted in the foreword, RTCA has been gracious to allow me to reference and quote portions of their documents. However, this book is not a replacement for those documents. I've attempted to cover the highlights and guide you through some of the nuances of DO-178C and related documents. However, if you are developing software that must comply with the RTCA documents, be sure to read the full document. You can learn more about RTCA, Inc. or purchase their documents by contacting them at the following:

RTCA, Inc.

1150 18th Street NW Suite 910

Washington, DC 20036

Phone: (202) 833-9339

Fax: (202) 833-9434

Web: [www.rtca.org](http://www.rtca.org)

- This book is written so that you can read it from beginning to end, or you can read selected chapters as needed. You will find occasional repetition between some chapters. This is intentional, since some readers may choose to use this book as a reference rather than read it straight through. References to related chapters are included throughout to help those who may not read the text cover to cover.

## 1.4 Book Overview

This book is divided into five parts. Part I (this part) provides the introduction and sets the foundation. Part II explains the role of software in the overall system and provides a summary of the system and safety assessment processes used for aviation. Part III starts with an overview of RTCA's DO-178C, entitled *Software Considerations in Airborne Systems and Equipment Certification*, and the six other documents that were published with DO-178C. The section then goes through the DO-178C processes—providing insight into the guidance and suggestions for how to effectively apply it. Part IV explores four RTCA guidance documents that were released with DO-178C. The topics covered are software tool qualification (DO-330), model-based development and verification (DO-331), object-oriented technology and related techniques (DO-332), and formal methods (DO-333). Part V covers special topics related to DO-178C and safety-critical software development. These special topics are focused on aviation but may also be applicable to other domains and include noncovered code (extraneous, dead, and deactivated code), field-loadable software, user-modifiable software, real-time operating systems, partitioning, configuration data, aeronautical databases, software reuse, previously developed software, reverse engineering, and outsourcing and offshoring.

There are many other subjects that are not covered, including aircraft electronic hardware, electronic flight bags, and software security. These topics are related to software and are occasionally referenced in the text. However, due to space and time limitations, they are not covered. Some of these topics are, however, slated to be covered in the new edition of CRC Press's *Digital Avionics Handbook*.

## References

1. K. S. Mendis, Software safety and its relation to software quality assurance, ed. G. G. Schulmeyer, *Software Quality Assurance Handbook*, 4th edn., Chapter 9 (Norwood, MA: Artech House, 2008).
2. IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std-610-1990 (Los Alamitos, CA: IEEE Computer Society Press, 1990).

3. National Aeronautics and Space Administration, *Software Safety Standard*, NASA-STD-8719.13B (Washington, DC: NASA, July 2004).
4. B. A. O'Connell, Achieving fault tolerance via robust partitioning and N-modular redundancy, Master's thesis (Cambridge, MA: Massachusetts Institute of Technology, February 2007).
5. L. R. Wiener, *Digital Woes: Why We Should Not Depend on Software* (Reading, MA: Addison-Wesley, 1993).

\*For those of you who don't know, speed brakes are on the wings.

# Part II Context of Safety-Critical Software Development

Part II provides an overview of how software fits into the overall system development effort. In order to successfully implement *safe software*, one must first understand its role in the system. Focusing only on the software without considering the system and its safety characteristics is like treating the symptoms of a serious illness without getting at the root cause of the illness. In my experience, there are five key factors that must be thoroughly and constantly addressed when developing safety-critical software:

- 1. Well-documented systems architecture and requirements**  
**definition.** The system architecture and requirements must focus on safety and be written down. It is next to impossible to successfully implement undocumented requirements. To ensure they are the right requirements, the system requirements also need to be validated for correctness and completeness.
- 2. Solid safety practices at all levels of development.** Throughout the development, potential hazards and risks must be identified and addressed. Safety should be integral to all processes at all levels of the development—not just the responsibility of the safety organization.
- 3. Disciplined implementation.** The documented requirements and safety attributes must be accurately implemented. This includes both assuring that the requirements are implemented and that no unintended functionality is added. A well-defined change management process is essential to successful implementation and iterative development.

4. **Well-qualified personnel.** Safety-critical systems are implemented by human beings. The personnel must be knowledgeable of the domain, safety, and the technology being used to implement the system. People working in the safety-critical field should be the cream of the crop—perfectionists who strive for 100% on every task. Safety-critical systems demand a commitment to *excellence*—not just *good enough*.
5. **Thorough testing at all levels.** There is definitely a role for reviews and analyses, but they cannot replace the need to prove functionality and safety claims with the integrated software and hardware.

This book concentrates on software, but always be mindful that the software is only one aspect of the overall system. The five factors mentioned earlier should be addressed throughout the entire system development. This Part (II) on systems development and safety sets the context for subsequent Parts (III to V), where the software development and integral processes, as well as special software topics, will be detailed.

# 2 Software in the Context of the System

## Acronym

|         |                                                                              |
|---------|------------------------------------------------------------------------------|
| AC      | Advisory Circular                                                            |
| ARP     | Aerospace Recommended Practice                                               |
| DAL     | development assurance level                                                  |
| EASIS   | Electronic Architecture and System Engineering for Integrated Safety Systems |
| EUROCAE | European Organization for Civil Aviation Equipment                           |
| FAA     | Federal Aviation Administration                                              |
| PSSA    | preliminary system safety assessment                                         |
| RAM     | random access memory                                                         |
| RBT     | requirements-based test                                                      |
| ROM     | read only memory                                                             |
| SAE     | Society of Automotive Engineers                                              |
| TCAS    | traffic collision and avoidance system                                       |

## 2.1 Overview of System Development

Before diving into the details of how to develop safety-critical software, it is important to understand the context of software in the overall system. Software operates in the context of a system, and safety is a property of the overall system.

Software, in and of itself, is neither safe nor unsafe. However, software influences safety in a variety of system types, including aircraft systems, automotive systems, nuclear systems, and medical systems. In order to develop software that enhances rather than impairs safety, one must first understand the system in which the software operates and the overall system safety process. This chapter examines the process for developing safetycritical systems. Chapter 3 examines the system safety assessment process, which is part of the overall system development life cycle. The system and safety assessment overviews are presented from an aviation perspective; however, the concepts may be transferred to other safety-critical domains.

An aircraft is a large system comprising multiple systems—it is essentially a system of systems. Aircraft systems can include navigation, communication, landing gear, flight control, displays, collision avoidance, environmental control, in-flight entertainment, electrical power, engine control, ground steering, thrust reversers, fuel, air data, and more. Each system has an effect on the overall operation of the aircraft. Obviously, some have more safety impact than others. Because of this, the civil aviation regulations and supporting advisory material classify failure categories according to risk. Five failure condition categories are used: catastrophic, hazardous/severe-major, major, minor, and no effect (these categories are discussed in Chapter 3). Each function in each system, as well as the combination of functionality, is assessed for its overall impact on safe aircraft operation. The aircraft operation includes multiple phases, such as, taxi, take-off, climb, cruise, descent, and landing. System safety is evaluated for each phase, as the effects of failure conditions can vary greatly, depending on the phase.

Developing a safe system typically requires years of domain expertise with the specific type of system, as well as a thorough understanding of how an aircraft and its systems operate and interact with each other and human operators. In recent years, I have become alarmed that some organizations seem to believe that anyone with an engineering degree can develop an aircraft system. I am not opposed to globalization and profit, but developing the core expertise and competency to build safety-critical systems requires significant time, resources, and integrity.

For civil aviation, developers are encouraged to use SAE's Aerospace Recommended Practice (ARP) 4754 revision A (referred to as ARP4754A), entitled *Guidelines for Development of Civil Aircraft and Systems*. ARP4754 was published by SAE (formerly known as Society of Automotive Engineers) in 1996 and developed by a team of aircraft and avionics manufacturers, with input from certification authorities. EUROCAE (European Organization for Civil Aviation Equipment) also published an equivalent document, ED-79. In December 2010, SAE and EUROCAE published an update to the recommended practice:

ARP4754A and ED-79A, respectively. On September 30, 2011, the Federal Aviation Administration (FAA) published Advisory Circular (AC) 20–174, entitled *Development of civil aircraft systems*. AC 20–174 recognizes ARP4754A as “an acceptable method for establishing a development assurance process.”

ARP4754A divides the system development process into six phases: planning, aircraft function development, allocation of aircraft functions to systems, development of system architecture, allocation of system requirements to items (including allocation to software and hardware), and system implementation (including the software and hardware development) [1]. In addition, the following integral processes are identified in ARP4754A: safety assessment, development assurance level assignment, requirements capture, requirements validation, implementation verification, configuration management, process assurance, and certification authority coordination [1]. ARP4754A’s scope includes the system development through the certification completion; however, in addition to the system phases described in ARP4754A, system development includes activities before product launch (as in needs determination and concept exploration) and after certificate issuance (such as, production and manufacturing, delivery and deployment, operations, maintenance and support, and retirement or disposal) [2,3]. Figure 2.1 illustrates the overall system development framework with the ARP4754A processes shown in gray.

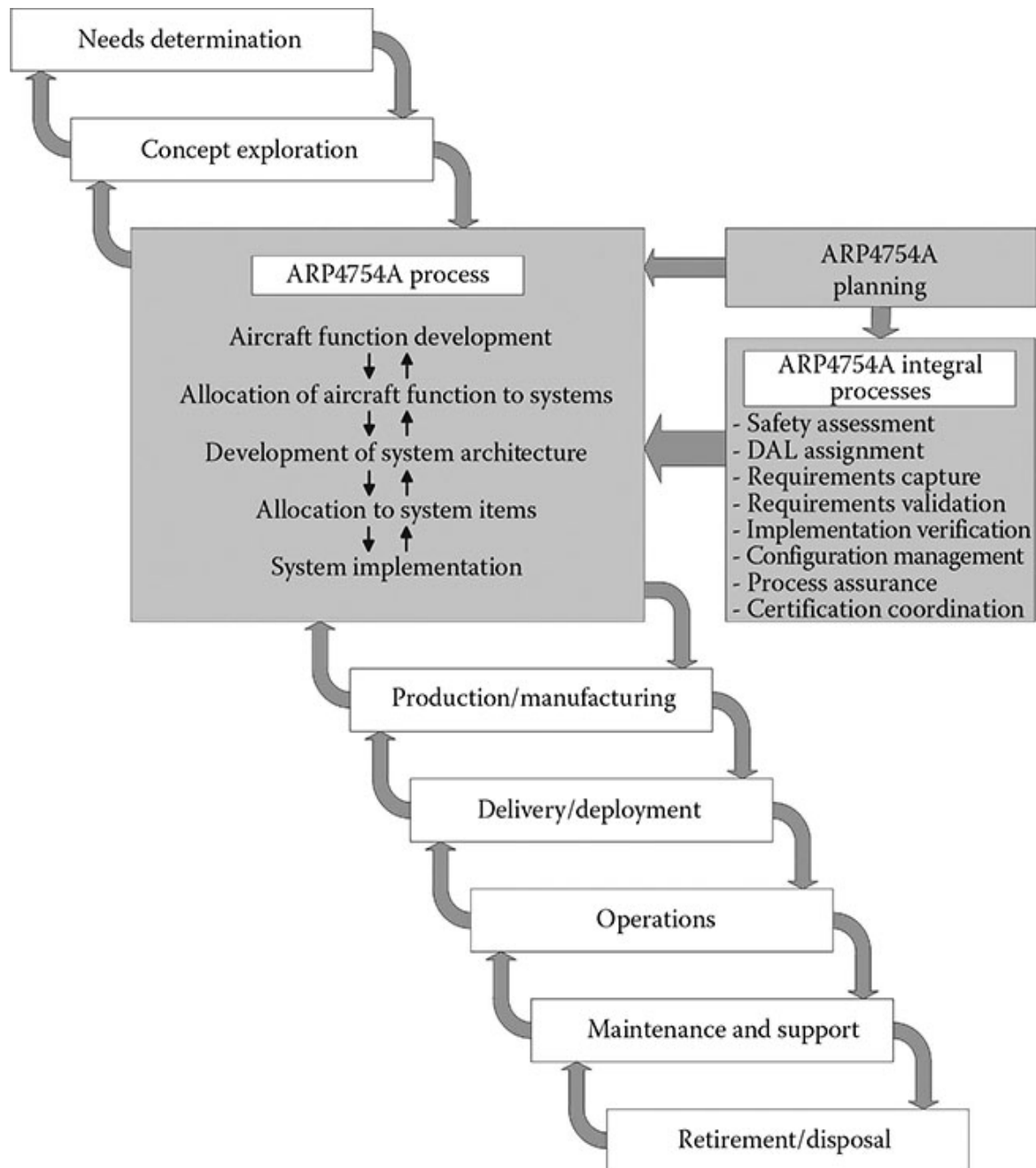


**Figure 2.1** Overall system development process.

As will be discussed in Chapter 3, the safety assessment process runs parallel to the system development process. As the system progresses from high-level design to implementation, the safety aspects are evaluated to ensure that the system satisfies the required safety levels. While ARP4754A proposes a top-down approach, it recognizes that most systems are developed with an iterative

and concurrent approach, using both top-down and bottom-up development strategies. Most aircraft are not made of systems developed from a clean sheet. Instead, aircraft and systems tend to be *derivative projects* that build upon existing functionality and products. Therefore, it is important that the overall development life cycle consider the baseline systems and develop a realistic life cycle that meets both the customer needs (documented in the form of requirements) and the safety objectives.

## 2.2 System Requirements

### 2.2.1 Importance of System Requirements

Chapter 3 will delve into the safety assessment process. For now, however, let us consider some important aspects of the system development. After living through numerous product developments, I have come to realize that many of the problems encountered in software development come from poorly defined system requirements and architecture. Software developers definitely have their problems (which will be discussed in future chapters), but many of the software issues are exacerbated by immature, incomplete, incorrect, ambiguous, and/or poorly defined system requirements. From my experience, most companies ought to spend more effort defining, validating, and verifying system requirements. This is mandatory, given the increase in system complexity and criticality. While there has been some improvement, much more is needed if we are to continue to develop safe, reliable, and dependable systems. The certification authorities also realize this need, and with the formal recognition of ARP4754A, they are increasing their attention on the system development processes—particularly, the requirements definition, validation, and implementation verification.

### 2.2.2 Types of System Requirements

ARP4754A recommends that the following types of requirements be developed [1]:

- *Safety requirements* identify functions that either contribute to or directly affect the aircraft safety. The safety requirements are identified by the safety assessment process and include minimum performance constraints for both

availability (continuity of function) and integrity (correctness of behavior). They should be uniquely identified and traced throughout the system life cycle.

- *Functional requirements* specify functionality of the system to obtain desired performance. Functional requirements are usually a combination of customer requirements, operational requirements, performance requirements (for example, timing, speed, range, accuracy, and resolution), physical and installation requirements, maintainability requirements, interface requirements, and constraints. Functional requirements may also be safety requirements or have safety consequences that should be evaluated for impact on safety.
- *Other requirements* include regulatory requirements and derived requirements. Derived requirements are typically the result of design decisions that are made as the system architecture matures. Since they may not trace to a higher level requirement, they need to be validated and assessed by the system safety process.

## 2.2.3 Characteristics of Good Requirements

Good requirements do not just happen. They require considerable effort, focus, and discipline. To write good requirements, it is important to understand the characteristics of such requirements. Requirements authors and reviewers should incorporate the following characteristics of good system requirements:

- *Atomic*—each requirement should be a single requirement.
- *Complete*—each requirement contains all of the necessary information to define the desired system functionality. Each requirement should be able to stand alone without further amplification.
- *Concise*—each requirement should simply and clearly state what must be done and only what must be done. It should be easy to read and understand—even by nontechnical users of the system. In general, a textual requirement should not contain over 30–50 words. Requirements represented using graphics should also be concise.
- *Consistent*—requirements should not contradict or duplicate other requirements. Consistent requirements also use the same terminology throughout the specification. Requirements are not the place to practice creative writing.
- *Correct*—each requirement should be the right requirement for the system being defined. It should convey accurate information. This attribute is

ensured by the requirement's validation effort.

- *Implementation free*—each requirement should state *what* is required without identifying *how* to implement it. In general, requirements should not specify design or implementation. However, there may be some exceptions, such as interface or derived system requirements.
- *Necessary*—each requirement should state an essential capability, characteristic, or quality factor. If the requirement is removed, a deficiency should exist.
- *Traceable*—each requirement should be uniquely identified and easily traceable through to lower level requirements, design, and testing.
- *Unambiguous*—each requirement should only have one interpretation.
- *Verifiable*—it should be possible to confirm the implementation of each requirement. Therefore, requirements should be quantifiable and include tolerances, as appropriate. Each requirement should be written such that it can be verified by review,<sup>\*</sup> analysis, or test.<sup>†</sup> Except for rare cases, if the behavior cannot be observed during verification, the requirement should be rewritten. For example, negative requirements are generally not verifiable and require a rewrite.
- *Viable*—each requirement should be able to be implemented, usable when implemented, and helpful to the overall system construction.

A requirements standard, requirements management plan, or best practices guide is recommended to provide guidance for requirements authors. Writing good requirements is not easy, even if you know for sure what you want to build. When I taught DO-178B classes for the FAA, we had a requirements writing exercise. The class was divided into teams and each team was given a LEGO<sup>®</sup> model. Each team was to write requirements for the model, tear it apart, put the pieces and the written requirements in a bag, and give it to another team to assemble using the requirements. At times, it was hilarious to see the end product, because it did not resemble the original model. Not everyone can write good requirements. In fact, from my experience, very few excel at this task. Good guidance, examples, and training based on the specific domain are invaluable. If graphical representations are used to explain the requirements, the standards should explain the rules and limitations of the graphics. For example, the standards may limit the number of symbols on a page, limit the number of pages for one graphic, limit the depth of a model, and provide detailed meaning for each symbol in the library (since various modeling tools have different interpretations for the symbols and developers have diverse levels of understanding). Chapters 5, 6, and 14 provide more discussion on requirements standards, software requirements development, and model-based development, respectively. These upcoming chapters focus on the software

aspects of requirements but most of the suggestions also apply to systems requirements definition.

## 2.2.4 System Requirements Considerations

Requirements for safety-critical systems specify numerous functionalities. This section highlights some fundamental concepts to consider when developing systems requirements for software-intensive and safety-critical systems.\*

### 2.2.4.1 Integrity and Availability Considerations

System requirements should address both integrity (correct functionality and operation) and availability (continuity of functionality). APR4754A defines integrity and availability as follows [1]:

- *Integrity*: “Qualitative or quantitative attribute of a system or an item indicating that it can be relied upon to work correctly. It is sometimes expressed in terms of the probability of not meeting the work correctly criteria.”
- *Availability*: “Qualitative or quantitative attribute that a system or item is in a functioning state at a given point in time. It is sometimes expressed in terms of the probability of the system (item) not providing its output(s) (i.e., unavailability).”

Loss of integrity results in incorrect operation (a malfunction) of the system. Examples include misleading data on the primary flight display, incorrect resolution advisory from a traffic collision and avoidance system (TCAS), or an autopilot hardover. In order to design for integrity, architectural mitigation is often required to prevent and/or neutralize faults. This may involve the use of redundancy, partitioning, monitoring, dissimilarity, or other mitigation strategies to prevent integrity faults or to protect from such faults. Additionally, in order to address integrity, fault detection and responses or fault containment is often applied and should be captured as systems requirements.

Many systems require protection from loss of availability. Availability differs from integrity in that the function operation must be continuous and/or recoverable under all foreseeable operating conditions (including normal and abnormal conditions). Key functionality must be available during and after random hardware faults, data faults, or software or hardware upset. Systems or functions requiring high availability generally require equipment with very high

reliability and/or utilize redundancy. When redundant designs are used to ensure availability, they should be able to tolerate faults and/or detect and recover from faults quickly. Systems incorporating redundant designs typically have redundancy management and fault management requirements.

It should be noted that some design decisions may impact both integrity and availability. Both must be considered. For example, a system may need to mitigate integrity faults without compromising availability. The original version of ARP4754 noted the relationship between integrity and availability when considering the use of dissimilarity: “It should be noted that architectural dissimilarity impacts both integrity and availability. Since an increase in integrity may be associated with a reduction in availability, and viceversa, the specific application should be analyzed from both perspectives to ensure its suitability” [4].

## 2.2.4.2 Other System Requirements Considerations

In addition to integrity and availability concerns, there are other aspects to consider in the system specification, including:

1. *Safety requirements*, that is, requirements driven by the functional hazard assessment or subsequent safety analyses.
2. *Fault tolerance effects on the systems*, including the identification of fault regions. The Electronic Architecture and System Engineering for Integrated Safety Systems (EASIS) Consortium identifies a *fault region* as “a set of components whose internal disturbances are counted as exactly one fault, regardless of where the disturbances are located, how many occur, and how far they stretch within the fault regions” [5].
3. *Graceful degradation* (e.g., fail-safe mode), including the need for any containment regions. EASIS explains that “a containment region is the set of components which may be adversely affected by the respective malfunction. In other words: The containment region defines the border where fault propagation must stop” [5]. Examples include autopilot failure detection before the flight surfaces move or failed data flagged before the pilot responds.
4. *Error detection*, such as [5]:
  - a. Data checks—to examine data to determine if they are valid, in range, fresh, short-circuited, open-circuited, etc.
  - b. Comparison of redundant data—to compare the outputs of redundant sensors or processing units.