ED-94C

SUPPORTING INFORMATION FOR ED-12C AND ED-109A

CORRIGENDUM 1

INCL CORRIGENDUM 1
FEBRUARY 2021

FOREWORD

1. This corrigendum was developed jointly by the EUROCAE/RTCA Forum for Aeronautical Software, and was approved by the Secretary General of EUROCAE on 12 February 2021.
2. Coordination with RTCA was achieved.
3. Corrigendum 1 corrects and updates minor editorial errors.
4. EUROCAE, an international non-profit organisation, is the European leader in the development of worldwide recognised standards for aviation. This is achieved by utilising the expertise from our members and stakeholders across the global aviation community.
5. EUROCAE standards are developed following an open, transparent and consensus-based process, governed by process and quality management principles, which are clearly documented and is in line with the World Trade Organisation (WTO) Technical Barriers to Trade (TBT) Agreement, Annex 3 “Code of Good Practice for the Preparation, Adoption and Application of Standards”.
6. EUROCAE standards and other deliverables are recommendations and best industry practices. EUROCAE is not an official body of the European governments. EUROCAE standards are intended to complement and support the regulatory and certification framework.
7. Whilst efforts are made during the standards-development process to avoid the inclusion of proprietary information and material, some of the elements of this document may be the subject of patent, copyright or other proprietary rights. EUROCAE shall not be held responsible for identifying any such rights. Trade names and similar terms used within this document do not constitute an endorsement by EUROCAE thereof.
8. Copies of this document may be obtained from:

EUROCAE
9 – 23 rue Paul Lafargue
93200 Saint-Denis
France
Telephone: +33 1 49 46 19 65
Email: eurocae@eurocae.net
Website: www.eurocae.net

CORRIGENDUM 1 TO ED-94C

ANNEX REFERENCE CORRECTION

Purpose: The purpose of this section is to provide a reference correction in Section 3.84.

Description: Amend the last sentence of the first paragraph of the answer in Section 3.84 to correct the cross reference error.

Was: As another example, how can objective 13 of Table A-8 ("Software partitioning integrity is confirmed") be met without knowledge of the high-level requirements, the software architecture, low-level requirements, and Source Code associated with the partitioning?

Change to: As another example, how can objective 13 of Table A-4 ("Software partitioning integrity is confirmed") be met without knowledge of the high-level requirements, the software architecture, low-level requirements, and Source Code associated with the partitioning?

CROSS REFERENCE CORRECTION

Purpose: The purpose of this section is to correct a cross reference error in Section 4.9.5.3.

Description: Amend the first sentence of the second paragraph of Section 4.9.5.3 to correct the cross reference error.

Was: ED-12C/ED-109A section 11.20.j regarding software status states that the SAS should provide a summary of Problem Reports unresolved at the time of approval.

Change to: ED-12C/ED-109A section 11.20.k regarding software status states that the SAS should provide a summary of Problem Reports unresolved at the time of approval.

DOCUMENT REFERENCE CORRECTION

Purpose: The purpose of this section is to correct a document number reference error in Section 5.6.2.

Description: Amend the final sentence of the third bullet of Section 5.6.2 as defined below:

Was: Therefore, objectives 2 and 4 in Table A-6 were added in ED-12B/ED-109A to provide this emphasis.

Change to: Therefore, objectives 2 and 4 in Table A-6 were added in ED-12C/ED-109A to provide this emphasis.

TRANSPOSED ACRONYM DEFINITIONS

Purpose: The purpose of this section is to correct a transposed acronym definition error in Appendix A.

Description: Amend the entries for WCET and WDT as follows:

Was:	WCET	Watchdog Timer
	WDT	Worst Case Execution Timing
Change to:	WCET	Worst Case Execution Timing
	WDT	Watchdog Timer

The European Organisation for Civil Aviation Equipment
L'Organisation Européenne pour l'Equipement de l'Aviation Civile

SUPPORTING INFORMATION FOR ED-12C AND ED-109A

ED-94C

January 2012

102 rue Etienne Dolet
92240 Malakoff, France
Web Site: www.eurocae.net

Tel: 33 1 40 92 79 30
Fax: 33 1 46 55 62 65
Email: eurocae@eurocae.net

SUPPORTING INFORMATION FOR ED-12C AND ED-109A

ED-94C

January 2012

FOREWORD

1. This document was prepared jointly by EUROCAE Working Group 71 (WG-71) and RTCA Special Committee 205 (SC-205), was approved by the Council of EUROCAE on 9 January 2012.

2. EUROCAE is an international non-profit making organisation in Europe. Membership is open to manufacturers and users of equipment for aeronautics, trade associations, national civil aviation administrations, and, under certain conditions, non-European organisations. Its work programme is principally directed to the preparation of performance specifications and guidance documents for civil aviation equipment, for adoption and use at European and world-wide levels.

3. The findings of EUROCAE are resolved after discussion amongst Members of EUROCAE and in collaboration with RTCA Inc, Washington D.C, through their appropriate committees.

4. EUROCAE performance specifications and other documents are recommendations only. EUROCAE is not an official body of the European Governments. Its recommendations are valid as statements of official policy only when adopted by a particular government or conference of governments.

5. Copies of this document may be obtained from:

EUROCAE

102, rue Etienne Dolet

92240 MALAKOFF

France

Tel: 33 1 40 92 79 30

Fax: 33 1 46 55 62 65

Email: eurocae@eurocae.net

Website: www.eurocae.net

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

ED-12C, “Software Considerations in Airborne Systems and Equipment Certification,” provides recommendations 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. ED-109A, “Software Integrity Assurance Considerations for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems,” provides similar guidance for the CNS/ATM ground-based community. ED-12C and ED-109A are based on ED-12B; therefore, many of the questions that were raised by the aviation community against ED-12B are also relevant to ED-12C and ED-109A.

This document addresses the questions of both the industry and authorities. It contains frequently asked questions (FAQs), discussion papers (DPs), and rationale. Many of the FAQs and DPs are based on ED-94B; however, some have been modified to address changes from ED-12B to ED-12C and to make applicable to ED-109A. Additionally, some new FAQs and DPs have been added to provide additional clarification on ED-12C and/or ED-109A. The errata against ED-12B (which were in section 2 of ED-94B) have been deleted, since they have been incorporated into ED-12C and are no longer relevant. The rationale for ED-12C and ED-109A objectives has also been added to ED-94C.

NOTE: *Prior to using this document, it is recommended that the reader consider section 1.3, “How to Use This Document”.*

1.1 PURPOSE

This document provides clarification of the guidance material in ED-12C and ED-109A. In order to accomplish this clarification the following products have been generated:

- Frequently Asked Question (FAQ) – Section 3: The purpose of a FAQ is to provide short and concise responses to questions that are frequently asked by industry concerning the material of ED-12C and/or ED-109A. These questions are frequently posed to certification authorities or others who provide interpretation of ED-12C and/or ED-109A. A FAQ contains no new guidance material. A FAQ is typically no longer than two pages.
- Discussion Paper (DP) – Section 4: The purpose of a Discussion Paper is to provide clarification for certain sections of ED-12C and/or ED-109A in cases where the clarification requires more than a short answer to a question. A DP contains no new guidance material.
- Rationale – Section 5: The purpose of the rationale is to document some items considered when developing ED-12B and then ED-12C and ED-109A. The rationale is only intended to be background information to aid the reader’s understanding of ED-12C and ED-109A — especially of those sections and objectives that industry feedback has shown might benefit from additional clarification. The rationale contains no guidance material.

1.2 DOCUMENT OVERVIEW

The products approved by EUROCAE Working Group 71 (WG-71) and RTCA Special Committee 205 (SC-205) are included in this document as follows:

- Section 3 includes the FAQs.
- Section 4 includes the DPs.
- Section 5 provides the rationale.
- Appendix A includes a list of all acronyms that are used throughout this document.
- Appendix B includes a list of the WG-71/SC-205 committee membership.
- Appendix C includes an index of keywords used in this document. The user may utilize this index to find information on related topics.
- Appendix D contains correlation table between ED-94C, ED-12C, and ED-109A.

1.3 HOW TO USE THIS DOCUMENT

Sections 3 and 4 are essentially chronological. ED-94 started out as an annual report, which was finalized with the publication of ED-94B. In order to keep the numbering for FAQs and DPs consistent with ED-94B, new FAQs were added to the end of section 3 and new DPs were added to the end of section 4. Several FAQs and DPs were deleted because the information is now clarified in ED-12C/ED-109A, in another FAQ or DP, or in one of the ED-12C/ED-109A supplements. The original titles of these FAQs and DPs are maintained for historical purposes. The rationale in section 5 is new to ED-94C. Because of the document structure, it is not recommended to read it from front to back. Instead the following approaches are recommended for using the document:

- Topical approach: If you are researching a specific topic, Appendix C provides an index of keywords. Each keyword is shown with the multiple sections that use that keyword.
- Reference approach: If you are researching a specific section in ED-12C/ED-109A, use Appendix D. Appendix D provides a mapping of the ED-12C/ED-109A sections to the related FAQs or DPs in this document.
- Hyperlink approach: Hyperlinks are provided from Appendices C and D to the FAQs and DPs in the main body in electronic versions of this document. You may use these hyperlinks to immediately go to the referenced section.

Major sections are numbered as X.0 throughout this document. It should be noted that references to an entire section are identified as “section X”; whereas, references to the content between section headers X.0 and X.1 are referenced as “section X.0”.

CHAPTER 2

ERRATA

At the time of this document's publication, no errata for ED-12C or ED-109A are known.

CHAPTER 3

FREQUENTLY ASKED QUESTIONS (FAQ)

This section compiles the ED-12C and ED-109A FAQs. The purpose of a FAQ is to provide short and concise responses to questions that are frequently asked by industry concerning the material of ED-12C and/or ED-109A. These questions are frequently posed to certification authorities or others who provide interpretation of ED-12C and/or ED-109A. A FAQ contains no guidance material.

Disclaimer: The responses to FAQs have been prepared and approved by Joint Committee WG-71/SC-205. These FAQs have no recognition by the certification authorities and are provided for information purposes only.

3.1 FAQ #1: SECTION 2 OF ED-12B PROVIDES AN INTRODUCTION TO THE SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT AND NOTES THAT GUIDELINES WERE UNDER DEVELOPMENT AT THE TIME OF WRITING. WHERE ARE THESE SYSTEM LIFE CYCLE GUIDELINES DOCUMENTED?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.2 FAQ #2: THROUGHOUT ED-12B REFERENCE IS MADE TO THE SYSTEM SAFETY ASSESSMENT PROCESS. WHERE CAN GUIDELINES FOR THIS PROCESS BE FOUND?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.3 FAQ #3: WHAT IS MEANT BY SAFETY MONITORING SOFTWARE EXPERIENCING TRANSIENTS IN ED-12B SECTION 2.3.2, PARAGRAPH 3?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.4 FAQ #4: DOES ED-12C/ED-109A'S DEFINITION OF COMMERCIAL OFF-THE-SHELF (COTS) SOFTWARE INCLUDE COTS SOFTWARE DESIGNED FOR OPTION-SELECTABLE SOFTWARE?

Reference: ED-12C: Sections 2.5.4, 4.2.h, 5.2.4, and 6.4.4.3.d

ED-109A: Sections 2.6.4, 4.2.h, 5.2.4, and 6.4.4.3.d

Keywords: commercial off-the-shelf software; COTS; option-selectable software

Answer:

The ED-12C glossary defines COTS software as: "*Commercially available applications sold by vendors through public catalog listings. COTS software is not intended to be customized or enhanced. Contract-negotiated software developed for a specific application is not COTS software.*"

ED-109A glossary defines COTS software slightly different from ED-12C to add clarification for CNS/ATM systems: "*Software under consideration for use in a CNS/ATM system that may have no or only partial evidence of compliance to this document sections 4 – 9 objectives.*"

As long as the COTS software is not changed as supplied from the vendor, it is still considered COTS software per the definitions above.

If the COTS software has option-selectable functionality, some deactivated code may exist. For an explanation of how to address deactivated code, see the ED-12C or ED-109A sections referenced above.

For all products, where the capability for the user to select options or functions is not provided by the vendor, and where the customer has to change the software, one needs to consider this software as previously developed software. This will not change the objectives to be fulfilled for certification/approval.

3.5 FAQ #5: WHAT ARE "END-TO-END CHECKS" IN THE CONTEXT OF FIELD-LOADABLE SOFTWARE?

Reference: ED-12C: Section 2.5.5.d

ED-109A: Section 2.6.5.d

Keywords: end-to-end checks; field-loadable software

Answer:

"End-to-end" checking means that the entire software load is verified – bit by bit. It starts with the production of the load media and ends with verification of the load being successfully completed. This includes the confirmation that displayed part numbers, etc. are shown to be correct.

Software that performs the loading, or integrity checking of the load, of field-loadable application software is typically operational when the system is not operational, for example, on the ground.

Proper system operation after the load is dependent on a correct load of the application software and the ability to verify the configuration (for example, a review of part numbers by maintenance personnel).

Integrity of the load can sometimes be assured without verification of the complex software responsible for this loading and checking to the same level as the software being loaded. This can be done via simple checks that are independent of the load checking software. These checks should be developed and verified to the level of the loaded software. These simple checks, referred to as end-to-end checks, generally use memory checksum, read/write verification, or cyclic redundancy checks (CRC) to confirm load correctness. Such checks are independent from the software that actually performed the load and from any associated part number display or checks.

3.6 FAQ #6: WHAT ARE THE DESIGN DESCRIPTION AND VERIFICATION ACTIVITY OBJECTIVES FOR A LEVEL D SYSTEM AND WHY ARE THERE APPARENT INCONSISTENCIES IN THE OBJECTIVES TO BE SATISFIED IN ANNEX A?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.7 FAQ #7: HOW CAN COMPLIANCE WITH ED-12C/ED-109A SECTION 5.2.3, DESIGNING FOR USER-MODIFIABLE SOFTWARE, BE OBTAINED?

Reference: ED-12C/ED-109A: Section 5.2.3

Keywords: non-modifiable software; user-modifiable software

Answer:

ED-12C/ED-109A section 5.2.3.b states: *"The means provided to change the modifiable component should be shown to be the only means by which the modifiable component can be changed."*

The assumption is made from the outset that compliance with ED-12C/ED-109A section 5.2.3.a is in place; that is, that the non-modifiable software is completely protected from interference from the user-modifiable portion, and that the conditions of ED-12C/ED-109A 5.2.3.a are also met. Further, the effects of the user-modifiable software should be shown to have no adverse effect on the overall system safety through the protection mechanisms employed, for example, during loading.

The issue now becomes one of how the change to the user-modifiable software is implemented. This is established through the applicant's defined means. The means can be represented in various ways (for example, defined tools, methodology, or rules). As long as the user adheres to these means, only the intended modification will result. For example, the applicant may define a set of design requirements for the user-modifiable component that is specific enough that, along with a defined implementation methodology, it constitutes the only means to implement the change. The applicant should show that the protection mechanisms around the modifiable component shield the non-modifiable component from unexpected changes. Caution should be exercised that a change in the user-modifiable software does not trigger a change in the non-modifiable software.

3.8 FAQ #8: CAN OPTION-SELECTABLE SOFTWARE CONTAIN DEACTIVATED CODE?

Reference: ED-12C: Sections 2.5.4, 4.2.h, 5.2.4, and 6.4.4.3.d

ED-109A: Sections 2.6.4, 4.2.h, 5.2.4, and 6.4.4.3.d

Keywords: deactivated code; option-selectable software

Answer:

Yes, please refer to ED-12C section 2.5.4 (ED-109A section 2.6.4) for details for description of option-selectable software, and ED-12C/ED-109A sections 4.2.h and 5.2.4 for designing for deactivated code.

3.9 FAQ #9: DO ALL HIGH-LEVEL REQUIREMENTS REQUIRE HARDWARE/SOFTWARE INTEGRATION TESTING? AND, WHAT DOES *"TO VERIFY THE INTERRELATIONSHIPS BETWEEN SOFTWARE REQUIREMENTS AND COMPONENTS"* MEAN?

Reference: ED-12C/ED-109A: Sections 6.4, 6.4.1, and 6.4.3

Keywords: hardware/software integration test; high-level requirements; integration; interrelationships; software integration test

Answer:

ED-12C/ED-109A section 6.4 defines hardware/software integration testing as a process that verifies the *"correct operation of the software in the target computer environment."*

ED-12C/ED-109A section 6.4.3.a states: "*Requirements-based hardware/software integration testing ensures that the software in the target computer will satisfy the high-level requirements.*"

So, the answer to the first question is "no": It is not required that all high-level requirements testing be executed on the target computer. Reference ED-12C/ED-109A section 6.4.1 that states that "*more than one test environment may be needed to satisfy the objectives for software testing.*" However, testing conducted on a host computer (non-target computer environment) may need justification for their results validity for the target computer system.

For the second question, see ED-12C/ED-109A section 6.4.3.b, which states: *"Requirements-based software integration testing ensures that the software components interact correctly with each other and satisfy the software requirements and software architecture."*

3.10 FAQ #10: ARE BASELINES ALLOWED TO BE CHANGED? SECTION 7.2.2.C STATES BASELINES SHOULD BE PROTECTED FROM CHANGE, WHEREAS SECTION 7.2.4.C TALKS ABOUT CHANGES TO BASELINES.

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.11 FAQ #11: IS THE "APPROVED SOURCE" IN SECTION 7.2.7.A OF ED-12B THE PREVIOUS APPROVED PRODUCT OR IS IT THE ORGANIZATION BUILDING THE PRODUCT?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.12 FAQ #12: WHAT ARE THE DEFINITIONS OF CONTROL CATEGORIES 1 AND 2 (CC1 AND CC2)?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.13 FAQ #13: HOW IS TABLE 7-1 IN SECTION 7.3 USED TO UNDERSTAND CONTROL CATEGORIES 1 AND 2 (CC1 AND CC2)?

Reference: ED-12C/ED-109A: Section 7.3

Keywords: configuration management; control categories; Control Category 1; CC1; Control Category 2; CC2

Answer:

ED-12C/ED-109A section 7.3 defines the configuration management objectives that should be satisfied for CC1 and for CC2. The entries in column 2 of table 7-1 are references to all the objectives of SCM that apply to software life cycle data, with the exceptions of Software Load Control and Software Life Cycle Environment Control. The black dots in column 3 of the table indicate that all the SCM objectives apply for data which should meet CC1 objectives. The black dots in column 4 indicate the subset of SCM objectives that apply for data which should meet CC2 objectives.

3.14 FAQ #14: WHAT DO CONTROL CATEGORIES 1 AND 2 (CC1 AND CC2) MEAN WHEN APPLIED TO THE OBJECTIVES OF ANNEX A?

Reference: ED-12C/ED-109A: Section 7.3 and Annex A

Keywords: configuration management; data control categories; Control Category 1; CC1; Control Category 2; CC2

Answer:

CC1 and CC2 define the processes and activities for handling the data that is generated to show that each Annex A objective has been achieved. CC1 and CC2 address configuration management (CM) issues. These issues are handled in different ways depending on the type of data.

The tables in Annex A of ED-12C/ED-109A specify which data control category (CC1 or CC2) applies by software level/assurance level for each item of software life cycle data.

The last two columns of ED-12C/ED-109A Table 7-1 define which objectives are applicable to CC1 and which objectives are applicable to CC2. In general, CC2 is a simpler method of control and is applied to data items based on the type of data and the software level/assurance level. Annex A defines the categorization for each type of data.

Not all data needs to be controlled using the same method to meet the CC1 and CC2 objectives. For example, Software Quality Assurance (SQA) Records may be controlled by placing them in a development library, whereas the product executable code may be controlled by an independent CM organization.

3.15 FAQ #15: IS SOFTWARE CERTIFIED AS A STAND-ALONE PRODUCT?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.16 FAQ #16: WHAT IS THE HIGHEST SOFTWARE LEVEL (PER ED-12C) OR ASSURANCE LEVEL (PER ED-109A) THAT CAN BE ATTAINED FOR PREVIOUSLY DEVELOPED SOFTWARE (PDS)?

Reference: ED-12C/ED-109A: Section 12.1

Keywords: assurance level; commercial off-the-shelf software; COTS; previously developed software; PDS; software level

Answer:

The highest software level/assurance level obtainable for previously developed software (PDS) can be Level A/AL1. For this to be achieved, the PDS should satisfy all the applicable objectives of ED-12C/ED-109A.

Business considerations of using PDS could prevent an applicant from reaching the highest level that is technically possible for that software.

3.17 FAQ #17: WHAT ARE THE ISSUES RELATED TO CHANGING PREVIOUSLY DEVELOPED SOFTWARE (PDS) VERSIONS FROM AN EARLIER BASELINE?

Reference: ED-12C/ED-109A: Sections 7.2.2, 7.2.4, and 12.1

Keywords: baseline; change; commercial off-the-shelf software; COTS; modification; previously developed software; PDS

Answer:

For this FAQ, commercial off-the-shelf (COTS) software is considered a subset of PDS.

The system with PDS in question may have been certified or approved using means other than ED-12C/ED-109A. Regulatory information exists to provide guidance for approval for software changes in legacy systems using ED-12C/ED-109A.

The effort to incorporate the changed PDS baseline should follow the same process as incorporating changes in any software product per the objectives of ED-12C/ED-109A. This includes, but is not limited to, an evaluation of the:

- Impact of PDS requirements changes on system requirements.
- Impact of the change on the system safety analysis.
- Impact of the change on the previously accepted certification data package to determine the scope of the change and to decide if the change should or can be accomplished.
- Impact of changes on software life cycle data.
- Impact to re-verification efforts.
- Process effort to implement and verify the PDS modification (if selected) into the system.

The assumption is made that the PDS has been previously incorporated into a certified system. The decision to change the baseline version of PDS is based on the same considerations used in the original selection of the PDS. A new accept or reject decision should be made before going forward with the change. It is important to gain access to the provider's Problem Reports that were incorporated into the change, as well as the revised software requirements (new functionality) as a roadmap to incorporation and re-verification of the new version of the system software. It is important to capture any information related to problems fixed in between versions. Any life cycle data pertaining to the PDS should be obtained. The Problem Reports may be provided, even for COTS software. The revised software requirements statement may be available through a list of new features or an updated user's manual.

Configuration considerations for PDS should ensure baseline and subsequent updates are controlled by the applicant. For example, all received copies of PDS with the same identification may not be identical, yet all delivered copies from the applicant should be identical.

Further information on PDS may be obtained using the keyword index (Appendix C).

3.18 FAQ #18: SINCE THERE IS NO SPECIFIC GUIDANCE FOR HANDLING CHANGES TO THE AIRCRAFT'S OPERATIONAL ENVIRONMENT, WHAT PART OF ED-12C ADDRESSES THIS TYPE OF CHANGE?

Reference: ED-12C: Sections 12.1.1 and 12.1.2

Keywords: aircraft installation; change; modification; operational environment; previously developed software; PDS

Answer:

ED-12C section 12.1.1 addresses modifications to previously developed software (PDS). ED-12C section 12.1.2 addresses the case when the aircraft installation changes. While a change in an aircraft's operating environment (for example, maximum altitude or maximum airspeed) is not strictly a change in the aircraft installation, ED-12C section 12.1.2 can be used to cover a change to the operating environment of an existing aircraft installation. Specifically, ED-12C section 12.1.2.b covers the functional modifications to the aircraft which may include changes to software requirements due to changes to the operating environment. When using s ED-12C section 12.1.2, it should be assumed that the term "aircraft installation" includes both the installation and the environment.

3.19 FAQ #19: HOW DOES ONE DETERMINE IF IN-SERVICE PROBLEMS INDICATE AN INADEQUATE PROCESS, AND CAN ONE CONTINUE TO PURSUE A SERVICE HISTORY MEANS OF COMPLIANCE WITH SOME PROCESS INADEQUACIES?

This FAQ has been removed, as approved by the WG-71/SC-205 committee. The information has been incorporated into DP #4.

3.20 FAQ #20: WHAT IS THE SOURCE OF THE GLOSSARY OF TERMS IN ED-12C, ED-109A, AND THE SUPPLEMENTS, AND WHY DO THEY APPEAR TO BE DIFFERENT FROM OTHER STANDARD DEFINITIONS?

Reference: ED-12C/ED-109A: Annex B and Section 1.4

Keywords: definitions; glossary; terms

Answer:

Where possible, ED-12C/ED-109A uses standard definitions for the terms used in the document. These definitions are defined in industry standards such as Institute of Electrical and Electronic Engineers (IEEE) and Society of Automotive Engineers (SAE). Additionally, terms from European Aviation Safety Agency (EASA) and Federal Aviation Administration (FAA) documents are used.

The ED-12C/ED-109A glossary (Annex B) defines some terms more precisely or narrowly, where a special interpretation is required by the aviation community and in order to prevent ambiguity during the interpretation of the document. However, it was a goal of the joint committee developing the document to minimize the use of special terms or unique definitions in ED-12C. The ED-109A glossary stands alone from ED-12C (that is, ED-109A has its own complete glossary). It started with the ED-12C glossary and modified or added terms to address CNS/ATM non-airborne software development.

The ED-12C/ED-109A supplements also include glossaries. These glossaries contain terms which are unique to the technology discussed in the specific supplement. For terms that are not technology specific, the ED-12C/ED-109A glossary serves as the glossary for the supplements.

3.21 FAQ #21: HOW IS THE SECOND SENTENCE OF THE DEFINITION OF "PATCH" IN ED-12C/ED-109A ANNEX B RELEVANT TO THE DEFINITION ITSELF?

Reference: ED-12C/ED-109A: Annex B

Keywords: embedded identifiers; modification; patch

Answer:

ED-12C/ED-109A Annex B defines "patch" as: *"Modification to Executable Object Code, in which one or more of the planned steps of re-compiling, re-assembling, or re-linking is bypassed. This does not include embedded identifiers."*

The second sentence of this definition is intended to show recognized exceptions to the normal case. Typically, embedded identifiers are values such as checksums or part numbers. In cases where the values cannot be determined at compile time, memory is allocated and values are inserted during or after linking. This type of modification is not considered a patch since it is not modifying the executable code. The second sentence provides this distinction.

3.22 FAQ #22: CAN VARIOUS INDUSTRY PROCESS ASSESSMENTS SUCH AS THE SOFTWARE ENGINEERING INSTITUTE (SEI) CAPABILITY MATURITY MODEL INTEGRATION (CMMI), SOFTWARE PROCESS IMPROVEMENT CAPABILITY EVALUATION (SPICE), ETC. BE USED FOR CERTIFICATION CREDIT?

Reference: ED-12C/ED-109A: Sections: 1.3, 9.1, 11, and Annex A

Keywords: Capability Maturity Model Integration; CMMI; process assessment; Software Engineering Institute; SEI; Software Process Improvement Capability Evaluation; SPICE

Answer:

The simple answer is that an applicant cannot claim certification "credit" for the use of the SEI CMMI, SPICE, etc.

Although ED-12C/ED-109A and its predecessors advocate good process characteristics, process objectives are given primarily in terms of attributes of the process data and the associated product. Many of the ED-12C/ED-109A objectives specifically relate to completeness and rigor of development and verification processes in specific terms, whereas the generalized process models, such as the SEI CMMI and SPICE, are more focused on business process measurement and improvement. They stipulate and assess against many process attributes that are necessary, but not sufficient conditions, for high product integrity and safety. For example, the SEI CMMI has no process activities or capabilities or metrics associated with verification (review, analysis, or test) coverage, whereas this is a very significant aspect of ED-12C/ED-109A guidance, because verification is a key factor in system safety.

Although there is some overlap, ED-12C/ED-109A remains the more comprehensive guidance for airborne and CNS/ATM systems. Applicants seeking process recognition for the use of the SEI CMMI, SPICE, etc. may readily reuse the same process compliance data to substantiate compliance with ED-12C/ED-109A objectives. Such data would still be required to meet the software life cycle data attributes of ED-12C/ED-109A section 11 and the objectives of ED-12C/ED-109A Annex A with respect to independence and configuration control.

3.23 FAQ #23: IS SOFTWARE RELIABILITY ADDRESSED BY ED-12C/ED-109A?

Reference: ED-12C/ED-109A: Sections 2.3 and 12.3.3

Keywords: software reliability

Answer:

ED-12C/ED-109A does mention software reliability; however, the concept of using numerical reliability figures for software is dismissed in ED-12C/ED-109A by making the following observations:

- ED-12C/ED-109A section 2.3 states that a software level/assurance level cannot be interpreted as a failure rate.
- ED-12C/ED-109A section 12.3.3 indicates that software reliability analysis based on development metrics does not provide sufficient confidence.

3.24 FAQ #24: WHAT IS THE RELATIONSHIP BETWEEN ED-79A/ARP4754A AND ED-12C?

Reference: ED-12C: Section 2

Keywords: ARP4754A; ED-79A; hardware development; system development processes; system safety assessment; SSA; system safety assessment process

Answer:

EUROCAE document ED-79A or Society of Automotive Engineers (SAE) Aerospace Recommended Practice ARP4754A, “Guidelines for Development of Civil Aircraft and Systems,” was produced to provide guidance at the system level for systems and equipment that implement aircraft-level functions and to provide links to specific related disciplines, such as the software development life cycle. The relationships between the system safety assessment process, hardware development process, and software development process are discussed in ED-12C section 2. The following diagram illustrates these relationships.

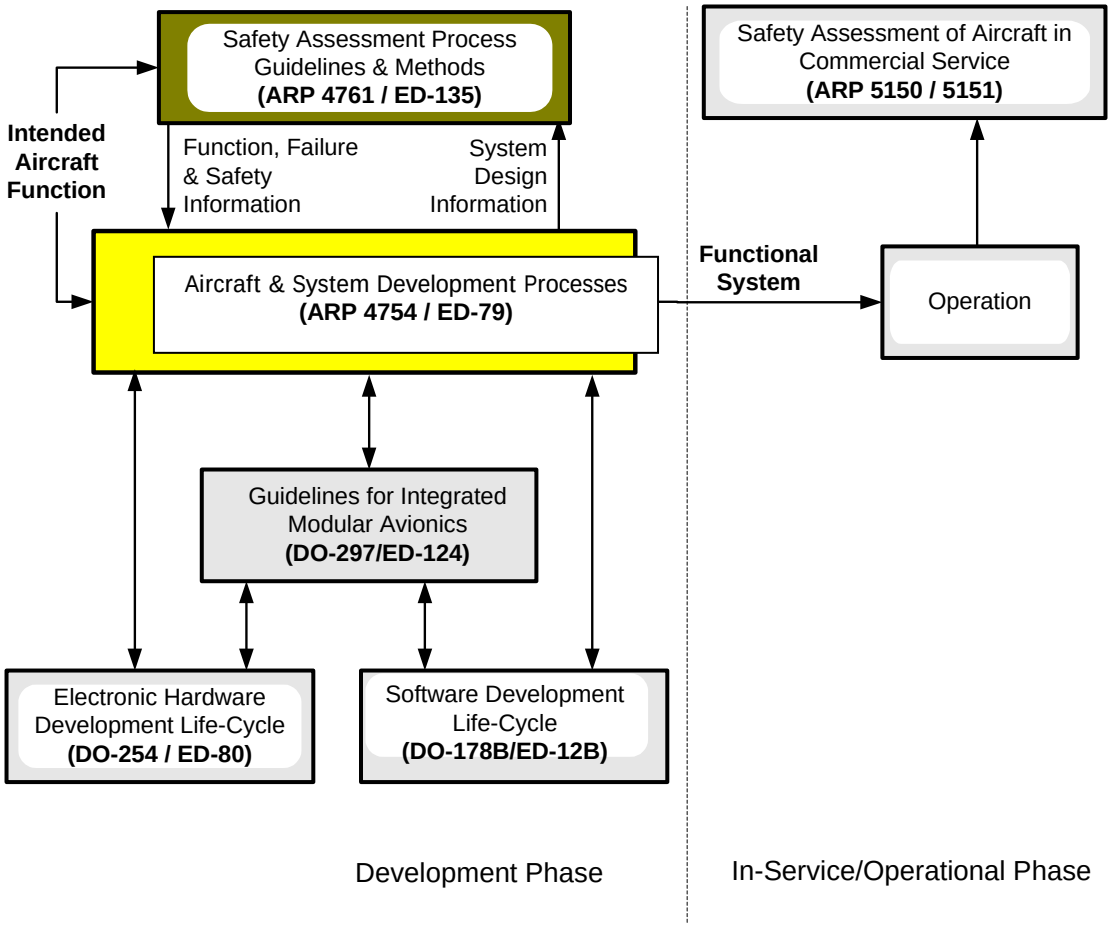


FIGURE 3-1: ED-79/ARP4754 RELATIONSHIP TO OTHER DOCUMENTS

3.25 FAQ #25: CAN ARCHITECTURAL MEANS BE USED TO REDUCE THE SOFTWARE LEVEL/ASSURANCE LEVEL NEEDED FOR THE INCORPORATION OF PREVIOUSLY DEVELOPED SOFTWARE (PDS) IN A SYSTEM?

Reference: ED-12C/ED-109A: Section 2.4

Keywords: architectural means; architecture; assurance level; commercial off-the-shelf; COTS; partitioning; previously developed software; PDS; software level

Answer:

Yes, architectural means can be used to reduce the software level/assurance level needed by mitigating all identified risks that could result from anomalous behavior of the PDS. The methods discussed in ED-12C/ED-109A section 2.4 are applicable to all types of software including PDS. The required justification associated with these methods (such as analysis with respect to design independence, etc.) is also the same for PDS as for any other software. If sufficient justification data cannot be obtained, this may limit the extent to which the architectural means may be used. Methods for implementing architectural means discussed in EUROCAE ED-79A (and SAE ARP4754A) and ED-12C/ED-109A are also potentially applicable to PDS. Some of the architectural means referenced in these documents are listed below:

- Partitioning.
- Safety monitoring.
- Multiple-version dissimilar software (with use of monitors, comparators, and polling) may be used for redundancy or backup.

Assurance needed to implement these methods is also discussed in ED-12C/ED-109A section 2.4. Further considerations in the implementation of these methods include but are not restricted to:

- Whether the resulting function is a primary or a secondary function.
- Probability of common mode failures.
- Analysis of common cause errors.

Depending upon the resultant architecture, more assurance of the architectural means may be needed than that of the PDS.

Note also that restriction of functionality may be used in the case of PDS. In general, many PDS products include a variety of functions that might not be needed for any given application and thus it might be necessary to subject all of them to the same degree of scrutiny unless unneeded functions are restricted. There should be an analysis provided to show why and how unneeded code is deactivated. Restriction of functionality may result in deactivated code in the resultant PDS component. Additionally, PDS may also contain extraneous or dead code. All objectives regarding deactivated, extraneous, and dead code also apply to PDS.

3.26 FAQ #26: DOES THE FULFILLMENT OF "INDEPENDENCE OF MULTIPLE-VERSION DISSIMILAR SOFTWARE" (ED-12B SECTION 12.3.3.1) SUPERSEDE THE INDEPENDENCE REQUIREMENTS AS DEFINED IN ANNEX A OF ED-12B?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.27 FAQ #27: WHAT IS MEANT BY "USER-MODIFIABLE SOFTWARE"?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.28 FAQ #28: WHAT IS THE VALUE OF REMOVING DEAD CODE OR UNUSED VARIABLES?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.29 FAQ #29: WHAT DOES ED-12C SECTION 2.5.5.B (ED-109A SECTION 2.6.5.B) MEAN, WHEN IT ADDRESSES REQUIREMENTS RELATED TO A DEFAULT MODE, IF ONE IS PROVIDED TO PROTECT AGAINST SOFTWARE LOAD ERRORS?

Reference: ED-12C: Section 2.5.5.b

ED-109A: Section 2.6.5.b

Keywords: default mode; load

Answer:

ED-12C section 2.5.5.b (ED-109A section 2.6.5.b) states: "*If a system recovers to a default mode or safe state upon detection of a corrupted or inappropriate software load, then each partitioned component of the system should have safety-related requirements specified for recovery to and operation in this mode. Interfacing systems may also need to be reviewed for proper operation with the default mode.*"

When loading new software into a system, a number of problems can arise including interrupted loads and incorrect software loads. In most cases, the system designer will have considered these problems and arrived at a mechanism for ensuring a known system state can be established.

ED-12C Section 2.5.5.b (ED-109A section 2.6.5.b) is simply saying that the requirements for recovering to this known state have been captured. These requirements are, by definition, considered safety-related requirements since they establish the mechanism by which the system can be returned to a safe state and that the problem encountered during loading is contained. Note that interfacing systems may need to be reviewed for proper operation with the default mode.

3.30 FAQ #30: WHAT DOES ED-12B SECTION 2.6A(2) MEAN REGARDING SYSTEM SAFETY REQUIREMENTS ADDRESSING SYSTEM ANOMALOUS BEHAVIOR?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.31 FAQ #31: HOW DOES VERIFICATION OF PRODUCT RELATE TO "COMPILER ACCEPTABILITY"?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.32 FAQ #32: WHAT ARE DEFENSIVE PROGRAMMING PRACTICES?

Reference: ED-12C/ED-109A: Sections 1.4 and 4.5

Keywords: defensive programming; software development standards

Answer:

Note 1 of ED-12C/ED-109A section 4.5 mentions defensive programming when it states, "*Defensive Programming practices may be considered to improve robustness.*" Section 1.4.m of ED-12C/ED-109A states that notes are intended to "*provide explanatory material, emphasize a point, or draw attention to related items that are not entirely within context.*"

Defensive programming practices are techniques that may be used to prevent code from executing unintended or unpredictable operations by constraining the use of structures, constructs, and practices that have the potential to introduce failures in operation and errors into the code.

Defensive programming is related to the robustness of the software, not its correctness or maintainability. It should be acknowledged that "good" programming practice will promote robust software and hence could be considered defensive programming. Additionally, it should be noted that the robustness of the software should be defined by software requirements specifying the correct software response to abnormal conditions and inputs. Defensive programming may be considered in the design and coding standards.

NOTE: *Reference the Object-Oriented Technology and Related Techniques supplement for related techniques that may provide supporting information.*

Some examples of defensive programming are included below (this is not an exhaustive list):

a. Avoidance of input data errors: To minimize input data errors:

   1. Check errors or bounds in data input whether it is from a human user or from another system.
   2. Use reasonableness checks on the incoming data.
   3. Account for potential data buffer overflow conditions (for example, queuing controls) and data aging (for example, time-stamps).
   4. Account for data type coercion, scaling, and loss of precision or accuracy that may occur in the interface between systems.
   5. Include run-time checks that allow detection of errors (for example, due to incorrect implementation).

b. Avoidance of non-determinism: Non-determinism may be reduced through one or more of the following practices:

   1. Choose a language with a well-defined standard.
   2. Do not permit self-modifying code.
   3. Minimize memory paging and swapping.
   4. Avoid use of dynamic binding.

5. Do not assume initial values for variables that are not explicitly initialized.

6. Avoid use of memory allocation and deallocation or when applicable, use the guidance of the Object Oriented Technologies supplement.

c. Avoidance of complexity: Complexity may be reduced through the following practices:

1. Maximize cohesion and minimize coupling.

2. Use overloading of operators and variable names cautiously.

3. Use a single point of entry and exit for subprograms.

4. Minimize use of interrupt-driven processing.

5. Minimize use of multi-tasking and multi-processing in the architecture.

6. Beware of side effects in the use of built-in functions and compiled libraries.

d. Avoidance of interface errors: Interface errors may be minimized through the following practices:

1. Minimize the complexity of interfaces (for example, a large number of arguments, cryptic names, etc.).

2. Use a standard set of units and precision throughout the software application.

3. Avoid type coercion (for example, implicit or automated type conversions, mixed mode operations, fixed point scaling, etc.) in interfaces between procedures or follow the guidance of the Object Oriented Technologies supplement.

4. Restrict use of global variables.

e. Avoidance of logical errors: To minimize the potential for inducing these errors:

1. Understand possible loss of accuracy during computation.

2. Be mindful of conversion errors (for example, fixed point scaling).

3. Avoid using expressions with side effects such as assignment statements within evaluation statements.

4. Verify the proper implementation of computed indices and array-bounds.

5. Handle exceptions locally and uniformly.

3.33 FAQ #33: IS IT PERMISSIBLE TO NOT MEET THE SAFETY OBJECTIVES BY JUSTIFYING ANY DEVIATIONS FROM THE DESIGN STANDARDS?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.34 FAQ #34: WHAT IS THE CONCEPT OF INDEPENDENCE AS USED IN ED-12B?

This FAQ has been removed, as approved by the WG-71/SC-205 committee. The information has been incorporated into DP #19.

3.35 FAQ #35: WHAT ARE LOW-LEVEL REQUIREMENTS AND HOW MAY THEY BE TESTED?

Reference: ED-12C/ED-109A: Sections 5.0, 5.5, 6.0, 6.5, and Annex A

Keywords: architecture; derived requirements; high-level requirements; low-level requirements; requirements-based testing; testing; traceability

Answer:

The software requirements for a particular project are developed from system requirements, hardware architecture-dependent software requirements, and any additional derived requirements.

"During the software design process, the software architecture is defined and low-level requirements are developed" (reference ED-12C/ED-109A section 5.0). The aim of the architectural design is to break down the functions of the software requirements so that they can be regrouped into a logical software structure.

Once the architectural design is completed, the detailed design of the individual software functions is initiated. During this process, low-level requirements and any additional derived requirements are developed. These requirements are those from which the Source Code will be generated. "*However, if Source Code is generated directly from high-level requirements, then the high-level requirements are also considered low-level requirements…"* (reference ED-12C/ED-109A section 5.0). In order to enable the verification of complete implementation of the requirements, to give visibility to derived requirements, and to support the requirements-based test coverage analysis, Trace Data should be developed (reference ED-12C/ED-109A sections 5.5 and 6.5).

Compliance of the executable object code to the low-level requirements may be demonstrated using requirements-based tests. Test cases are developed for all levels of requirements. Then the requirements coverage analysis assesses the coverage of all requirements by the test cases.

3.36 FAQ #36: WHAT IS THE DEFINITION OR INTERPRETATION OF DERIVED REQUIREMENTS IN ED-12C/ED-109A?

Reference: ED-12C/ED-109A: Section 5.0 and Annex B (glossary)

Keywords: derived requirements; system safety assessment process

Answer:

The ED-12C/ED-109A glossary defines derived requirements as: "*Requirements produced by the software development processes which (a) are not directly traceable to higher level requirements, and/or (b) specify behavior beyond that specified by the system requirements or the higher level software requirements*."

During the software development process, requirements may be developed that do not trace directly to higher level requirements. Development of these derived requirements is typically based upon design, performance, or architectural decisions and therefore not directly traceable to higher level requirements. Even though a derived requirement may not be traced to higher level requirements, the derived requirement may affect the higher level requirements. ED-12C/ED-109A section 5 gives the following example: "*the need for interrupt handling software to be developed for the chosen target computer*." In this example, the need for interrupt handling software will not appear in the higher level requirements, but it will be specified in the lower level requirements as derived requirements.

During the software development process, requirements may be developed that specify additional behavior that is not specified by the system requirements or higher level requirements. ED-12C/ED-109A section 5 gives the following example: "*The addition of scaling limits when using fixed point arithmetic*." While the arithmetic to be performed would be specified in requirement(s), the use of fixed point arithmetic and the scaling limits on the fixed point arithmetic would be specified in a derived requirement if not specified in the higher level requirement(s). This derived requirement may be considered traceable to the higher level requirement(s) but it does add behavior that limits the values that may be used in the arithmetic operations. These limits need to be made "*available to the system processes including the system safety assessment process*."

3.37 FAQ #37: WHAT IS MEANT BY PROVIDING DERIVED SOFTWARE REQUIREMENTS TO THE SYSTEM PROCESSES, INCLUDING THE SYSTEM SAFETY ASSESSMENT PROCESS?

Reference: ED-12C/ED-109A: Sections 2.2.1.g, 2.2.2.a, 2.2.3.a, 5.1.1.b, 5.1.2.i, 5.2.1.b, 5.2.2.c, and 11.6.d

Keywords: derived requirements; system safety assessment process

Answer:

All derived requirements are provided to the system processes, including the system safety assessment process, to ensure that the implementation of the derived requirements is visible to the system processes and that it does not adversely affect system safety (for example, produce a new hazard). The applicant should have a means for communicating the derived requirements to the system processes, including the system safety assessment process. The method that the applicant chooses to use for communicating the derived requirements may be defined by the applicant's Software Requirements Standards, Software Design Standards, and/or Software Development Plan.

3.38 FAQ #38: WHAT IS THE DIFFERENCE BETWEEN INTEGRATION PROCESS AND INTEGRATION TESTING?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.39 FAQ #39: WHAT IS THE DEFINITION OF AN UNBOUNDED RECURSIVE ALGORITHM IN ED-12C/ED-109A SECTION 6.3.3.D?

Reference: ED-12C: Sections 6.3.3.d and 6.3.4.f and Annex B

ED-109A: Sections 6.3.3.d, 6.3.4.f, 12.4.11.2.4, and Annex B

Keywords: recursion; unbounded recursive algorithm

Answer:

An unbounded recursive algorithm is an algorithm that directly invokes itself (self-recursion) or indirectly invokes itself (mutual recursion) and does not have a mechanism to limit the number of times it can do this before completing (see ED-12C/ED-109A Annex B glossary).

The motivation for this FAQ is a consequence of the question: “Can recursive algorithms be used in airborne applications?” The answer to this question is “yes.” However, if a recursive algorithm is used, the algorithm should be shown to be bounded (see ED-12C/ED-109A section 6.3.3.d). There are two reasons for this. The first is that the maximum execution time of the algorithm should be able to be determined, so that the execution of the algorithm can be shown to be within the real-time allocation for the function performed by the algorithm. Thus, a mechanism to establish an upper bound on the number of recursive calls is needed. Secondly, the stack space allocated to support recursion is limited by the actual stack space that is available. It should be shown that the stack space is available to handle the maximum level of recursion. Thus, the restriction of available stack space also requires that the recursive algorithm be bounded (see ED-12C/ED-109A section 6.3.4.f).

3.40 FAQ #40: WHAT REPRESENTATION OF THE SOFTWARE IS USED TO PERFORM REVIEWS AND ANALYSES OF SOURCE CODE?

Reference: ED-12C/ED-109A: Section 6.3.4

Keywords: Source Code analysis; Source Code review

Answer:

When performing a review and analysis of the Source Code, the software should be reviewed in the representation as written in the programming language. It may be necessary to include other programming-related items (for example, linker directives, make files, etc.) in the review, as well as the other inputs to a software review in order to determine that the review objectives are satisfied. Source Code review and/or analysis is not concerned with examining the output of compilers and linkers/assemblers which are verified by other means.

3.41 FAQ #41: WHY IS SOURCE CODE TO OBJECT CODE TRACEABILITY REQUIRED FOR LEVEL A SOFTWARE?

This FAQ has been removed and is superseded by DP #12 (in section 4.12 of this document).

3.42 FAQ #42: WHAT NEEDS TO BE CONSIDERED WHEN PERFORMING STRUCTURAL COVERAGE AT THE OBJECT CODE LEVEL?

Reference: ED-12C/ED-109A: Sections 4.5, 6.4.4.2, and 12.2

Keywords: object code; object code coverage; OCC; Source Code; structural coverage; tool qualification

Answer:

The main consideration is to demonstrate that the coverage analysis conducted at the object code level will provide the same level of confidence as that conducted at the Source Code level.

a. To achieve this same level of confidence, the analysis needs to confirm the following, as a minimum:

   1. How the compiler produces the object code expected as assumed in the object code coverage (OCC) approach (for example, short-circuit behavior, compiler assumptions, compiler options and optimizations, etc.).

   2. How compiler options and optimizations impact the coverage achieved.

   3. How the results achieved from the OCC method provide the appropriate coverage assurance.

4. How each of the following is addressed:
   - Control from outside to inside a function.
   - Jump statements (for example, break, continue, return, goto).
   - Bitwise operators, for example if Software Code Standards prohibit their use inside decisions.
5. The need for qualification of tools used in OCC approach, for example, special test tools or in-circuit hardware devices.
6. If necessary, specific design and code rules to enforce this same level of confidence are included in the Software Design Standards and Software Code Standards.

b. Plans and standards need to support the analysis explained above and should address the following:
   1. Grammar rules, coding restrictions and limitations, compiler restrictions and limitations, complexity limitations, etc. that are necessary to ensure that OCC will provide the appropriate structural coverage.
   2. Any complexity and software architecture limitations for ensuring the validity of the OCC approach (for example, number of nested if statements, number of conditions in a decision, nested function calls, etc.).
   3. Compliance with the rules, restrictions, and limitations needed for the OCC approach.

3.43 FAQ #43: WHAT IS THE INTENT OF STRUCTURAL COVERAGE ANALYSIS?

Reference: ED-12C/ED-109A: Sections 6.4.4.2 and 6.4.4.3; Figure 6-1; and Annex A Table A-7

Keywords: intended function; structural coverage; verification

Answer:

ED-12C/ED-109A sections 6.4.4.2 and 6.4.4.3 define the structural coverage analysis activities and the possible resolution for code structure that was not exercised during requirements-based testing.

The purpose of structural coverage analysis, along with the associated structural coverage analysis resolution, is to complement requirements-based testing with the following:

1. Provide evidence that the code structure was verified to the degree required for the applicable software level.
2. Provide a means to support demonstration of absence of unintended functions.
3. Establish the thoroughness of requirements-based testing.

With respect to intended function, evidence that testing was rigorous and complete is provided by the combination of requirements-based testing (both normal range testing and robustness testing) and requirements-based test coverage analysis.

Requirements-based testing alone cannot completely verify the absence of unintended functionality. Code that is implemented without being linked to requirements may not be exercised by requirements-based tests. Such code could result in unintended functionality. Therefore, something additional needs to be done since unintended functions could affect safety. A technically feasible solution was found in structural coverage analysis.

The rationale is that if requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there is no unintended functionality. Structural coverage analysis will:

- indicate to what extent the requirements-based test procedures exercise the code structure, and
- reveal code structure that was not exercised during testing.

NOTE: *In the above text, the term "exercised during requirements-based testing" does not only mean that the specific code was exercised. It also means that the behavior of the code has been verified with respect to the requirements.*

3.44 FAQ #44: WHY IS STRUCTURAL TESTING NOT A ED-12C/ED-109A REQUIREMENT?

Reference: ED-12C/ED-109A: Sections 6.4.4.2 and 6.4.4.3

Keywords: structural coverage; structural testing

Answer:

There is a distinction between structural coverage analysis and structural testing. Whereas structural testing is the process of exercising software with test scenarios written from the Source Code, ED-12C/ED-109A section 6.4.4.2 explains that structural coverage analysis "*determines which code structure, including interfaces between components, was not exercised by the requirements-based test procedures.*" Structural testing does not satisfy the structural coverage analysis objectives.

The correct approach when structural coverage analysis identifies untested code is to consider the possible causes in accordance with ED-12C/ED-109A section 6.4.4.3, Structural Coverage Analysis Resolution. If any additional testing is required, it should be requirements-based testing, using high-level, low-level, or derived requirements, as appropriate.

Structural testing verifies that the Executable Object Code complies with the Source Code. Since the starting point for developing structural test cases is the code itself, there is no way of finding requirements (high-level, low-level, or derived) not implemented in the code through structural tests. It is a natural tendency to consider outputs of the actual code (which is de facto the reference for structural testing) as the expected results. This bias is avoided when expected outputs of a tested piece of code are determined by analysis of the requirements.

Since the code itself is used as the basis of the test cases, structural testing may fail to find simple coding errors.

3.45 FAQ #45: WHAT IS THE RELEVANCE OF THE EXCEPTION CASE STATED IN THE DEFINITION OF DEAD CODE?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.46 FAQ #46: WHAT IS THE MEANING OF SECTION 7.2.2.G IN ED-12C/ED-109A, WHEN IT STATES THAT A BASELINE OR CONFIGURATION ITEM SHOULD BE TRACEABLE EITHER TO THE OUTPUT IT IDENTIFIES OR TO THE PROCESS WITH WHICH IT IS ASSOCIATED?

Reference: ED-12C/ED-109A: Section 7.2.2.g

Keywords: baseline; configuration item; traceable

Answer:

The first step in understanding the meaning of this sentence is to realize that ED-12C/ED-109A glossary defines a baseline as one or more configuration items. A configuration item is one or more software components or software life cycle data treated as a unit for software configuration management purposes. ED-12C/ED-109A section 7.2.2.g states that a configuration item should be traceable to at least one of two items: (1) "*the output it identifies*", or (2) "*the process with which it is associated*". Both are described below.

The "*output it identifies*" is applicable for configuration items that are used as inputs. For example, the Software Requirements Data, the Software Development Plan, and the Software Design Standards are all inputs to the software design process. Therefore, these configuration items (the Software Requirements Data, Software Development Plan, and Software Design Standards) should be traceable to the "*output it identifies*," namely, the Design Description.

The "*process with which it is associated*" is applicable for configuration items that are outputs. For example, the Design Description is the output of the software design process. The software design process uses the Software Requirements Data, the Software Development Plan, and software development standards as inputs. Therefore, the output configuration item, the Design Description, should be traceable to the "*process with which it is associated*," the software design process, by its associated software life cycle data: the Software Requirements Data, the Software Development Plan, and the software development standards.

3.47 FAQ #47: WHAT IS MEANT BY THE TERM "CERTIFICATION CREDIT" OR "APPROVAL CREDIT"?

Reference: ED-12C/ED-109A: Sections 8.3.a and Annex B

Keywords: approval credit; certification credit; evidence of compliance; means of compliance

Answer:

ED-12C and ED-109A use different terminology; therefore, each is discussed separately below.

ED-12C "certification credit":

The ED-12C Annex B glossary defines "certification credit" as: "*Acceptance by the certification authority that a process, product, or demonstration satisfies a certification requirement*."

For software, the applicant proposes their means of compliance for satisfying the software aspects of the certification requirements in the Plan for Software Aspects of Certification (PSAC). In other words, the applicant proposes processes, activities, methods, tools, etc. for which they are seeking "credit" for satisfying the software aspects of certification. Then, to achieve credit, the applicant performs the processes, satisfies the objectives, provides demonstrations, and produces the evidence of compliance, which are summarized in the Software Accomplishment Summary (SAS). Since certification credit is provided by "*acceptance by the certification authority*," the certification authority's approval of the SAS, design data, test data, etc. (that is, evidence of compliance) constitutes the granting of certification credit for the software.

NOTE: *The applicant may conduct other processes and activities whose results are not submitted for certification credit.*

ED-109A "approval credit":

The ED-109A Annex B glossary defines "approval credit" as: *"Acceptance by the approval authority that a process, software product, or demonstration satisfies an applicable requirement."*

For software, the applicant proposes their means of compliance for satisfying the software aspects of the approval requirements in the Plan for Software Aspects of Approval (PSAA). In other words, the applicant proposes processes, activities, methods, tools, etc. for which they are seeking "credit" for satisfying the software aspects of approval. Then, to achieve credit, the applicant performs the processes, satisfies the objectives, provides demonstrations, and produces the evidence of compliance, which are summarized in the Software Accomplishment Summary (SAS). Since approval credit is provided by "*acceptance by the approval authority*," the approval authority's approval of the SAS, design data, test data, etc. (that is, evidence of compliance) constitutes the granting of approval credit for the software.

NOTE: *The applicant may conduct other processes and activities whose results are not submitted for approval credit.*

3.48 FAQ #48: IN ADDITION TO SECTIONS 9 AND 10 OF ED-12C, WHICH PROVIDE A HIGH-LEVEL OVERVIEW OF THE CERTIFICATION LIAISON AND CERTIFICATION PROCESSES, WHERE CAN FURTHER GUIDANCE ON THE APPROVAL PROCESS FOR SOFTWARE BE FOUND?

Reference: ED-12C: Sections 9 and 10

Keywords: approval authority; approval process; certification authority; certification basis; certification liaison; certification processes; guidance

Answer:

Further guidance, information and appropriate links for issues related to the certification processes may be found by searching the web-sites of the Federal Aviation Administration (FAA) (http://www.faa.gov), the European Aviation Safety Agency (EASA) (http://www.easa.europa.eu), or the reader's local certification or approval authority. Additional information (for example, policy and guidance materials such as Advisory Circulars, Orders, and Notices) may also be found through these web-sites. Additional information, such as the certification basis for a particular aircraft type, should be requested from the applicant's local certification authority.

NOTE: *Web-site addresses occasionally change. If the above web-sites do not connect, perform a web search or contact your local certification authority for an address update.*

3.49 FAQ #49: WHERE CAN CURRENT CERTIFICATION AUTHORITY GUIDANCE REGARDING ISSUES NOT COVERED IN ED-12B OR EXPANDING UPON ISSUES IN ED-12B BE FOUND?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.50 FAQ #50: WHAT DATA ITEMS ARE DELIVERABLE TO THE CERTIFICATION/APPROVAL AUTHORITY TO SUPPORT SOFTWARE APPROVAL AND PRODUCT CERTIFICATION/APPROVAL?

Reference: ED-12C/ED-109A: Sections 9.3 and 11

Keywords: approval authority; certification authority; data item; software life cycle data

Answer:

ED-12C/ED-109A section 9.3 specifies that the minimum set of software life cycle data submitted to the certification/approval authority includes the Plan for Software Aspects of Certification/Plan for Software Aspects of Approval, the Software Configuration Index, and the Software Accomplishment Summary. The certification/approval authority may, at their discretion, request additional software life cycle data items. Such a decision to request additional software life cycle data items is normally based upon the software level/assurance level of the system and the certification/approval authority's knowledge of the developing organization's life cycle processes.

Software data is only a part of the data needed to be submitted by the applicant for "product certification" or "product approval". Other data might include: system certification/approval plans, system validation and verification plans, flight test plans, safety analyses, system type design data, verification and validation results, flight test results, etc.

3.51 FAQ #51: WHAT IS MEANT BY THE TERM "TYPE DESIGN," AS USED IN SECTION 9.4 OF ED-12B?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.52 FAQ #52: WHY DO THE CERTIFICATION/APPROVAL AUTHORITIES NOT APPROVE AN ORGANIZATION'S PROCESS ONCE, RATHER THAN APPROVE EACH PRODUCT SUBMITTED AS PART OF A CERTIFICATION/APPROVAL APPLICATION?

Reference: ED-12C/ED-109A: Section 10.3

Keywords: process approval; product

Answer:

The regulatory responsibility of the certification/approval authority is to approve the product. However, the certification/approval authority may, at their discretion, adjust the level of involvement in the review and approval process associated with a particular application. This adjustment is typically based on the certification/approval authority's previous experience with the applicant. Therefore, applicants lacking extensive experience may be subjected to greater certification/approval authority involvement than more experienced applicants.

3.53 FAQ #53: DO THE DATA ITEMS NEED TO BE PREPARED AND PACKAGED AS SPECIFIED IN SECTION 11 OF ED-12B?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.54 FAQ #54: IS THE DOCUMENTATION REQUIRED IN ED-12C/ED-109A SECTION 11 EXCESSIVE, ESPECIALLY FOR SMALL PROJECTS?

Reference: ED-12C/ED-109A: Section 11; Annex A

Keywords: documentation; small projects

Answer:

It is not the software's size but its impact on safety (and thereby the objectives to be met) that determine the type and quantity of data to be produced.

3.55 FAQ #55: WHAT ARE THE CONTROL CATEGORY CONSIDERATIONS WHEN DETERMINING HOW TO PACKAGE THE DATA ITEMS DISCUSSED IN SECTION 11 OF ED-12C/ED-109A?

Reference: ED-12C/ED-109A: Section 7.3, 11, and Annex A

Keywords: assurance level; control category; CC1; CC2; software level

Answer:

The applicant is responsible for ensuring that the data items are controlled at the appropriate level. In some situations, it may be useful to combine data that have differing control categories into a single data item. When this is done, the data item should be controlled in accordance with the guidance for the higher control category.

A ED-12C example follows: An applicant decides that some Plan for Software Aspects of Certification (PSAC) content, as defined by ED-12C section 11, is to be included in the Software Development Plan (SDP). There is not an issue for Levels A and B systems since both documents are CC1, but for Levels C and D systems the SDP is controlled as CC2. This would mean the SDP should be controlled as CC1 to accommodate the control needed on the data that would otherwise be contained in the PSAC. Additionally, the SDP may need to be provided to the certification authority, as it contains information that is typically in the PSAC and may need certification authority approval.

A similar ED-109A example follows: An applicant decides that some Plan for Software Aspects of Approval (PSAA) content, as defined by ED-109A section 11, is to be included in the Software Development Plan (SDP). There is not an issue for AL 1 and 2 systems since both documents are CC1, but for AL 3, 4, and 5 systems the SDP is controlled as CC2. This would mean the SDP should be controlled as CC1 to accommodate the control needed on the data that would otherwise be contained in the PSAA. Additionally, the SDP may need to be provided to the approval authority, as it contains information that is typically in the PSAA and may need approval by the authority approval.

3.56 FAQ #56: HOW ARE REDUNDANCIES INHERENT IN THE SOFTWARE VERIFICATION DOCUMENTS ELIMINATED?

Reference: ED-12C/ED-109A: Sections 11.b, 11.3, 11.13, and 11.14

Keywords: documentation; verification

Answer:

There are no intended redundancies in the data required by ED-12C/ED-109A.

The three types of verification data stated in ED-12C/ED-109A are the Software Verification Plan (see ED-12C/ED-109A section 11.3), the Software Verification Cases and Procedures (see ED-12C/ED-109A section 11.13), and the Software Verification Results (see ED-12C/ED-109A section 11.14). The terms documentation and documents are avoided in ED-12C/ED-109A, since the output of the verification process may take different forms as discussed in ED-12C/ED-109A section 11.b.

The output of the verification process is used to show that the verification process objectives have been met. The data can exist in a single data item and can be referenced instead of repeating information.

3.57 FAQ #57: IS IT NECESSARY TO MENTION ALL THE ADDITIONAL CONSIDERATIONS IN THE PLAN FOR SOFTWARE ASPECTS OF CERTIFICATION (PSAC) OR PLAN FOR SOFTWARE ASPECTS OF APPROVAL (PSAA) OR IS IT SUFFICIENT TO MENTION ONLY THE APPLICABLE ADDITIONAL CONSIDERATIONS?

Reference: ED-12C/ED-109A: Sections 11.1.g and 12

Keywords: additional considerations; plans

Answer:

In the PSAC/PSAA, the applicant need only address the additional considerations found in ED-12C/ED-109A section 11.1.g and 12 that affect the certification/approval process. There is no specific guidance within ED-12C requesting the applicant to mention all the non-applicable additional considerations. However, the applicant may choose to list all the additional considerations in order to minimize questions from the certification/approval authority regarding their applicability.

It should also be noted that the list of additional considerations in ED-12C/ED-109A is not exhaustive. Other issues may arise, depending on the nature of the project, and should be identified in the PSAC/PSAA.

3.58 FAQ #58: HOW DO YOU IMPLEMENT RE-VERIFICATION?

Reference: ED-12C/ED-109A: Sections 11.3.h and 12.1.1.e

Keywords: re-verification

Answer:

Re-verification is the process of determining what specific verification activities should be performed to ensure that a change to previously verified software was the correct change and that the change did not adversely affect other software components or characteristics. Re-verification consists of these activities:

a. An analysis is performed to determine the software components and characteristics that may be affected by the change.

b. Appropriate verification items (for example, requirement and/or design reviews, test procedures and cases, timing analysis, memory analyses, etc.) are identified that should be performed to ensure the change was implemented correctly in the appropriate life cycle data, and that no adverse effects exist because of the change. This may include the development of new test procedures and cases.

c. The identified verification is performed, the results recorded, and a determination made that all applicable verification objectives have been satisfied for the change.

3.59 FAQ #59: WHAT TYPE OF NON-FLIGHT SOFTWARE IS COVERED BY ED-12C?

Reference: ED-12C: Sections 1.2 and 12.2

Keywords: non-flight software; tool qualification

Answer:

The scope of ED-12C is limited to software aspects of airborne systems and equipment certification. Therefore, ED-12C objectives apply to the airborne software, related life cycle data, and tools used directly in the development and verification of the airborne software. ED-12C does not apply to, nor does regulatory guidance currently suggest the use of ED-12C on any other non-flight software, though there may be a safety issue from the use of the non-flight software.

Nevertheless, there are some cases where the applicant may choose to use ED-12C, even though it is not required by the regulatory authority. Also, some contracts might be issued that require ED-12C for non-flight software even though the regulations do not.

The only type of non-flight software addressed by ED-12C is software tools used by software life cycle processes in the development of airborne software. The ED-12C section on tool qualification (see section 12.2) can be used to assess if a tool needs to be qualified, and the applicable Tool Qualification Level. Then, ED-215, Software Tool Qualification Considerations, defines the tool qualification process and the needed tool qualification data.

3.60 FAQ #60: IS A COMPLETE SET OF PLANS REQUIRED FOR MODIFICATIONS OF A SYSTEM?

Reference: ED-12C/ED-109A: Sections 9.3 and 12.1.1

Keywords: change; modification; plans; reuse

Answer:

The guidance of ED-12C/ED-109A specifies the need to submit plans and data to support the software aspects of certification/approval. Many applicants have questions about what plans and data are required for changes to a system which has previously been approved to ED-12C/ED-109A. Additionally, some applicants make frequent minor changes and have a question concerning the best approach to support certification/approval. More specifically, many applicants want to know if all documents need to be resubmitted for every change or if some documents can be omitted.

ED-12C/ED-109A section 12.1.1 addresses software modification. The objectives of ED-12C/ED-109A should be satisfied even if the product is a modification or derivative of a system that complies with the objectives of ED-12C/ED-109A. ED-12C/ED-109A does not preclude the reuse of the plans or other data items used to show compliance with ED-12C/ED-109A.

ED-12C/ED-109A and other regulatory guidance material allow flexibility in the documents and data submitted for certification/approval compliance. If the applicant is careful in producing their software plans, these documents and data may be reusable for changes (regardless of whether they are major or minor changes, made frequently or infrequently).

When changes are small in their scope and impact to the system, the applicant may be able to gain agreement with the certification/approval authority that a simplified data package is appropriate. However, this simplified package should still support all the objectives of ED-12C/ED-109A. The simplified data package may achieve this goal by using references to existing data.

When changes are large in scope or impact on the system, it is likely that more supporting data and analysis may be needed. This additional information may result in submitting new or updated plans, if the former do not support the scope of the change to the system or software.

In summary, ED-12C/ED-109A objectives should be satisfied for all changes. However, the applicant can minimize the need to resubmit data and plans, if the applicant properly develops and coordinates the original plans to allow for expected changes to the system and its software.

3.61 FAQ #61: WHAT CONSTITUTES A DEVELOPMENT TOOL AND WHEN SHOULD IT BE QUALIFIED?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.62 FAQ #62: WHAT ARE THE REQUIREMENTS FOR FLIGHT TEST ANALYSIS SOFTWARE AND GROUND-BASED TEST SOFTWARE?

Reference: ED-12C/ED-109A: Section 12.2

Keywords: test software; tool qualification

Answer:

For the purpose of this FAQ, flight test analysis software and ground-based test software are software tools used to monitor or analyze performance or compliance data of a system.

Flight test analysis software or ground-based test software can be used to automate some verification activities such as requirements-based testing. When used as a tool that eliminates, reduces, or automates a ED-12C/ED-109A verification activity without its output being verified, then the guidance for tool qualification for the flight test analysis software or ground-based test software is applicable (see ED-12C/ED-109A section 12.2). When the flight test analysis software or ground-based test software is not used for a ED-12C/ED-109A verification activity, such as when used to satisfy a system regulatory requirement, then qualification of the tool is outside the scope of ED-12C/ED-109A.

3.63 FAQ #63: FOR EXHAUSTIVE INPUT TESTING, THE APPLICANT SHOULD PROVIDE AN ANALYSIS WHICH CONFIRMS THE ISOLATION OF THE INPUTS TO THE SOFTWARE. WHAT DOES IT MEAN TO CONFIRM THE ISOLATION?

Reference: ED-12C/ED-109A: Section 12.3.1

Keywords: exhaustive input testing; isolation

Answer:

“To confirm the isolation” means to demonstrate by means of analysis and/or test that the defined set of valid inputs to the software (as defined by compliance with ED-12C/ED-109A item 12.3.1.b) are the only inputs allowable to the software to produce the defined set of valid outputs of the software (also as defined for ED-12C/ED-109A item 12.3.1.a). In other words, the inputs and outputs are bounded. This isolation may be achieved and/or enforced by the subject software itself, by a software component outside the subject software, or by another system component.

If the inputs can be bounded and independently isolated, then exhaustive input testing can be justified by testing the appropriate combinations of inputs and input equivalent classes to satisfy the verification coverage for the subject software.

3.64 FAQ #64: IS IT SUFFICIENT TO USE DIFFERENT LINKER OR LOADER TO PRODUCE DISSIMILAR VERSIONS FOR AVIONICS SOFTWARE?

Reference: ED-12C/ED-109A: Section 12.3.2

Keywords: dissimilar software; linker; loader

Answer:

No, using a different linker and/or loader alone would not be a suitable way of producing multiple, dissimilar versions of software. ED-12C/ED-109A states that multiple, dissimilar versions of software may be produced by combinations of the techniques described in the bulleted text of ED-12C/ED-109A section 12.3.2. These techniques in combination are expected to minimize the probability of common cause errors. The reason for bullet 6 is to remind the reader that using the same linker and/or loader may compromise the independence of the dissimilar systems.

3.65 FAQ #65: WHAT IS MEANT BY *"EQUIVALENT SOFTWARE VERIFICATION PROCESS ACTIVITY"* IN ED-12C/ED-109A SECTIONS 12.3.2.4 (TOOL QUALIFICATION FOR MULTIPLE-VERSION DISSIMILAR SOFTWARE) AND 12.3.2.5 (MULTIPLE SIMULATORS AND VERIFICATION)?

Reference: ED-12C/ED-109A: Sections 12.2.1, 12.3.2, 12.3.2.4, and 12.3.2.5

Keywords: dissimilar software; multiple-version dissimilar software; simulators; software verification process; tool qualification; tools

Answer:

ED-12C/ED-109A sections 12.3.2.4 and 12.3.2.5 deal with the impact on tool qualification for tools used in the framework of multi-version dissimilar software.

ED-12C/ED-109A section 12.2.1 states: *"The purpose of the tool qualification process is to ensure that the tool provides confidence at least equivalent to that of the process(es) eliminated, reduced, or automated."*

When dissimilar tools are used for multiple-version dissimilar software, it may be possible to modify the tool qualification process. It should be demonstrated that the use of such dissimilar tools (with the modified qualification process) provides the same confidence that the use of a single tool whose process is not modified. This confidence should be understood as noted in ED-12C/ED-109A section 12.3.2; that is, "*that the software verification process objectives are satisfied and that equivalent error detection is achieved for each software version.*" This is the meaning of "*equivalent software verification process activity"* in ED-12C/ED-109A. This confidence depends on the independence of the dissimilar tools, similarity of actions performed by both tools, and equivalency of the verification process activities that the dissimilar toolset is intended to eliminate, reduce, or automate. ED-12C/ED-109A sections 12.3.2.4 and 12.3.2.5 establish the minimum independence guidance for the dissimilar tools.

Multiple-version dissimilar software is described in ED-12C/ED-109A section 12.3, "Alternative Methods". As such, all proposals for modifying the tool qualification guidance should have certification authority agreement.

3.66 FAQ #66: WHAT IS THE DIFFERENCE BETWEEN CERTIFICATION, APPROVAL, AND QUALIFICATION?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.67 FAQ #67: WHAT IS ANALYSIS OF DATA COUPLING AND CONTROL COUPLING?

Reference: ED-12C/ED-109A: Sections 6.3.3, 6.3.4, and 6.4.4; Table A-7 of Annex A; and Annex B

Keywords: control coupling; data coupling; verification

Answer:

The ED-12C/ED-109A glossary (see Annex B) contains the following definitions for data coupling and control coupling:

- *"Data coupling - The dependence of a software component on data not exclusively under the control of that software component."*
- *"Control coupling - The manner or degree by which one software component influences the execution of another software component."*

The verification of data and control coupling involves a combination of the following:

- Reviews and analysis of software architecture, as stated in ED-12C/ED-109A section 6.3.3.b.
- Reviews and analysis of Source Code, as stated in ED-12C/ED-109A section 6.3.4.b.
- Requirements-based testing, confirmed by structural coverage analysis, as stated in ED-12C/ED-109A section 6.4.4.d.

The focus of this FAQ is on the objective stated in ED-12C/ED-109A section 6.4.4.d, which is:

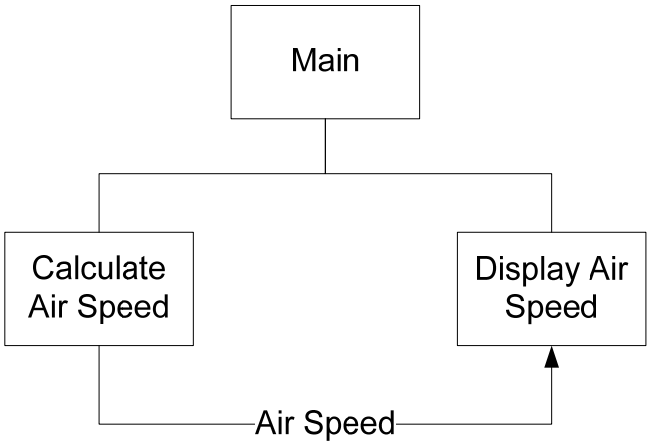
- *"Test coverage of software structure, both data coupling and control coupling, is achieved."*

The related activity described in ED-12C/ED-109A section 6.4.4.2.c is:

- *"Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components."*

The intent behind this objective is to ensure that applicants perform a sufficient amount of hardware/software integration testing and/or software integration testing.

The example below shows three components: a subprogram "Main", which calls two subprograms "Calculate Air Speed" and "Display Air Speed". The calculated air speed is passed between the two subprograms using a global variable "Air Speed".



In this example, there exists control coupling between "Main" and "Calculate Air Speed", and control coupling between "Main" and "Display Air Speed". There also exists data coupling between "Calculate Air Speed" and "Display Air Speed".

To satisfy the objective stated in ED-12C/ED-109A section 6.4.4.d, all of the following should be invoked during integration testing:

- The subprogram call from "Main" to "Calculate Air Speed".
- The subprogram call from "Main" to "Display Air Speed".
- The passing of "Air Speed" from "Calculate Air Speed" to "Display Air Speed".

These need not be three separate tests; they could be the same test.

3.68 FAQ #68: THE THIRD SENTENCE OF THE THIRD PARAGRAPH OF SECTION 3.2 OF ED-12C STATES THAT *"COMPONENT X ILLUSTRATES THE USE OF PREVIOUSLY DEVELOPED SOFTWARE USED IN A CERTIFIED PRODUCT."* (THE EQUIVALENT SENTENCE IN ED-109A STATES THAT *"COMPONENT X ILLUSTRATES THE USE OF PREVIOUSLY DEVELOPED SOFTWARE USED IN AN APPROVED SYSTEM."*) IS IT NECESSARY, FOR A REUSED COMPONENT TO HAVE BEEN USED IN THE CONTEXT OF A PREVIOUSLY CERTIFIED PRODUCT OR AN APPROVED SYSTEM?

Reference: ED-12C/ED-109A: Sections 3.2 and 12.1

Keywords: component; previously developed software; PDS; reuse; software life cycle

Answer:

No. The intent of ED-12C/ED-109A section 3.2 is to illustrate a sequence of software development processes for several components of a single software product with different software life cycles. For further information on the reuse of software, refer to ED-12C/ED-109A section 12.1 and other FAQs or DPs on previously developed software (PDS).

3.69 FAQ #69: WHAT IS THE RATIONALE TO HAVE SOFTWARE DESIGN PROCESS FEEDBACK TO THE PLANNING PROCESS IN SECTION 5.2.2.G OF ED-12C/ED-109A, WHERE FEEDBACK TO THE SYSTEM LIFE CYCLE PROCESS AND SOFTWARE REQUIREMENTS PROCESS SEEMS ADEQUATE?

Reference: ED-12C/ED-109A: Section 5.2.2.g

Keywords: plans; software design process; software planning process; software requirements process

Answer:

ED-12C/ED-109A section 5.2.2.g states: "*Inadequate or incorrect inputs detected during the software design process should be provided to the system life cycle process, the software requirements process,* ***or*** *the software planning process as feedback for clarification or correction.*" (Note: The underline and bold are added for emphasis.)

Whether or not the deficiencies are provided as feedback to the software planning process depends on the cause of the inadequate or incorrect input.

Software development and integral processes are defined during the software planning process. Inadequacies in either the development or integral processes can result in inadequate or incorrect inputs to the design process. Errors found in the design process such as those resulting from the misinterpretation of the software plans, incomplete or erroneous software standards, or incorrect or inadequate software requirements, may indicate the need for changes in the software planning process outputs themselves. That is, the planning process and/or the outputs of the planning process (the software plans and standards) may need to be modified to prevent similar deficiencies from continuing to occur.

3.70 FAQ #70: WHAT IS THE PURPOSE OF THE SECOND SENTENCE IN ED-12C/ED-109A SECTION 5.2.4.B?

Reference: ED-12C/ED-109A: Section 5.2.4.b

Keywords: deactivated code

Answer:

The second sentence of ED-12C/ED-109A section 5.2.4.b states: *"Unintended execution of deactivated code due to abnormal system conditions is the same as unintended execution of activated code."*

The purpose of the statement is to ensure that the same level of verification rigor is applied to in order to ensure the following:

(1) That activated code is not inadvertently or erroneously activated by an abnormal system condition.

(2) That deactivated code is not activated by an abnormal system condition.

NOTE: *The definition of deactivated code is included in the ED-12C/ED-109A glossary (see Annex B).*

3.71 FAQ #71: WHAT IS THE PURPOSE OF TRACEABILITY, HOW MUCH IS REQUIRED, AND HOW IS IT DOCUMENTED? FOR EXAMPLE, IS A MATRIX REQUIRED OR ARE OTHER METHODS ACCEPTABLE?

This FAQ has been removed, as approved by the WG-71/SC-205 committee.

3.72 FAQ #72: WHAT HAPPENS IF AN ERROR INDICATES A WEAKNESS IN THE DEVELOPMENT PROCESS ITSELF?

Reference: ED-12C/ED-109A: Sections 6.2.c, 7.2.3 through 7.2.5, and 11.17

Keywords: error; Problem Report; process weakness

Answer:

When the cause of an error is determined to be a process weakness, the defective process should be analyzed to determine whether it needs to be corrected and specific activities repeated.

The process weakness may indicate that similar errors occurred in other life cycle data produced by the same process. The impact of the errors and whether another process would capture these errors will determine the extent and timing of the needed corrective action. The following questions should be addressed as part of this corrective action:

- Will another process detect the potential life cycle data errors? If so, the life cycle data previously verified would not need re-verification.
- Do the potential life cycle data errors affect the system operation or safety?
  - a) If not, recording a Problem Report and correcting the process for future work may be adequate.
  - b) If the errors do affect safety, system operation, or assurance level substantiation, an analysis should be performed to identify the other life cycle data produced with this process that may contain similar errors. These life cycle data may need to go through the corrective action process using a corrected process before seeking product approval.

3.73 FAQ #73: ARE TIMING MEASUREMENTS DURING TESTING SUFFICIENT OR IS A RIGOROUS DEMONSTRATION OF WORST-CASE TIMING NECESSARY?

Reference: ED-12C/ED-109A: Sections 6.3, 6.3.1.c, 6.3.2.c, 6.3.4.f, 6.4.2.2, 6.4.3.a, 11.9, 11.10, 11.20.i, 12.1.1; and Annex A Table A-5

Keywords: timing; worst-case execution time; WCET

Answer:

Background and Scope

Worst-case execution time (WCET) is the longest possible time that is needed to complete the execution of a set of tasks on a given processor in a real-time computational environment. Generally, the WCET addresses the total time used in a fixed time slice, and possibly over multiple time slices. Those time slices may be set by a real-time interrupt-driven clock, or some other deterministic manner.

There are factors which can complicate the calculation of WCET. Some of these include:

- Multiple interrupts that drive the system.
- Many decisions in the algorithms.
- Use of processor cache for instructions or data.
- Lack of a real-time driven clock interrupt.
- Differing methods of scheduling the tasks to be executed.
- Use of a real-time operating system.
- Operational behavior that is non-deterministic (for example stack usage).

These factors should be considered when the software architecture and requirements are defined to reduce or eliminate the complexities in analyzing the WCET.

In addition to verifying that the software requirements related to timing have been met, ED-12C/ED-109A Table A-5 objective 6 shows that the objective defined in section 6.3.4.f (which mentions worst-case timing) is applicable to Levels A, B, and C for ED-12C (AL1, 2, and 3 for ED-109A).

One cannot simply add the worst-case execution time from individual units and expect to obtain a meaningful system level assessment. The worst-case timing could be calculated by review and analysis of the Source Code and architecture, but impact of compiler (including options), linker (including options), and processor behavior should also be addressed. Timing measurements should be accompanied by an analysis demonstrating that the worst-case timing would be achieved; the analysis should consider processor behavior, such as cache performance.

WCET Assessment Methods

ED-12C/ED-109A discusses WCET and permits a variety of verification methods for determining the WCET for a system (including analyses and tests).

ED-12C/ED-109A Section 6.3, Software Reviews and Analysis, discusses the use of test on the software product to supplement reviews and analysis as a means to determine worst-case execution time.

ED-12C/ED-109A Section 6.3.4.f states that as part of meeting the verification objective of the Source Code being accurate and consistent, the worst-case execution timing should consider the impact of factors such as compiler options and some hardware features.

Related Timing Considerations

The Software Requirements Data and Design Description of ED-12C/ED-109A sections 11.9 and 11.10, respectively, require that timing requirements and constraints in 11.9.d and resource limitations in 11.10.e be determined. Although the term WCET is not used explicitly, WCET is certainly one aspect of these timing aspects to be addressed.

Similarly, ED-12C/ED-109A section 12.1.1.d identifies the need to analyze timing impact when modifying previously developed software.

WCET Results

ED-12C/ED-109A section 11.20.i identifies that the WCET data should be summarized and presented in the Software Accomplishment Summary.

3.74 FAQ #74: WHAT IS THE DIFFERENCE BETWEEN THE DEVELOPMENT AND LIFE CYCLE OBJECTIVES STATED IN ED-12C FOR LEVEL A VERSUS LEVEL B SOFTWARE, AND HOW DOES THAT RELATE TO SAFETY? SIMILARLY, WHAT IS THE DIFFERENCE IN ED-109A FOR AL1 VERSUS AL2 SOFTWARE?

Reference: ED-12C/ED-109A: Section 6.4.4.2 and Annex A

Keywords: AL1; AL2; Level A; Level B; independence; modified condition/decision coverage (MC/DC); Source Code to object code traceability; structural coverage

Answer:

The major difference between Level A/AL 1 and Level B/AL 2 is that the objectives of Level A/AL 1 address the following:

- MC/DC structural coverage analysis (see section 6.4.4.2.a of ED-12C/ED-109A).
- Object code traceability (see section 6.4.4.2.b of ED-12C/ED-109A).
- More verification objectives with independence (see Annex A, Tables A-4 through A-7, of ED-12C/ED-109A).

The above objectives provide additional assurances for Level A/AL 1 software for the following reasons:

- MC/DC assures that the structure of the Source Code is further exercised. This further demonstrates the Source Code functions as they relate to the software requirements and the correctness of the design. Additionally, any extraneous code (such as dead code) or unreachable paths in the code may be identified.
- Object code traceability provides assurances that the compiler does not produce object code that has not been verified.
- Independence may detect more errors due to objectivity of evaluation.

Overall, Level A/AL 1 provides for more rigorous verification than Level B/AL 2; therefore, a higher level of confidence in safety and compliance with airworthiness requirements is achieved.

3.75 FAQ #75: CAN SAMPLING BE USED FOR SOME VERIFICATION ACTIVITIES (SUCH AS CODING RULES ON SOURCE CODE)?

Reference: ED-12C/ED-109A: Sections 6, 8, 11.8, and 12.3; Table A-5 of Annex A; and Annex B

Keywords: monitoring; sampling; software quality assurance; verification

Answer:

ED-12C/ED-109A does not define sampling; however, the industry accepted concept of sampling is that of a statistically valid method for acceptance of a product without examining the whole product.

In addressing this question, this paper does not use the industry accepted concept of sampling, but rather uses the term to refer to verification that is less than 100%.

The verification objectives described in ED-12C/ED-109A section 6 should be fully satisfied. Sampling should not be used to fulfill these objectives. This also applies to coding standards referenced in the question above (ED-12C/ED-109A section 6.3.4.d). The only exception to this is that regulatory authorities may accept less than 100% verification of conformance to the code presentation rules as defined in ED-12C/ED-109A section 11.8.b.

While not defined as a verification activity, the software quality assurance objectives do allow for monitoring, which is defined as follows:

"Monitoring – The act of witnessing or inspecting selected instances of test, inspection, or other activity, or records of those activities, to assure that the activity is under control and that the reported results are representative of the expected results. Monitoring is usually associated with activities done over an extended period of time where 100% witnessing is considered impractical or unnecessary. Monitoring permits authentication that the claimed activity was performed as planned." (See ED-12C/ED-109A Annex B.)

An applicant can propose an alternative method (ED-12C/ED-109A section 12.3). If an alternative method of compliance is sought, the applicant should demonstrate the validity of the proposed method and should show that the method is defined and controlled.

3.76 FAQ #76: CAN PROBLEM REPORTS AND VERIFICATION ACTIVITIES PERFORMED ON A SOFTWARE CONFIGURATION ITEM BE REFERENCED IN PREVIOUSLY APPROVED PRODUCTS WITHOUT REPEATING THIS EFFORT FOR EACH PRODUCT THAT USES THIS SOFTWARE CONFIGURATION ITEM?

Reference: ED-12C/ED-109A: Sections 12.1.2, 12.1.3, and 12.1.5

Keywords: configuration item; operational environment; Problem Report; re-verification; reuse

Answer:

Yes, if the Problem Reports and verification activities are relevant for each product that uses the software configuration item.

The extent of re-verification to be performed for each of the products is dependent on the nature of the hardware/software interfaces in each of the applications, as well as the magnitude and complexity of the change. Verification (for example, testing or analysis) should be performed, as necessary, in instances where the changed software configuration item is used to ensure that the software will continue to perform as intended in the operational environment. However, credit may be taken when verification is performed on portions of the change that are independent of the target operational environment or when the target operational environment is identical. Guidance in ED-12C/ED-109A sections 12.1.2, 12.1.3, and 12.1.5 may be applicable.

3.77 FAQ #77: THE SOFTWARE REQUIREMENTS DATA ARE DESCRIBED BY ED-12C/ED-109A SECTION 11.9. WHAT IS MEANT IN STEP 11.9.G: *"FAILURE DETECTION AND SAFETY MONITORING REQUIREMENTS"*?

Reference: ED-12C/ED-109A: Section 11.9.g

Keywords: failure detection; safety monitoring

Answer:

Software may be used to detect, monitor, and control failures. In these instances, the software may be passively monitoring and reporting hardware conditions (pass/fail discrete reporting) or actively running a routine that seeks to determine if there is an abnormal or unexpected system or component value within the hardware or software. If software is used to detect, monitor, and control failures, the appropriate requirements should include:

a. Built-in test (BIT) constraints and monitoring requirements, including detection, isolation, and reporting.

b. Requirements for recovery from hardware and software anomalous behavior.

c. Requirements for recovery from abnormal conditions for each state as determined by the system safety assessment process.

BIT constraints may include conditions which require limited or no-test conditions for a given routine (for example Start-up BIT) that are dependent on the system state when a fault occurred. Timing considerations of BIT routines and reporting of faults should also be considered in addition to throughput timing of the software run-time.

Derived software requirements for fault detection, isolation, reporting, and recovery should be provided to the system for evaluation with the system safety analysis.

3.78 FAQ #78: FOR SOFTWARE REQUIREMENTS EXPRESSED BY LOGIC EQUATIONS, HOW MANY NORMAL RANGE TEST CASES ARE NECESSARY TO VERIFY THE VARIABLE USAGE AND THE BOOLEAN OPERATORS?

Reference: ED-12C/ED-109A: Section 6.4.2.1.d

Keywords: requirements-based testing; testing

Answer:

One method is to test all combinations of the variables. For complex expressions, this method is impractical due to the large number of test cases required. A different strategy could be developed to verify the variable usage and the Boolean operators and to ensure that requirements-based test coverage is achieved. For example, the Boolean operators could be verified by analysis or review. To complement this activity, test cases could be established to demonstrate that each variable correctly influences the output.

3.79 FAQ #79: CAN AN APPLICANT FOR AIRCRAFT, ENGINE, OR PROPELLER CERTIFICATION TAKE CREDIT FOR ED-12C COMPLIANCE FOUND UNDER AN ARTICLE APPROVAL (THAT IS, TECHNICAL STANDARD ORDER (TSO) OR EUROPEAN TECHNICAL STANDARD ORDER (ETSO))?

Reference: ED-12C: Section 10

Keywords: certification credit; European Technical Standard Order; ETSO; Technical Standard Order; TSO

Answer:

Articles may be “authorized” by a certification authority. The authorization is completed under a process whose title varies by certification authority, but generally consists of a certification authority accepting a compliance statement and associated data from the applicant. For example, in the United States it is the Technical Standard Order (TSO) process, and in Europe the European Technical Standard Order (ETSO) process. For simplicity, the remainder of this section uses the term “TSO” when referring to any article authorization process. A TSO authorization includes compliance with the minimum performance standards and other requirements called out by the TSO. For software, compliance with ED-12C may be “approved” under the Technical Standard Order authorization (TSOA).

The certification authority for the aircraft, engine, or propeller may grant credit for a TSOA, if compliance with ED-12C was found under the TSO process, and the TSO is recognized by the certification authority. The TSOA, however, does not constitute an installation approval. Examples of characteristics that should be approved at the installation level are:

- The software level should be confirmed to be acceptable for the failure condition category associated with the intended functions of the appliance in the certification configuration.
- Any open Problem Reports should be confirmed to be acceptable for the certification configuration. Any safety-related Problem Reports should be corrected, reverified, and closed.
- Non-TSO functions performed by software in the article should be approved under the type certificate process, even if the software is accepted as part of the TSOA.
- The article should be confirmed to be compatible with the certification configuration of the aircraft, engine, or propeller.
- Additional guidance material not included in ED-12C and Supplements.

3.80 FAQ #80: WHAT NEEDS TO BE CONSIDERED WHEN USING INLINING?

Reference: ED-12C/ED-109A: Section 6.3.4.f

Keywords: control coupling; data coupling; inlining; memory usage; object code Source Code to object code traceability; stack usage; structural coverage; worst-case execution time

Answer:

The inline directive is issued to the compiler to indicate that the expansion of a subprogram body is to be within the code of a calling method and is to be preferred to the usual call implementation. The compiler can follow or ignore the directive to inline.

Since inlining is optional, the following analyses may be impacted: memory and stack usage, worst-case execution time, structural coverage, Source Code to object code traceability, and data and control coupling.

Conditional or iterative statements within inlined sections may cause omissions in structural coverage analysis. Some structural coverage tools may not handle decisions traced through inlined code. One method of control is to not allow conditional or iterative statements within code to be inlined. This may be controlled by the Software Design Standards and Software Code Standards for the project.

Analysis of the object code is a way to determine if inlining was implemented by the compiler.

3.81 FAQ #81: WHAT ASPECTS SHOULD BE CONSIDERED WHEN THERE IS ONLY ONE LEVEL OF REQUIREMENTS (OR IF HIGH-LEVEL REQUIREMENTS AND LOW-LEVEL REQUIREMENTS ARE MERGED)?

Reference: ED-12C/ED-109A: Sections 5, 11.9, and 11.10

Keywords: high-level requirements; HLR; low-level requirements; LLR; one level of requirements

Answer:

ED-12C/ED-109A section 5.0 (second paragraph) states: "*Software development processes produce one or more levels of software requirements. High-level requirements are produced directly through analysis of system requirements and system architecture. Usually, these high-level requirements are further developed during the software design process, thus producing one or more successive, lower levels of requirements. However, if Source Code is generated directly from high-level requirements, then the high-level requirements are also considered low-level requirements and the guidance for low-level requirements also apply.*"

This part of ED-12C/ED-109A section 5 addresses the case where the system level requirements are refined into software requirements suitable for coding in one refinement step; hence, allowing a single level of software requirements. However, this paragraph is sometimes misused to justify combining high-level requirements (HLRs) and low-level requirements (LLRs) into the same set of requirements. This results in a "flattening" of the requirements hierarchy and in the combination of all the requirements into a single set of requirements, possibly without establishing traceability between the LLRs and the HLRs. (Note: This does not refer to separate HLRs and LLRs in separate sections of the same document, but refers to the combination of HLRs and LLRs into a single set of requirements without distinguishing which are high-level and which are low-level.)

a. In general, HLRs represent "what" is to be implemented and LLRs represent "how" to carry out the implementation. Combining (or merging) these two levels into a single set could lead to the following certification concerns:

   1. The use of ED-12C/ED-109A section 5 to justify combining (or merging) HLRs and LLRs, such that visibility of the activities of the requirements and/or design processes is lost, represents a misinterpretation of the original objective of section 5. Combining HLRs and LLRs complicates the demonstration of compliance with ED-12C/ED-109A objectives, introducing confusion and increasing the authority oversight tasks for software aspects. Basically, compliance with the objectives of ED-12C/ED-109A Tables A-3, A-4, A-5, and A-6 is weakened when HLRs and LLRs are combined (or merged) into a single set of requirements (combining the "what" with the "how").

   2. Also, when LLRs and HLRs are combined (or merged) into one set, without traceability between HLR aspects and LLR aspects of the requirements, accurate re-verification is more difficult. A good change impact analysis is dependent on the traceability between HLRs and LLRs in order to identify the specific requirements and components affected by software changes.

   3. If unable to distinguish whether the software change(s) will impact only LLRs or both HLRs and LLRs, combined HLRs and LLRs may also lead to re-verification at the system level and software/hardware integration level even for minor software changes.

b. Combining (or merging) HLRs and LLRs is not recommended. However if it is done, a software developer should consider the following suggestions:

1. When airborne software components are large or complex, the software requirements process produces the HLRs and the software design process produces the LLRs and architecture. Thus, HLRs and LLRs are not the outputs of the same development processes. There is no reason to reduce the amount of verification activities required for compliance demonstration with ED-12C/ED-109A section 6. If HLRs and LLRs are merged, the objectives related to both the LLRs and the HLRs are therefore applicable to each requirement and to the outputs of both the software requirements and design processes. The attributes of both the Software Requirements Data (ED-12C/ED-109A section 11.9) and the Design Description (ED-12C/ED-109A section 11.10) should therefore be included in the single level of requirements. Both the Software Requirements Standards (ED-12C/ED-109A section 11.6) and the Software Design Standards (ED-12C/ED-109A section 11.7) would also apply to the single set of requirements.

2. There may be some systems where the system level requirements are refined into software requirements suitable for coding in one refinement step. In this case, a single level of software requirements may be feasible; however, a detailed analysis to show how the software will comply with each of the objectives of ED-12C/ED-109A is necessary.

3. If HLRs and LLRs are merged, although compliance with ED-12C/ED-109A may eventually be shown, the level of confidence that maintainability will be ensured over a long period of time is lower than when separate HLRs and LLRs are developed. In reality, if the HLRs and LLRs are merged into a single set of requirements, the impact of such an approach will need to be re-evaluated each time a change is made to the software (to ensure that the objectives of ED-12C/ED-109A section 6 are addressed).

3.82 FAQ #82: IF PSEUDOCODE IS USED AS PART OF THE LOW-LEVEL REQUIREMENTS, WHAT ISSUES NEED TO BE ADDRESSED?

Reference: ED-12C/ED-109A: Section 5.0, 5.2, 6.3.4.b, 6.4.4.2.c, 6.4.4.3

Keywords: control coupling; data coupling; design process; low-level requirements; pseudocode; structural coverage

Answer:

Pseudocode is a textual description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are not essential for human understanding of the algorithm (for example, variable declarations, system-specific code, and subroutines).

Pseudocode may be used in a variety of ways in software development. Some applicants may use pseudocode to provide more precise information within some of their low-level requirements (for example, to provide details of an algorithm). This may be a suitable way to use pseudocode.

However, the systematic use of pseudocode for all or most low-level requirements is not encouraged, whether it is an intermediate step between low-level requirements and Source Code or whether it is used in order to satisfy the following sentence in ED-12C/ED-109A, section 5.0: "*Low-level requirements are software requirements from which Source Code can be directly implemented without further information*."

It should be noted that pseudocode is neither required nor recommended by ED-12C/ED-109A. When using pseudocode as LLRs, several aspects of compliance with ED-12C/ED-109A objectives may prove difficult. For example:

- In cases where the pseudocode is syntactically very close to the Source Code, a gap between high-level and low-level requirements may be significant. As a result:
  - it may become more difficult to detect unintended or missing functionality at the level of the low-level requirements by means of HLR/LLR review; and
  - the software architecture may not include enough detail for verification.
- If the software architecture is insufficiently or not defined, the data and control coupling activity explained in ED-12C/ED-109A section 6.4.4.2.c ("*Analysis to confirm that the requirements-based testing has exercised the data and control coupling between code components*") may not be adequately performed.
- Unique identification of each LLR may be difficult when the LLRs are in the form of pseudocode, which may complicate the bi-directional traceability between the pseudocode (LLRs) and HLRs.
- Performing low-level testing against the pseudocode (LLRs) may lead to testing directly against the structure defined in the pseudocode. In this case, the functional aspects of the LLR may be missed, and robustness cases may fail to be created.
- In cases where low-level tests are established from the structure of the pseudocode, the capability of low-level tests to detect errors, missing functionality, and unintended functions introduced during the elaboration of the pseudocode (LLRs) is reduced.
- Pseudocode may be less functionally-oriented than textual LLRs. Thus, derived requirements may not be adequately identified and fed back to the safety assessment processes.

If pseudocode is used as low-level requirements (LLRs), the following topics should be taken into consideration, which may help to justify that the above concerns have been mitigated:

- All the ED-12C/ED-109A guidance for design development and verification should be applied.
- The review of the pseudocode (LLRs) against the HLRs needs to be performed to verify that none of the pseudocode included in the low-level requirements introduces any unintended functionality or unexpected behavior, and that all the derived requirements have been identified. In order to facilitate this analysis, some activities could be added into the process (for example, the addition of one or more levels of high-level requirements to specify the detailed functionality of the pseudocode, the addition of Trace Data within the pseudocode).
- It should be verified that the structural coverage analysis defined in ED-12C/ED-109A section 6.4.4.3 has not been affected by the use of pseudocode. In particular, the low-level requirements-based Software Verification Cases and Procedures (including the robustness cases) should be based on the expected functionality expressed by the pseudocode, and not established primarily based on the structure of the pseudocode. Alternatively, structural coverage of the code should be performed using the HLR-based tests.
- A software architecture should be developed, that allows the verification of data flow and control flow in the Source Code (see ED-12C/ED-109A section 6.3.4.b) and the complete verification of requirements-based tests (see ED-12C/ED-109A 6.4.4.2.c).

3.83 FAQ #83: SHOULD COMPILER ERRATA BE CONSIDERED?

Reference: ED-12C/ED-109A: Sections 4.4.1 and 4.4.2

Keywords: compiler; errata

Answer:

The answer is “yes,” according to ED-12C/ED-109A section 4.4.1.f.

In addition, ED-12C/ED-109A section 4.4.2 discusses some of the activities to consider during the planning phase when making decisions about the compiler. The areas to consider are the compiler optimization’s impact on structural coverage, non-traceable object code generated by the compiler, use of a new compiler, and use of different compiler options.

3.84 FAQ #84: HOW CAN ALL LEVEL D (AL 5) OBJECTIVES BE MET IF LOW-LEVEL REQUIREMENTS AND SOURCE CODE ARE NOT REQUIRED?

Reference: ED-12C/ED-109A: Section 1.4.p; and Annex A Tables A-2 and A-4

Keywords: AL5; high-level requirements; Level D; low-level requirements

Answer:

For Level D (AL5) software, ED-12C/ED-109A Annex A shows that there is no requirement to meet objective 6 of Table A-2 ("*Source Code is developed*"). Yet there are other objectives that typically cannot be met without Source Code. For example, how can objective 7 of Table A-2 ("*Executable Object Code and Parameter Data Item Files, if any, are produced and loaded in the target computer*") and objective 4 of Table A-8 ("*Archive, retrieval and release are established*") be met if objective 6 of Table A-2 is not met? As another example, how can objective 13 of Table A-8 ("*Software partitioning integrity is confirmed*") be met without knowledge of the high-level requirements, the software architecture, low-level requirements, and Source Code associated with the partitioning?

ED-12C/ED-109A section 1.4.p says, "*Compliance is achieved when all applicable objectives have been satisfied by performing all planned activities and capturing the related evidence.*" A "dot" in the Annex A tables means an applicant is able to show evidence that an objective has been met.

For Level D (AL 5), if there is no "dot" for Level D (AL 5), then that objective does not have to be met and no evidence has to be produced; additionally, the guidance in section 5.3 ("*Software Coding Process*") may be considered as optional. That is different than saying that, if there is no "dot," then the artifacts to be produced by that objective are not produced.

Most applicants will create Source Code as a part of the overall process to produce the Executable Object Code. If so, then configuration-managed Source Code is a necessary step to achieving compliance to ED-12C/ED-109A objective 7 of Table A-2 and objective 4 of Table A-8. However, since compliance to ED-12C/ED-109A objective 6 of Table A-2 is not required, the Source Code itself is not subject to any formal finding of compliance.

In order to confirm the integrity of the partitioning of a Level D (AL5) system, one needs knowledge of the high-level requirements, the software architecture, and the detailed implementation. So, although objectives 4, 5, and 6 of ED-12C/ED-109A Table A-2 are not required for Level D (AL 5), to satisfy objective 13 of ED-12C/ED-109A Table A-4, those low-level requirements and related Source Code associated with the partitioning should be both available and verified.

CHAPTER 4

DISCUSSION PAPERS (DP)

This section compiles the ED-12C/ED-109A Discussion Papers (DP). The purpose of a DP is to provide clarification for certain sections of ED-12C/ED-109A in cases where the clarification requires more than a short answer to a question. A DP contains no new or additional guidance material.

Disclaimer: DPs have been prepared and approved by Joint Committee WG-71/SC-205. These DPs have no recognition by the certification/approval authorities and are provided for information purposes only.

4.1 DP #1: VERIFICATION TOOL SELECTION CONSIDERATIONS

This Discussion Paper has been removed, as approved by the WG-71/SC-205 committee.

4.2 DP #2: THE RELATIONSHIP OF ED-12B/ED-12B TO THE CODE OF FEDERAL REGULATIONS (CFRS) AND JOINT AVIATION REQUIREMENTS (JARS)

This Discussion Paper has been removed, as approved by the WG-71/SC-205 committee.

4.3 DP #3: THE DIFFERENCES BETWEEN ED-12A AND ED-12B GUIDANCE FOR MEETING THE OBJECTIVE OF STRUCTURAL COVERAGE

This Discussion Paper has been removed, as approved by the WG-71/SC-205 committee. See FAQ #44 for information on structural testing.

4.4 DP #4: SERVICE HISTORY RATIONALE FOR ED-12C

Reference: ED-12C: Section 12.3.4

Keywords: service history

This paper clarifies guidance from ED-12C on considerations when using service history as an alternative method of compliance for embedded software. Commentary is provided for each topic in ED-12C section 12.3.4. The ED-12C headings are italicized and underlined; the commentary is not. The full text of ED-12C section 12.3.4 is not repeated here – only the section titles and number. Therefore, the DP should be read in conjunction with ED-12C section 12.3.4.

NOTE: *For clarification of service experience relevant to ED-109A, see DP #18 (section 4.18).*

12.3.4 Service History

Service history is under ED-12C section 12.3, “Alternative Methods.” Section 12.3.a states: “*An alternative method should be shown to satisfy the objectives of this document or the applicable supplement*.” This is the definition of “equivalent safety” for this discussion paper. “Certification/approval credit” in this instance refers to using service history towards satisfying the intent of the objectives in ED-12C Annex A.

The proponent for the software product is responsible for the collection of data justifying the requested certification credit based on the software's service history.

Necessary conditions for this justification are:

- The service period duration is sufficient.
- The new operational environment is the same or similar (with additional verification needed if not the same).
- The product is stable and mature (that is, few Problem Reports and/or modifications occurred during the service period).

12.3.4.1 Relevance of Service History

a. *Type of service history*. The extent and appropriate measures of service history to be used should be defined and agreed.

b. *Known configuration*. Configuration management allows effective assessment of the product's service history in the presence of changes, such as added functionality or error correction.

c. *Operating time collection process.* For collection of operating time or event data, the following needs to be considered:

- Method by which operating time or number of events was measured.
- Reliability of the means of measuring operating time or number of events.
- Method by which operating time or number of events was reported.
- Reliability of the means of reporting operating time or number of events.
- Impact of portions of software unused during normal operations on service history credit. For example, credit may not be granted for software components that were not active or used during the service period.

d. *Changes to the software*. The applicant should have records of all changes made to configuration items. The configuration history should record discrete changes made to add functionality or correct errors. It should provide the evidence to substantiate the integrity of the processes applied to make such changes. Records should exist to identify all changes made to the Executable Object Code or Parameter Data Item Files, regardless of the source of the change (for example, changes introduced through the use of different compiler options). The following areas should be considered when assessing the adequacy of configuration management and control of the integrated software system (all components for which certification credit is sought using service history):

- Completeness and reliability of the evidence that all components of the software have been change controlled throughout the service period.
- Number and significance of the modifications.
- Method by which the validity of modified software for service history credit is established.

Software or hardware changes could affect the applicability of service history collected prior to the change.

e. *Usage and Environment.*

(1), (2) The applicant should provide an analysis which shows that the software will be performing the same function in the proposed new application as it performed during the service history period. For example, in real-time control and related software, basic performance and accuracy requirements, as well as more detailed issues such as input ranges, output ranges, and data rates, should be examined. In operating systems and protocol stacks, relevant execution characteristics, such as event sequences which might drive the software operation should also be considered. This analysis may be contained in the Plan for Software Aspects of Certification or other document.

No credit for service history can be claimed for software functions that were not exercised in the previous application. These software functions may need to be evaluated in the proposed new application to ensure they do not present a hazard, even if they are still not intended to be exercised.

(3) The service history environment should be assessed to determine whether problems may have been avoided due to technology-related usage differences (for example, manual versus automatic operations) or system level recovery mechanisms. If so, the intended environment needs to be assessed to determine whether these same mechanisms will exist.

Software service history will often be part of the service history of a larger system. Therefore operating environment is an important consideration in determining relevance. Considerations in assessing the relevance of the operating environment include:

- Adequacy of data available to support analysis of similarity of operating environment. If environment changes over time, data may need to support differentiation between relevant and irrelevant portions of the service history duration.
- Similarity of the hardware environment of service history to the target hardware environment. Resource differences (for example, time, memory, precision, communication services) between the two environments should be analyzed. Changes needed for the software to be compatible with the new environment may invalidate the service history or may preclude service history credit for hardware compatibility objectives.

For reuse of avionics, the process is to compare the system hazards to which the equipment could contribute in the previous system to those in the target application. If there were fewer hazards or less severe hazards assigned to the software under consideration in the previous application, its software level would be likely to increase in the new application. Additional work may be required to assure compliance with the new safety objectives if it cannot be shown that the service usage regularly exercised the aspects of the software which support the management of the newly allocated hazards. For any application of service history, the applicant should identify whether there are different requirements of the new installation or environment that make any demands on the system that may affect the safety objectives. The applicant should assure that the implementation of these requirements have been exercised in service or are subjected to verification to the appropriate level as defined by ED-12C. See ED-12C section 12.1 for additional guidance.

(4) The relationship between the service history environment and the intended environment includes, but is not limited to, processor and memory utilization, accuracy, precision, communication services, built-in-tests, fault tolerance, channels and ports, queuing modes, priorities, and error recovery actions. If there were changes to hardware during the service history duration, analysis should be conducted to assess whether it is still appropriate to consider the service history duration before the modifications.

f. *Deactivated code*. Functions that did not cause a problem may not have left any indications regarding their usage; absence of problems may not mean that the function was free from errors. Additional verification may be required to support compliance with the safety objectives for the subset components, if they are found to be used differently than in the prior service due to the fact that the whole software package is not reused as a whole.

If the analysis of the software's use in its previous environment identifies code that is rarely executed, but which may be executed in the new environment, then verification evidence is required for the code and the collected service history may not be applicable. Even if the unused functionality is unlikely to be invoked in the new environment, supporting evidence is needed to show that differences in operational usage will not cause an invocation of the unused functionality.

12.3.4.2 Sufficiency of Accumulated Service History

a. The amount of service history necessary for certification credit is usually derived from safety objectives. Safety objectives are usually related to failure severity or hazard classification and are identified during the system safety assessment. The system safety assessment process will thus need to provide the safety objectives as inputs to the determination of sufficient service history.

In some cases, time duration is not as relevant to determining failure rates as using the number of events such as takeoffs or landing, flight hours, flight distance, total population operating time, or the number of times an operator queried the software for specific information.

The amount of service history needed will also depend on the required confidence the safety objective has been achieved (for example, 90% or 95% confidence that the safety objective has been met). More service history is needed for higher confidence. Methods for calculating the amount of service history needed as a function of confidence level should be based on a documented and justified method accepted by the approval authority. Such methods can be found in international industrial and military reliability standards.

b. Differences between the service history environment and the system operational environment may necessitate additional service history, and may mean that techniques other than service history need to be used.

c. Some objectives require more service history than others to attain the confidence required. The rationale for why the amount of service history proposed is sufficient to address the objectives proposed needs to be documented.

d. Coupling service history with other forms of evidence may reduce the amount of service history needed in order to meet some objectives. The extent of the reduction depends on the type and completeness of the other means of certification being used.

12.3.4.3 Collection, Reporting, and Analysis of Problems found during Service History

a. *Problem reporting process.* Service history application requires collection of problem data in order to substantiate assertions regarding the integrity of the software. The intent of this item is that users are able to report anomalous behavior and that any such reports are recorded. The system should contain information necessary to support assessment of problems as described in paragraphs b and c, below. Issues to consider in assessing the problem reporting process include:

- Method and reliability of detecting in-service problems.
- Method and reliability of recording in-service problems.
- Method for determining problem severity.
- Method for differentiating software-related problems from other problems.
- Method for assessing relevance of and (if appropriate) collecting non-service problems (for example, problems detected during rehearsals or preliminary testing).
- Method for assessing impact of product improvements or unresolved software Problem Reports (if any) on service history credit.

Common-cause problems will occur together in time and may have a greater impact than problems with separate causes. Furthermore, common-cause problems may defeat system redundancy strategies and so could have safety implications. Cascading problems will generally occur closer together in time than other problems, so they may have a greater impact than the same number of non-cascading problems. Common-cause and cascading problems violate the assumptions usually made when measuring software failure and repair rates, and so should be treated with great caution if quantitative analysis techniques are used at the system level to determine the amount of service history needed.

b. *Process-related problems.* System design errors, hardware design errors, and hardware failures are outside the scope of ED-12C. While they are still a cause for concern and may invalidate the product service history application, they do not indicate an inadequate software process.

The presence of multiple errors, particularly systemic errors that should have been detected during software development, may indicate an inadequate software development process. These errors should be examined to assess whether they may be attributed to a weak development process. This assessment should consider trends and characteristics of the errors such as:

- Large number of defects: A large number of defects could indicate an inadequate development process. The size of the product should be considered to determine whether the density of defects is indicative of a process issue.
- Failures caused by off-nominal inputs: Failures of this nature could indicate a lack of robustness in the testing process or an inadequate requirements process.
- Partially fixed defects: This could indicate poor requirements, design, or regression testing processes.
- Reintroduction of errors: This could indicate a poor configuration management process.
- Failures caused by borderline values: Failures of this nature could be caused by improper algorithm or interface design – particularly with respect to numeric processing.

In the case of inadequacies which are clearly traceable to an identifiable process deficiency or omission, additional analysis, inspection, or other verification should be performed. Service history may continue to be used, provided the identified process deficiency or omission and its effects are addressed. In this case, the software should also be analyzed to find any other latent errors introduced by the inadequate process.

c. *Safety-related problems.* The existence of safety-related problems may be indicative of an inadequate development process (that is, a development process that does not conform to all the objectives of ED-12C). If an inadequate software process is indicated, three actions need to be taken:

1. Identify the gaps in the development process.
2. Analyze the product to determine if other safety related errors exist.
3. Provide assurance for the missing or incomplete elements of the development process, generally via additional verification activity to meet ED-12C objectives.

Credit for service history may not be possible if safety-related problems indicate an inadequate development process, and the process of reverifying the software and correcting defects could introduce new defects or change the results of any quantitative analysis.

Service history may be used even if a safety-related problem has been found if evidence can be presented showing that the problem has been resolved, and it was not the result of a systemic cause (for example, there was a problem with the test environment that would not be seen in operation).

12.3.4.4 Service History Information to be Included in the Plan for Software Aspects of Certification

a-b. The applicant should define the rationale for claiming service history credit, the type of service history to be claimed, the amount required to support the application, and the measures to be applied. Note that service history is unlikely to be applicable to Level A, because of the impact of any differences in operational environment on assurance and because of the amount of service history that would be needed.

c. The rationale for calculating number of hours in service should require collection and analysis of any discrepancies in:

- total operating time and workload by operating mode (if there are multiple modes);
- workload by function for each operating mode; and
- proportion of failures in software functionality not involved in proposed use.

The requirement that a single copy account for a pre-agreed percentage of the total operating time is intended to ensure that at least one copy ran long enough to have encountered any stability problems present in the software.

d-f. Service history from systems with long service lives may include failures. These failures do not necessarily render the software invalid for its intended application, particularly at lower software levels. At higher software levels, any failures may invalidate the use of the service history, particularly if those failures were safety-related. There should not have been any failures observed in the service history data collected for Level A or Level B. These details should be agreed in advance based on the software's contribution to system safety.

In cases where the plan called out a number of failures that on later analysis was exceeded, this section allows for re-negotiation of the applicability of the service history to the intended software level.

4.5 DP #5: APPLICATION OF POTENTIAL ALTERNATIVE METHODS OF COMPLIANCE FOR PREVIOUSLY DEVELOPED SOFTWARE (PDS)

Reference: ED-12C: Sections 12.3 and 12.3.4, Annex A, and Annex B

Keywords: alternative methods; formal methods; legacy software; PDS; previously developed software; prior product certification/approval; restriction of functionality; reverse engineering; service history

4.5.1 Purpose

This discussion paper has two main goals: (1) to present some potential alternatives that contribute to satisfying ED-12C objectives in cases where the conventional artifacts are not available or are only partially available, and (2) to suggest methods for presenting this information to a certification authority. This condition will most likely occur for software-based systems that incorporate previously developed software (PDS). As defined in ED-12C, PDS is: "Software already developed for use. This encompasses a wide range of software, including COTS software through software developed to previous or current software guidance."

This information is developed on the basis that the conventional ED-12C development processes are performed on the software-based system as a whole, but credit is sought for as much of the PDS development as possible. Integration of the PDS into the software-based system and subsequent verification would be included as part of the software-based system development.

NOTE: *ED-109A users should note that the overall content of this DP is included in ED-109A section 12.4.11; therefore, only ED-12C is mentioned in this DP.*

4.5.2 Limitations

This discussion paper suggests potential alternative methods for satisfying the objectives in ED-12C for PDS. This paper addresses only the PDS and not its integration into the system. The paper is not intended to be guidance material. It is recommended that applicants considering the use of legacy software review any legacy software guidance by certification authorities.

This discussion paper is to be used in conjunction with ED-12C. It does not replace or supersede ED-12C.

4.5.3 How to Use this Discussion Paper

This discussion paper presents seven potential alternative methods along with ideas and rationale that may be used in seeking credit for satisfying the objectives of ED-12C. These are applied at the process level to the tables in Annex A of ED-12C. The seven potential alternative methods were selected as candidates for additional clarification based on how the methods may be applied using ED-12C. Two of the methods are from ED-12C section 12.3 and are discussed along with five additional methods from industry experience. Another method, the use of architectural methods, has also been identified as a candidate for clarification; however, it is not addressed in this discussion paper.

There are other possible alternative methods that may be applied to PDS software. However, the intent of this discussion paper is to provide a clarification of how some traditional techniques may be applied to satisfy the ED-12C objectives. The general analysis approach demonstrated in this discussion paper may be utilized to determine what analysis may be necessary for other potential alternative methods.

It should also be understood that several alternative methods may be combined to support the accomplishment of the ED-12C objectives.

This paper is organized such that section 4.5.4 provides definitions of seven potential alternative methods. Section 4.5.5 provides suggestions on applying these alternative methods to the processes defined in ED-12C Annex A tables.

During the project planning process, the applicant is expected to provide a clearly defined basis for certification to the certification authority (typically in a Plan for Software Aspects of Certification (PSAC)), describing how these specific process objectives will be satisfied for the PDS as all or part of a software application. If an applicant proposes a novel approach to one or more of the ED-12C objectives, they should show how that approach satisfies the objective(s). For example, the intent of software testing as stated in ED-12C should be met regardless of the method chosen, whether traditional or alternative methods are used.

This discussion paper recommends that the PSAC contain a process-level definition of how alternative methods are being applied to the ED-12C processes for the PDS. It also suggests that the Software Accomplishment Summary (SAS) contain a description of how the alternative methods were applied for the inclusion of the PDS. It is recommended that the applicant periodically coordinate the use of the PDS with the certification authority to help maintain continuity from the PSAC to the SAS. Figure 4-1 shows the application of this discussion paper with respect to PDS in the development of the PSAC and SAS.

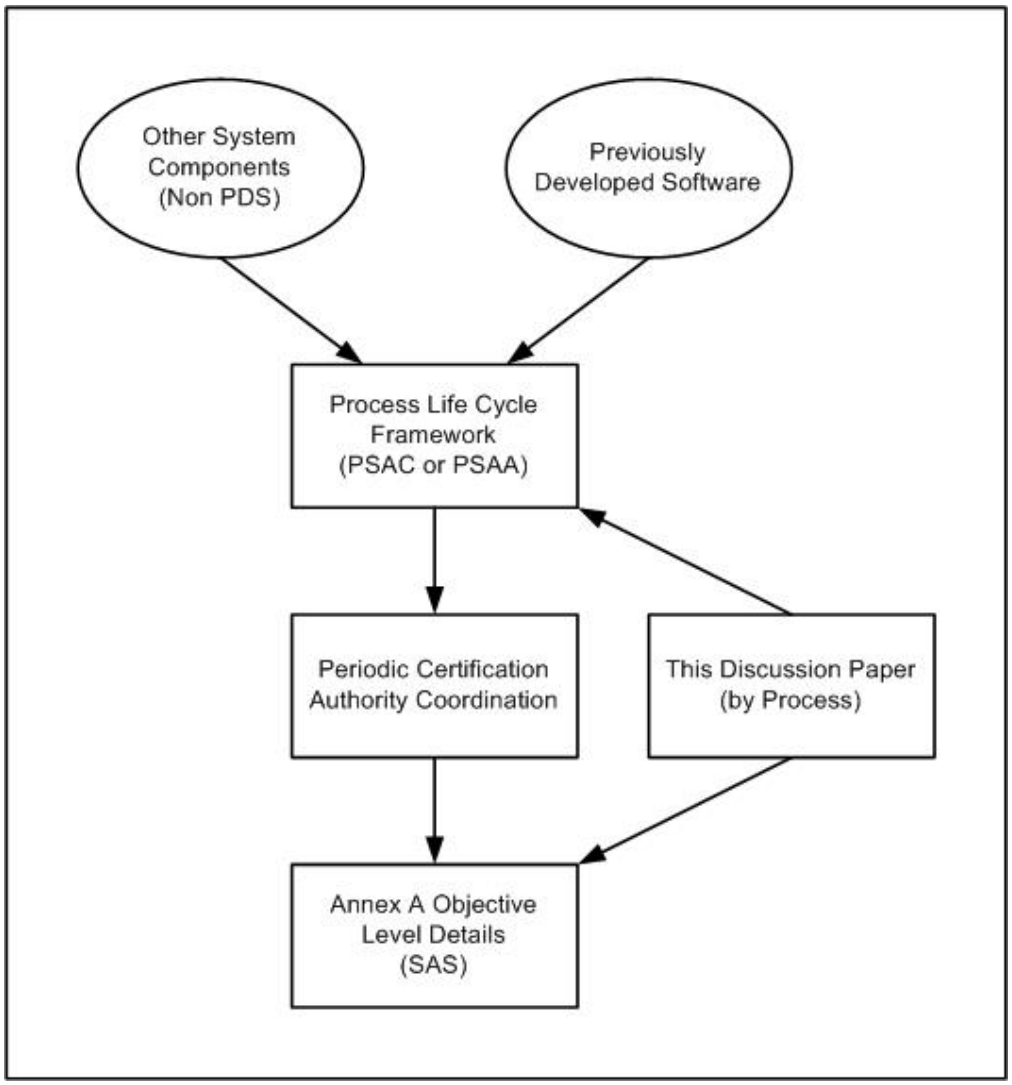


FIGURE 4-1: PDS RELATIONSHIP TO PSAC/PSAA

4.5.4 Definitions

The following sub-sections describe definitions of seven potential alternative methods. The potential alternatives are process recognition, prior product certification/approval, reverse engineering, restriction of functionality, service history, formal methods, and audits and inspections.

4.5.4.1 Process Recognition

4.5.4.1.1 Description

Process recognition is the acceptance of the evidence that a recognized development process was applied to a PDS product.

4.5.4.1.2 Potential Achievements

If it can be demonstrated that the PDS was developed under a defined process that had an independent accreditation throughout the software life cycle, then it may be possible to show that the defined process complies with the ED-12C objectives. While all of the objectives of ED-12C should be met (for the applicable software level), satisfactory evidence of the software development processes may allow credit to be taken for satisfying objectives achieved in the process used by the original PDS developer. The data from this process should be analyzed against the objectives of ED-12C to determine which objectives were satisfied by the process.

4.5.4.1.3 Inputs

The applicant should show that the developer of the PDS can provide data acceptable to the certification authorities and that it was utilized for the full development of the PDS. The PDS process data should be available to allow the analysis of the PDS software development process against the ED-12C objectives. The user of the PDS should develop a matrix that shows compliance.

4.5.4.1.4 Limitations

The process recognition method is limited because the development process may not have produced the necessary data needed to evaluate for ED-12C credit.

4.5.4.2 Prior Product Certification/Approval

4.5.4.2.1 Description

Prior product certification/approval may have occurred where the PDS was approved as a part of a previously certified, approved, or qualified system application. Examples of product certifications/approvals that may be considered include civil aircraft application approvals, ground equipment approvals, security certifications, medical device certifications, military application qualifications, and nuclear industry certifications.

4.5.4.2.2 Potential Achievements

Successful use of prior product certification/approval evidence may result in the acceptance of that certification's software development process and data as compliant to ED-12C objectives. This may result in a reduction of effort to show compliance of the PDS to the desired ED-12C level.

4.5.4.2.3 Inputs

The needed inputs are data items that support the objectives of the product's PDS prior certification and the basis of that certification. A mapping of ED-12C objectives to the prior product certification/approval objectives may show compliance with the objectives and thereby provide a direct means of demonstrating compliance with ED-12C.

4.5.4.2.4 Limitations

The prior product certification/approval method may not provide a complete mapping of objectives with ED-12C.

4.5.4.3 Reverse Engineering

4.5.4.3.1 Description

Reverse engineering is the process of developing higher level software data from existing software data. Examples include developing Source Code from object code (or Executable Object Code), or developing high-level requirements from low-level requirements (see ED-12C Glossary Annex B).

4.5.4.3.2 Potential Achievements

Reverse engineering may produce software life cycle data that can be reviewed or analyzed to satisfy the objectives of ED-12C, such as design structure, Source Code, or calling trees.

4.5.4.3.3 Inputs

The inputs needed are the PDS lower level software life cycle data, such as Source Code, object code, or Executable Object Code.

4.5.4.3.4 Limitations

The use of reverse engineering is subject to interpretation on many points. It may be difficult to produce a complete set of data to satisfy all ED-12C objectives.

4.5.4.4 Restriction of Functionality

4.5.4.4.1 Description

The concept "restriction of functionality" involves restricting the use of a PDS to a subset of its functionality, by methods such as, but not limited to, run-time checks, design practices (for example, company Software Design Standards and Software Code Standards), or build-time restrictions. The restricted set of functional requirements may then become the basis for use of the PDS.

4.5.4.4.2 Potential Achievements

This restriction of functionality method may make it feasible to show compliance with ED-12C objectives, particularly by reducing verification and auditing activities to the restricted subset of functional requirements.

4.5.4.4.3 Inputs

The definition of subset functionality should be determined and documented (for example, Software Requirements Data). Additionally, the functionality of the PDS should be known. The description of the restriction mechanisms should be documented and verified.

4.5.4.4.4 Limitations

The mechanisms by which the restrictions are enforced may need additional verification. It may be difficult to ensure the PDS has an architecture that prevents execution of undesired (that is, deactivated) code.

4.5.4.5 Product Service History

4.5.4.5.1 Description

Product service history is the utilization of previous in-service experience of the product and is described in ED-12C section 12.3.4.

4.5.4.5.2 Potential Achievements

Previous use of a software product that is relevant to its intended application may constitute evidence of product integrity. For example, it may be possible to offer evidence of acceptable product service history for some aspects of software verification. More credit is likely to be given if the product has been used in an application that required the same or a higher criticality or software level.

Note that ED-12C section 12.3.4 sets out criteria that should be met for the service history to be applicable.

4.5.4.5.3 Inputs

Full details of the service history are needed, including information about the problem tracking process and software configuration management process. See ED-12C, section 12.3.4 for further assistance.

4.5.4.5.4 Limitations

Limitations will arise where differences exist in the actual service environment and the proposed service environment. (See DP #4 in section 4.4 for additional information.)

4.5.4.6 Formal Methods

4.5.4.6.1 Description

ED-12C defines formal methods as "*descriptive notations and analytical methods used to construct, develop and reason about mathematical models of system behavior.*" The Formal Methods Supplement provides detailed guidance on using formal methods.

4.5.4.6.2 Potential Achievements

By the use of the mathematical proofs available from formal methods, the verification data needed to demonstrate compliance with higher level requirements may be replaced under certain cases. The formal analysis of PDS that has been developed with formal methods may itself be reusable in a more complex system.

4.5.4.6.3 Inputs

Inputs include full details of the particular formal methods used and the life cycle process to which it was applied. Evidence should be provided that the formal methods chosen are sufficient to satisfy the ED-12C objectives and the Formal Methods Supplement guidance.

4.5.4.6.4 Limitations

In practice, the application of formal methods may be limited by the complexity of the system or subsystem.

4.5.4.7 Audits and Inspections

4.5.4.7.1 Description

Audits and inspections are a means by which one can determine if a process has been adequately performed.

4.5.4.7.2 Potential Achievements

Reviews of PDS software data in conjunction with Software Quality Assurance (SQA) Records substantiating the process applied to these software data may help satisfy these objectives. In some cases, a traditional inspection of the data itself may be needed. In other cases, the alternative method of reviewing other evidence of the process activities may help satisfy these objectives. This method may also be used in conjunction with other alternative methods.

4.5.4.7.3 Inputs

Inputs needed include software data items in support of the audit or inspection.

4.5.4.7.4 Limitations

Audits and inspections are limited, since evidence can only be provided when necessary documentation is available.

4.5.5 Applicability of Alternative Methods

This section shows the applicability of seven potential alternative methods to the process objectives of ED-12C (Annex A Tables A-1 through A-10). This section follows the same organization as ED-12C (Annex A Tables A-1 through A-10).

4.5.5.1 Software Planning Process (Annex A Table A-1)

There is no current alternative method or item evidence that may be substituted for the PSAC. The PSAC should define the framework for mapping the evidence from the PDS to the satisfaction of each objective in Annex A. Within this framework, the PSAC should include for the PDS certification credit sought (full or partial): analyses and assumptions made, means of compliance, safety aspects impacted by the use of this PDS or its lack of development data, and any activities remaining to be accomplished during the project in order to satisfy each Annex A objective for the appropriate software level. These alternative methods may either satisfy or partially satisfy software planning objectives:

- **Process Recognition**: May provide reuse of plans and standards developed for a previous development. The existence of a suitable plan from the previous program that can be mapped to ED-12C objectives may preclude the need to duplicate the same plan for the current program. In the absence of PDS planning data, a roadmap may need to be provided mapping PDS data to ED-12C objectives.
- **Prior Product Certification/Approval**: May provide reuse of plans and standards developed for a previous development. The existence of suitable plans and standards from the previous program may preclude the need to duplicate the same plans and standards for the current program.
- **Reverse Engineering**: Generally not applicable for these objectives.
- **Restriction of Functionality**: Generally not applicable for these objectives.
- **Product Service History**: Generally not applicable for these objectives.
- **Formal Methods**: Generally not applicable for these objectives.
- **Audits and Inspections**: Reviews of PDS data, such as Software Verification Results, applied development standards, and other life cycle data, in conjunction with SQA Records substantiating the process applied to the development of these software data items may help satisfy these objectives.

4.5.5.2 Software Development Processes (Annex A Table A-2)

The level of detail needed in the software life cycle data may increase as the software level increases.

- **Process Recognition**: An applicant may utilize the recognized process by using the ED-12C objective roadmap from the planning process. The roadmap may be used to clarify the usefulness and completeness of PDS data items and to determine if the process applied to the PDS can show compliance with ED-12C objectives.
- **Prior Product Certification/Approval**: PDS data items obtained from previous certifications may be applicable in satisfying the ED-12C objectives.
- **Reverse Engineering**: Elements of the development process such as requirements, architecture, Design Description, or even the Source Code may be produced from other available elements of the implementation.
- **Restriction of Functionality**: A restriction of PDS functionality may provide a reduction in the amount of applicable PDS functionality subject to software development process objectives.
- **Product Service History**: Adequate and relevant service history may mitigate the need for providing low-level requirements. Considerations should include the intended operational environment impact on the PDS, the role of the PDS in the safety model, and the mapping of PDS functionality to the application's needs. For example, if a PDS without adequate development data was used unchanged in a similar application within a system on the same operational platform, then a degree of relevance may be achieved by the service history of that PDS.
- **Formal Methods**: Formal methods may be relevant for some of these objectives, for example, PDS that was developed using a formal requirement development method.
- **Audits and Inspections**: Post-development reviews of PDS data items such as a Product Description, Equipment Specification, and other elements of the development process in conjunction with SQA records substantiating the process applied to these data items may help satisfy these objectives. In some cases, a traditional inspection of the data itself may be needed. In other cases, the alternative method of reviewing other evidence of the process activities may help satisfy these objectives. Third-party inspections may help alleviate proprietary data considerations.

4.5.5.3 Verification of Outputs of Software Requirements Process (Annex A Table A-3)

- **Process Recognition**: An applicant may utilize the recognized process by using the ED-12C objective roadmap from the planning process. The roadmap may be used to clarify the usefulness and completeness of PDS data items and to determine if the process applied to the PDS can show compliance with ED-12C objectives.
- **Prior Product Certification/Approval**: PDS data items obtained from previous certifications may be applicable in satisfying the ED-12C objectives. For example, a previous certification may provide a traceability analysis showing the flow from system requirements through software requirements. The applicant's ability to reuse the PDS effectively is related to its previous software level.
- **Reverse Engineering**: Data items of the previous development process may be used to generate missing or incomplete data items such as using test procedures to generate Software Requirements Data.

- **Restriction of Functionality**: A restriction of PDS functionality may provide a reduction in the amount of applicable PDS functionality subject to software requirements process objectives.
- **Product Service History**: Similarity of operating environments may provide empirical evidence that supports the compatibility of high-level requirements to their target environment.
- **Formal Methods**: This could provide mathematical proofs of correctness and accuracy of the requirements. The use of mathematical proofs may be used as a means of verification or to support verification.
- **Audits and Inspections**: Reviews of PDS data items, such as requirements and safety analyses, may provide evidence of accurate and consistent requirements. The PDS requirements specification can be audited against documented standards. In some cases, the PDS supplier may be able to provide relevant test, review, and analysis data for the stand-alone PDS. Such data may be used to supplement tests on the target computer in the target environment.

4.5.5.4 Verification of Outputs of Software Design Process (Annex A Table A-4)

- **Process Recognition**: An applicant may utilize the recognized process by using the ED-12C objective roadmap from the planning process. The roadmap may be used to clarify the usefulness and completeness of PDS data items and to determine if the process applied to the PDS can show compliance with ED-12C objectives.
- **Prior Product Certification/Approval**: PDS data items obtained from previous certifications may be applicable in satisfying the ED-12C objectives. For example, evidence of traceability and verification from prior certification/approval may be used for demonstrating low-level to high-level requirement's compliance.
- **Reverse Engineering**: Data items of the previous development process may be used to generate missing or incomplete data items, such as deriving the design architecture from the Source Code.
- **Restriction of Functionality**: A restriction of PDS functionality may provide a reduction in the amount of applicable PDS functionality subject to software design process objectives.
- **Product Service History**: If service history is deemed adequate to mitigate the need for providing low-level requirements, then it also mitigates the need to satisfy objectives pertaining to the quality of low-level requirements. Similarity of operating environments may provide empirical evidence that supports the accuracy of algorithms. Adequacy of the software architecture may be shown using service history if the data needed to assess the architecture's compatibility with the target computer is collected (for example, Central Processing Unit (CPU) utilization, memory utilization, throughput analysis and timing, and Input and/or Output (I/O) bandwidth).
- **Formal Methods**: This could provide mathematical proofs of correctness and accuracy of the algorithms. The use of mathematical proofs may be used as a means of verification or to support verification.
- **Audits and Inspections**: Reviews of PDS data items, such as requirements and safety analyses, may provide evidence of accurate and consistent requirements. The PDS requirements specification can be audited against documented standards. In some cases, the PDS supplier may be able to provide relevant test, review, and analysis data for the stand-alone PDS. Such data may be used to supplement tests on the target computer in the target environment (CPU utilization, memory utilization, throughput analysis and timing, and I/O bandwidth). It may be possible to show that the partitioning

integrity of the PDS is maintained by analysis of the partitioning design implementation.

4.5.5.5 Verification of Outputs of Software Coding and Integration Processes (Annex A Table A-5)

- **Process Recognition**: An applicant may utilize the recognized process by using the ED-12C objective roadmap from the planning process. The roadmap may be used to clarify the usefulness and completeness of PDS data items and to determine if the process applied to the PDS can show compliance with ED-12C objectives.
- **Prior Product Certification/Approval**: PDS data items obtained from previous certifications/approvals may be applicable in satisfying the ED-12C objectives. For example, evidence of traceability and verification from prior certification/approval may be used for demonstrating Source Code to low-level requirements compliance and Source Code verifiability (through test, demonstration, inspection, or analysis).
- **Reverse Engineering**: Elements of the process such as available code walkthroughs from the PDS originator may be used to show conformance to standards (identify trends showing adherence to standards). It may be possible to show that the coding process has not introduced any errors. This may be accomplished by review of the PDS Source Code, by verification that algorithms used satisfy the intended functionality, or by review of PDS code expected behavior under typical normal and abnormal conditions.
- **Restriction of Functionality**: A restriction of PDS functionality may provide a reduction in the amount of applicable PDS functionality subject to software coding and integration processes objectives.
- **Product Service History**: Generally not applicable for these objectives.
- **Formal Methods**: This could provide mathematical proofs of correctness and accuracy of the requirements and algorithms. Formal methods may be used to verify data flow analysis in the PDS. The use of mathematical proofs may be used as a means of verification or to support verification.
- **Audits and Inspections**: Reviews of PDS data items such as code walkthroughs results, complexity metrics, past test and integration results, and evidence of traceability can help satisfy objectives of this process.

4.5.5.6 Testing of Outputs of Integration Process (Annex A Table A-6)

Alternative methods are not generally applicable for this process. However, service history can show compliance with the objective of compatibility of Executable Object Code with the target computer. For testing of outputs of the integration process, all of the ED-12C data are available and the PDS is fully integrated into the system. Considerations include PDS testing for performance, functionality, robustness, limitations, restrictions, and input testing (normal, abnormal, and out of range). Also, system testing addresses system performance, functionality, software/software integration, and hardware/software integration.

4.5.5.7 Verification of Verification Process Results (Annex A Table A-7)

Substantiated and justified alternative methods may apply for the objectives contained in Table A-7.

- **Process Recognition**: An applicant may utilize the recognized process by using the ED-12C objective roadmap from the planning process. The roadmap may be used to clarify the usefulness and completeness of PDS data items and to determine if the process applied to the PDS can show compliance with ED-12C objectives.

- **Prior Product Certification/Approval**: PDS data items obtained from previous certifications may be applicable in satisfying the ED-12C objectives.
- **Reverse Engineering**: Generally not applicable for these objectives. However, reverse engineering can be used to augment test coverage analysis.
- **Restriction of Functionality**: The effort to satisfy these objectives may be reduced by this method.
- **Product Service History**: Analysis of service history data may be used to support the validity of the original verification process. For example, analysis of the defect data for patterns and trends to show that the original test suite remains valid.
- **Formal Methods**: Generally not applicable for these objectives.
- **Audits and Inspections**: Use of previous reviews and audits to satisfy the objectives of the current project. The applicant may be able to obtain evidence to show that test procedures used were correct and traceable to requirements, test results were verified, and any discrepancies have been identified and explained.

4.5.5.8 Software Configuration Management Process (Annex A Table A-8)

Alternative methods are not generally applicable for this process. However, the use of PDS may add the consideration of protection of proprietary data rights. For protection of proprietary data rights, the objectives may be met within an escrow (third-party data holding arrangement) or similar independent data holding arrangement between the applicant and the PDS supplier.

The mechanism to control the configuration of the PDS should be documented and is a function of the system using the PDS.

4.5.5.9 Software Quality Assurance Process (Annex A Table A-9)

Quality assurance aspects may be met by the results of audits and inspections of the PDS development process or by evidence from prior product certification/approval.

4.5.5.10 Certification Liaison Process (Annex A Table A-10)

Alternative methods are not generally applicable for this process.

The PSAC should define the framework for mapping the evidence from the PDS to the intent of each process in Annex A, and the Software Accomplishment Summary (SAS) should define the details of the mapping to each objective in Annex A. The details should include the PDS certification credit sought (full or partial), assumptions, evidence of compliance, and any activities accomplished during the project in order to satisfy each Annex A objective.

Compliance substantiation is a function of the system approval and is not unique to the PDS.

4.6 DP #6: TRANSITION CRITERIA

Reference: ED-12C: Sections 3.0, 3.3, 4.1, 4.2, 4.3, 5.1.2, 5.2.2, 5.3.2, 5.4.2, 8.1, 8.2, 11.2, 11.3, 11.4, 11.5; Annex A (Table A-1, objective 2 and Table A-9, objective 4); and Annex B

ED-109A: Sections 3.0, 3.3, 4.1, 4.2, 4.3, 5.1.2, 5.2.2, 5.3.2, 5.4.2, 8.1, 8.2, 11.2, 11.3, 11.4, 11.5, 12.4.1.3, 12.4.3; Annex A (Tables A-1 and A-9); and Annex B

Keywords: integral process; life cycle process; transition criteria

4.6.1 Scope

A number of questions regarding transition criteria have been raised. The questions to be addressed in this discussion paper are:

- What is the rationale for transition criteria in ED-12C/ED-109A?
- Is there a standard for accepting transition criteria?
- Why does ED-12C/ED-109A not define a minimum set of transition criteria for each life cycle process or for entering each elementary development process?
- Are transition criteria for integral processes necessary?
- What assurance activities and evidence are needed to ensure transition criteria are satisfied for ED-12C/ED-109A Table A-9, objective 4?
- What are the possible benefits of transition criteria in the ED-12C/ED-109A processes?
- How could transition criteria be defined?

4.6.2 What is the rationale for transition criteria in ED-12C/ED-109A?

Although ED-12C/ED-109A deliberately does not define any life cycle model, it does recognize that gates between life cycle processes can be important. These gates help to mitigate the risks of moving from one life cycle process to the next before the data for that process are defined in enough detail to prevent introduction or propagation of error. As an example, if you start to design or code before the requirements are reviewed, there is risk of error being introduced or propagated and the potential for rework. Note however, that it is not necessary to complete all of the requirements review before starting design or code.

Refer to ED-12C/ED-109A sections 3.3 and 4.3.b for additional information and guidance on transition criteria.

4.6.3 Is there a standard for accepting transition criteria?

Transition criteria are highly dependent on the life cycle model used. ED-12C/ED-109A does not place restrictions on what is acceptable, but transition criteria that are quantifiable, documented, and reduce error introduction or propagation should be considered acceptable. However, it is up to the developer to determine what the criteria should be.

4.6.4 Why does ED-12C/ED-109A not define a minimum set of transition criteria for each type of life cycle or for entering each elementary development process?

Although it might be possible to define generic transition criteria for each life cycle process, this would be specifying "how" rather than providing clarification of the objectives of ED-12C/ED-109A. Specifying "how" is not the intent of ED-12C/ED-109A or of this discussion paper. It is up to the developer to provide the specifics of the transition criteria.

4.6.5 Are transition criteria for integral processes necessary?

ED-12C/ED-109A section 3.3 only uses the word “process” when it is defining transition criteria and does not restrict the application of transition criteria to a particular type of process; therefore, integral processes do need transition criteria.

Such activities as baselining, release, and status reporting all need some kind of trigger events. In addition, quality assurance activities are driven by criteria for the level and timing of involvement. Some confusion may arise when configuration management activities are included as transition criteria for development processes. For example, consider design transition criteria that states, "Only requirements under configuration control can be input to the design activity." This criterion does not specify when the requirements are put under configuration control, only that the activity has occurred. There would likely be transition criteria as part of the configuration management plan identifying when it is acceptable to place the requirements under configuration control. Putting a data item under configuration control too early or too late in a process could result in unnecessary costs; this should be determined and negotiated as part of the configuration management planning process.

It should be noted again that ED-12C/ED-109A does not specify what the transition criteria should be; this is up to the developer and will vary depending on the application and process details of the organization.

4.6.6 What assurance activities and evidence are needed to ensure transition criteria are satisfied for ED-12C/ED-109A Table A-9, objective 4?

Typically, a Software Quality Assurance (SQA) organization sets up the plans that require auditing of the various activities associated with the development, verification, and configuration management processes. Their goal is to ensure compliance with the defined plans which contain transition criteria for the processes. ED-12C/ED-109A Table A-9 objective 4 ensures that activities are completed in compliance with defined transition criteria. This assurance is typically accomplished by the SQA organization. The SQA plans should include a method to record whether SQA personnel made an evaluation and whether the audited activities contain evidence that the activities satisfied the associated transition criteria. The resulting SQA record(s) that document this evaluation is an accepted means of providing the evidence that ED-12C/ED-109A Table A-9, objective 4 has been satisfied. Achieving the objective generally implies that quality assurance personnel understand the transition criteria, are able to clearly observe that the transition criteria have been met for a particular process through a review of the output of the process, and have documented the review as objective evidence that such a review was performed.

4.6.7 What are the possible benefits of transition criteria in the ED-12C/ED-109A processes?

Transition criteria that are documented, followed, and assured provide some evidence of a controlled and managed software life cycle process. A controlled process is the primary emphasis of ED-12C/ED-109A.

From a program management perspective, transition criteria allow evidence of where in the software process you are at any given time and can be one mechanism for early assurance that the outputs of the relevant integral process activities for that life cycle activity are produced and that the related ED-12C/ED-109A objectives are achieved. The potential benefits of transition criteria are the same as the benefits from any strong process: reduced risk, reduced defects, better safety, and predictable schedules.

Whether explicit or implicit, all software life cycles have transition criteria. Transition criteria would be similar to a call tree for a software program: they provide the gates and the sequencing of the different activities of the life cycle process. Having them explicitly defined allows them to be evaluated to ensure the life cycle process is completely defined.

4.6.8 How could transition criteria be defined?

Although ED-12C/ED-109A does not define what the transition criteria should be, there are some program management benefits to not specifying transition criteria which are either too rigid or too flexible. As an example, when overly restrictive transition criteria are defined, the benefits of using strong transition criteria to achieve strict process control may be realized, but unnecessary schedule and cost impacts may occur. On the other hand, overly flexible transition criteria may meet the ED-12C/ED-109A objectives without realizing the potential benefits in cost reduction and project risk reduction due to the time and effort spent on later rework. The best approach is to define transition criteria in such a way that the benefits are realized without unnecessary schedule and cost impact. For example, processes or activities that are determined to have no interdependencies could proceed independently with no transition requirement (with concurrence from the certification authority). In addition, it may be possible to show that even with processes or activities that have some interdependencies, errors cannot be propagated or that propagated errors can be identified and eliminated and therefore transition criteria could be less stringent.

4.7 DP #7: DEFINITION OF COMMONLY USED VERIFICATION TERMS

This Discussion Paper has been removed, as approved by the WG-71/SC-205 committee.

4.8 DP #8: STRUCTURAL COVERAGE AND SAFETY OBJECTIVES

Reference: ED-12C: Sections 6.4.4 and 6.4.4.2, and Annex A (Table A-7)

Keywords: 14CFR/CS XX.1309; safety objectives; structural coverage

4.8.1 Introduction

The purpose of this discussion paper is to discuss whether or not structural coverage analysis achieves any of the safety objectives detailed in 14CFR/CS XX.1309 (where 14CFR is Title 14 of the United States Code of Federal Regulations, CS is the EASA Certification Specifications, and XX is the applicable part).

4.8.2 What does 14CFR/CS XX.1309 require and how does it apply to software?

14CFR/CS XX.1309 requires that the probability of an aircraft failure condition occurring is shown to be inversely proportional to its effect. Software systems do not physically break or wear out; therefore, the probability of their failure cannot be defined within the scope of ED-12C. Software failure refers to incomplete or erroneous designs or implementations that result in anomalous behavior.

4.8.3 What does structural coverage analysis achieve?

Structural coverage analysis is a means by which an applicant can determine how much of the code has been exercised by their requirements-based testing.

Although structural coverage analysis is sometimes performed in conjunction with low-level requirements-based testing, it should be noted that there are certain aspects of a system that may not be testable as part of low-level requirements-based tests. These aspects include the interface between the software and the hardware.

NOTE: *Reference FAQ #43 (section 3.43 of this document) for more detail on this subject.*

4.8.4 What does structural coverage analysis not achieve?

There are various aspects of the compliance process to which structural coverage analysis makes no contribution. Structural coverage analysis alone cannot determine:

- whether all the requirements have been implemented;
- whether the requirements were correctly interpreted in the design; or
- whether the design was correctly implemented in the code.

4.8.5 Conclusion

Structural coverage analysis allows an applicant to assert that the code has been exercised as stated by the ED-12C Annex A coverage criteria for the software level. As such, it provides partial evidence for compliance with 14CFR/CS XX.1309 while showing that low-level errors which might contribute to failure conditions have been eliminated. Other processes should be used in conjunction with structural coverage analysis to enable the applicant to demonstrate that they have completely met the safety objectives.

4.9 DP #9: ASSESSMENT AND CLASSIFICATION OF OPEN SOFTWARE PROBLEMS

Reference: ED-12C: Sections 2.2.2.d, 7.2.1, 7.2.3, 7.2.5, 7.2.6, 8.3, 11.4, 11.17, 11.20.j, 12.1, 12.3.4.3

ED-109A: Sections 2.2.2.d, 7.2.1, 7.2.3, 7.2.5, 7.2.6, 8.3, 11.4, 11.17, 11.20.j, 12.1, 12.3.4.3, 12.4.6, 12.4.9, and 12.4.11.2.8

Keywords: open Problem Report; Software Accomplishment Summary (SAS)

4.9.1 Background and Purpose

ED-12C/ED-109A requires that the Software Accomplishment Summary (SAS) provide a summary of Problem Reports that are unresolved at the time of approval. The purpose of this DP is to provide an assessment and classification methodology for open Problem Reports (OPRs) that are unresolved at the time of approval.

A goal of the software life cycle processes is to achieve zero OPRs at the time of SAS submittal for approval. However, if Problem Reports are open at this time, then the objective is to minimize their number and analyze their impact at the system level, in any software release presented for approval. A high number of OPRs may indicate compromised software assurance and may put an applicant's ability to obtain approval at risk.

4.9.2 Discussion Paper Overview

This DP provides recommendations for the following:

- Safety-oriented OPR assessment and classification methodology.
- Documentation of OPR assessment data.
- Roles of software developer, system integrator and system/equipment manufacturer as related to OPRs.
- OPR assessment associated with reused software components.

4.9.3 Applicability

This DP applies to software levels A through D per ED-12C and assurance levels AL 1 through AL 5 per ED-109A.

4.9.4 Definitions

Refer to the glossary of ED-12C/ED-109A (Annex B) for the definition of the terms used herein.

4.9.5 Discussion

The potential impact of software OPRs on system/equipment functionality and safety should be thoroughly understood. The recommended activities to achieve a thorough understanding of the potential impact of OPRs are:

a. Define OPR classifications.

b. Perform OPR assessment.

c. Document OPR assessment results in the SAS.

d. Provide OPR assessment results to system integrator or system/equipment manufacturer to confirm potential effect at the system level.

e. Evaluate OPRs associated with reused software.

These topics are discussed in the following sections.

4.9.5.1 OPR Classifications

The following OPR classification scheme could be used to categorize problems based on their potential impact to the system and its software. The example classifications shown below are presented in decreasing order of consequence.

Type 0 (safety impact): A problem whose consequence is a failure, under certain conditions of the system, with an adverse safety impact.

Type 1 (functional failures): A problem whose consequence is a failure that has no adverse safety impact on the system/equipment. This classification could be divided into the following two sub-types:

- Type 1A (significant functional failure): A failure with a significant functional consequence. The meaning of “significant” needs to be defined in the context of the related system.
- Type1B (non-significant functional failure): A failure with no “significant” functional consequence.

Type 2 (non-functional fault): A problem that is a fault that does not result in a failure.

Type 3 (deviations from plans or standards): Any problem that is not Type 0, 1, or 2, but is a deviation to the software plans or standards. If agreed between system integrator or system/equipment manufacturer and software supplier, this type could be divided into two sub-types:

- Type 3A (significant deviation from plans or standards): A significant deviation whose effects could be to lower the assurance that the software behaves as intended and has no unintended behavior.
- Type 3B (non-significant deviation from plans or standards): A non-significant deviation that does not affect the assurance obtained.

Type 4 (all other types of problems): All other OPRs that do not fall into the above classification types 0 to 3. Due to the mutually exclusive nature of these classifications, the problems of type 4 are of no functional consequence. In many cases these problems might be described as typographical errors.

When a single reporting system is used to capture all planned changes in addition to Problem Reports, it is up to the software developer to filter their reporting system for those items that are problems for the current release.

4.9.5.2 OPR Assessment

Each OPR should be assessed to determine:

- classification per the definitions in section 4.9.5.1,
- impact due to functional limitations and operational restrictions,
- likelihood of occurrence,
- deviation(s) from software plans and/or standards,
- interrelationships with other OPRs, if needed,
- safety mitigations, when found to be necessary,
- justification for allowing the OPR to remain open,
- root cause, and
- anticipated resolution schedule, if known.

OPRs determined to be Type 0 should be corrected prior to approval or adequate mitigation means (for instance operational restrictions) should be defined, such that there is no adverse safety effect at the system or equipment level.

OPRs determined to be Type 1 should be assessed at the system or equipment level. If necessary, appropriate restrictions should be defined to ensure safety. Additionally, an assessment of no safety impact on the system or equipment should be justified.

OPRs determined to be Type 2 should justify that the error cannot cause a functional failure. For simple cases, this justification may be a statement based on engineering judgment. In some specific cases, this justification may imply specific additional validation and/or verifications activities.

OPRs determined to be Type 3 should be assessed for the extent or nature of deviations from the plans that may contribute to lower assurance of the software. In some cases, additional software life cycle activities may be appropriate.

OPRs determined to be Type 4 do not require any further assessment beyond the type classification.

The root cause and anticipated resolution schedule should be established as follows. This information should be documented in the software life cycle data (for example, Problem Reports, Software Configuration Management Records, or Software Quality Assurance Records).

- Root cause: Identification of the deficiency or deficiencies associated with the software life cycle processes that lead to the error for types 0, 1, and 2. In some cases, a root cause can contribute to multiple OPRs. Consequently, those OPRs may be grouped according to the root cause. In addition to identifying the root cause, the steps taken to prevent a future occurrence due to this root cause should be documented.
- Anticipated resolution schedule: Expected schedule for resolving the OPR. This schedule may be event driven.

4.9.5.3 OPR Assessment Results in the SAS

The following ED-12C/ED-109A sections provide relevant guidance on problem reporting and functional limitations:

- 2.2.2.d System Aspects Relating to Software Development (or lower paragraphs as more appropriate)
- 7.2.3 Problem Reporting, Tracking and Corrective Action
- 7.2.5 Change Review
- 7.2.6 Configuration Status Activity

- 11.17 Problem Reports
- 11.20 Software Accomplishment Summary
- 12.1.1 Modifications to Previously Developed Software

ED-12C/ED-109A section 11.20.j regarding software status states that the SAS should provide a summary of Problem Reports unresolved at the time of approval. The Problem Report summary includes functional limitations, operational restrictions, potential adverse effect(s) on safety, deviation(s) from software plans or standards, justification for allowing the Problem Report to remain open, and mitigations, if any. Although the applicant may submit an OPR summary, the applicant may find that supplying more data can result in a better understanding by the certification/approval authority.

It is recommended that the SAS contain the following regarding OPRs:

a. Identification: Identification of each OPR including configuration identification number (that is, the OPR number) (see ED-12C/ED-109A sections 11.17 and 11.20).

b. Description: A brief description of each OPR (see ED-12C/ED-109A section 11.17).

c. OPR assessment results (per 4.9.5.2 above), including:

- Classification: Classification of each OPR (see section 4.9.5.1 of this DP).
- Impact: The impact of the OPR at a system level (see ED-12C/ED-109A sections 7.2.5 and 11.17) and any functional limitations or operational restrictions that exist or may need to be proposed (see ED-12C/ED-109A section 11.20). Some OPRs exist because the software currently does not function as well as, or just differently than, originally intended (that is type 0 and 1). The disposition of these OPRs may represent a deviation from the intended function defined by the system requirements. Subsequently, some system level documents may need to be changed or updated. Therefore, the SAS should clearly identify any changes to the intended function of the system.
- Likelihood of Occurrence: An assessment of when the problem will be encountered and the corresponding trigger conditions.
- Deviation(s) from software plans or standards.
- Interrelationships with other OPRs, if needed. For example the safety or operational impact could be more significant given the interrelationship of multiple OPRs.
- Safety mitigations, when found to be necessary.
- Justification for why the system can be approved with the OPR (see ED-12C/ED-109A sections 7.2.5 and 11.20).

4.9.5.4 Assessment of OPR by System Integrator or System or Equipment Manufacturer

It should be noted that even if the software developer has sufficient knowledge to explain the functional effect of the OPR at the system or equipment level, only the system integrator or system manufacturer can assess or confirm the potential effect at the system level.

As such, the software developer should provide the SAS information pertaining to OPRs per section 4.9.5.3 to the system integrator or system manufacturer. The system integrator or system manufacturer should then evaluate the OPRs for any impact on the safety assessment of the system. For the airborne domain, the aircraft manufacturer should participate in this evaluation. For the CNS/ATM domain, the air navigation service provider should participate in this evaluation.

Software incorporated into Technical Standard Order (TSO) or European Technical Standard Order (ETSO) authorized equipment or Integrated Modular Avionics (IMA) platform poses a particular concern. Each subsequent user or integrator of Technical Standard Order Authorization or European Technical Standard Order Authorization (TSOA/ETSOA) or IMA platform should evaluate the OPRs identified in the software developer SAS for any safety assessment impact. Unless significant detail is provided in the SAS, the integrator will probably need additional information on the OPRs in order to perform this evaluation.

4.9.5.5 Assessment of OPRs Associated with Reused Software

The list of OPRs associated with previously developed software should be carefully evaluated for appropriateness of reuse and safety impact. Additional information on the OPRs may be needed in order to perform this evaluation. It may be necessary to resolve some of the OPRs associated with the previously developed software to ensure the system or equipment complies with approval requirements.

4.10 DP #10: CONSIDERATIONS ADDRESSED WHEN DECIDING TO USE PREVIOUSLY DEVELOPED SOFTWARE (PDS)

Reference: ED-12C: Sections 12.1 and 12.3.4

ED-109A: Sections 12.1, 12.3.4, and 12.4

Keywords: commercial off-the-shelf software; COTS software; non-developmental item; NDI; previously developed software; PDS

4.10.1 Discussion

The decision to use PDS includes both technical and business (cost and schedule) considerations. The final decision needs to balance these factors to arrive at an optimal decision, which ensures that the necessary safety requirements of the application are maintained. Cost and schedule considerations on the use of PDS are business issues and should not compromise the certification/approval or safety of a system. However, the technical issues on PDS usage should be addressed and are at times affected by the cost and schedule issues. ED-12C/ED-109A does not address cost or schedule issues. This discussion paper's only purpose is to provide a list of example considerations. The first section covers the technical considerations and the second section provides some business considerations.

The following terms are used in this discussion paper:

- Certification/Approval Software Data Package – The software life cycle data provided to show compliance with ED-12C/ED-109A objectives in support of the software aspects of the certification/approval process.
- Previously Developed Software (PDS) – PDS encompasses any software developed for use on another application. This includes commercial off-the-shelf software (COTS) products and software developed to previous or current software safety standards.
- PDS supplier –The developer of the PDS. This could range from a third party commercial vendor providing COTS software to a development organization reusing software from a previous certification/approval project within the same organization.
- Escrow – A legal agreement for the retention of software products or data by a third party for distribution to the user in the event that the PDS supplier is unable to meet contractual agreements or data retention requirements.
- Non-developmental item (NDI) – An NDI is used in industry to describe the reuse of hardware or software that may contain PDS.

4.10.2 Technical Considerations

The technical analysis should consider the following issues (this is not intended to be an all-inclusive list):

- Level of assurance of the system to be developed and the use of PDS in the context of the system safety assessment.
- PDS supplier's experience with the application domain and certification/approval requirements.
- Compatibility or similarity between the defined PDS environment and the application domain, with emphasis on safety requirements.
- Availability, relevance, and quality of applicable software life cycle data (that is, the new or prior certification/approval software data package).
- Availability and quality of PDS supplier's software life cycle data, including PDS interface specifications.
- Applicability, availability, and quality of service history data.
- Level of effort needed to bring PDS into ED-12C/ED-109A compliance at the needed software level.
- PDS supplier's commitment and capability to support software configuration management objectives, such as version control and maintenance.
- PDS supplier's commitment and capability to support certification/approval requirements, such as independent audits.
- Software, hardware, and tools availability and obsolescence issues (may include hardware and software components, portability, technology evolution, etc.).
- Applicability of the PDS to the application's requirements.
- Modifiability, including:
  - modifications needed to the PDS,
  - modifiability of the PDS by the PDS supplier or acquirer,
  - impact on the certification/approval software data package applicability,
  - ease of suppression or deactivation of unwanted functionality,
  - configuration management and change control, and
  - impact of modifications to the PDS outside the scope of this application (change history visibility).
- Long term maintenance, including product retirement, product enhancements, and tools.
- Software escrow.
- Feasibility or consideration of an alternative methods approach to providing equivalent software assurance.

4.10.3 Business Considerations

4.10.3.1 Cost Considerations

The cost-benefit or business analysis should consider the following issues (this is not intended to be an all-inclusive list):

1. Cost to acquire the PDS:
    - Cost and schedule compatibility (that is, the cost to incorporate the PDS into the target system versus to build your own software application).
    - Purchase cost of PDS.
    - Certification/approval costs of PDS (cost of adapting or altering the PDS to satisfy the objectives of ED-12C/ED-109A), including:
        - Additional costs caused by lack of available data from PDS supplier.
        - Cost of complete certification/approval software data package available from PDS supplier.
2. Cost to maintain the PDS:
    - Responsibility for software maintenance (that is, identifying the PDS supplier or the acquirer or both who will perform the software maintenance).
    - Configuration management and change control (for example, identifying who does it and when).
    - Intended application service life and long-term maintenance plan (including feature modification and error correction related change control).
3. Cost of run-time licenses and buyout.
4. Cost of Source Code buyout.
5. Cost to complete certification/approval software data package, including:
    - Who provides the resources to complete the work?
    - How much effort remains to be completed?
6. Experience with the PDS supplier (business relationship) and their financial stability and strength.
7. Responsibility for resources (cost) for responding to certification/approval authority concerns and requests, including audits.
8. Cost of escrow.
9. Cost of addressing any third party supplier impacts to the PDS.

4.10.3.2 Schedule Considerations

The project scheduling analysis should consider the following issues (this is not intended to be an all-inclusive list):

1. Impact on project schedule, including certification/approval schedule.
2. Schedule compatibility (that is, time to incorporate the PDS into the target system versus time to build your own software application, and ability of the PDS supplier to support the PDS user’s schedule).
3. Near-term versus long-term schedule trade-off (for example, pre-production delivery versus production delivery).

4. Critical path analysis (that is, considering how the PDS use affects the critical path of the schedule).
5. Internally and externally imposed milestones, such as the following:
   - Make or buy decision points.
   - Certification/approval authority decision points.
6. Time to market.
7. PDS supplier interactions, including process reviews and audits and software life cycle process and data reviews.

4.11 DP #11: QUALIFICATION OF A TOOL USING SERVICE HISTORY

This discussion paper has been removed, as approved by the WG-71/SC-205 committee.

4.12 DP #12: OBJECT CODE TO SOURCE CODE TRACEABILITY ISSUES

Reference: ED-12C/ED-109A: Sections 6.4.4.2, 6.4.4.3, and 12.3.2.3; and Annex A Table A-7

Keywords: compiler; object code; Source Code; structural coverage; traceability

4.12.1 Introduction

ED-12C/ED-109A section 6.4.4.2.b states: *"Structural coverage analysis may be performed on the Source Code, object code, or Executable Object Code. Independent of the code form on which the structural coverage analysis is performed, if the software level is A and a compiler, linker, or other means generates additional code that is not directly traceable to Source Code statements, then additional verification should be performed to establish the correctness of such generated code sequences.*

NOTE: *'Additional code that is not directly traceable to Source Code statements' is code that introduces branches or side effects that are not immediately apparent at the Source Code level. This means that compiler-generated array-bound checks, for example, are not considered to be directly traceable to Source Code statements for the purposes of structural coverage analysis, and should be subjected to additional verification."*

ED-12C/ED-109A section Table A-7 objective 9 states: "*Verification of additional code, that cannot be traced to Source Code, is achieved.*" This objective is applicable for Level A/AL1.

When performing structural coverage at the Source Code level for Level A software, it is necessary to establish whether or not the object code is directly traceable to the Source Code. Approaches to this assessment are discussed below. If the object code is not directly traceable to the Source Code, then additional verification should be performed. The additional verification is not addressed in this discussion paper.

4.12.2 Discussion

4.12.1 General Considerations

It should be first noted that the work involved in the process of demonstrating traceability from object code to Source Code may not be trivial. In undertaking any analysis, several issues should be considered, including, but not limited to:

- the extent of code optimization,
- compiler switch options, and
- object code inserted by the compiler.

Each issue is addressed below:

- The extent of code optimization should be considered, because the more extensive the optimization, the more difficult it may be to demonstrate traceability. Optimization can lead to significantly restructured object code, including the removal of code, in order to achieve improved performance.
- Compiler switch options should be considered, because switches used during the compilation for test purposes should be identical to those used for the final target object code build. This is for obvious reasons, but can easily be overlooked.
- Object code may be inserted by the compiler to perform functions such as range checks or index checks. Such code may not be traceable to Source Code, yet may be appropriate to remain in the final object code build and operational system.

4.12.2 Approach to Traceability Analysis

As can be established from ED-12C/ED-109A, the purpose of the object code to Source Code traceability analysis is to identify added functionality not visible at the Source Code level. This analysis is not intended to demonstrate that every object code statement is correct. Some programming languages may contain features that make detection of added functionality difficult. In all cases, the added functionality should be summarized so that its behavior can be understood and verification techniques can be applied.

One approach to this analysis is to produce code with fully representative language constructs of the application (for example, case statements, if-then-else statements, loops, etc.) and to analyze the resultant object code. Also, it is important that the individual constructs are completely representative of the constructs used in the target object code build, including complex structures (for example, nested routines). In some cases, object code may be data type dependent. The choice of the representative code constructs should be agreed with the appropriate certification authority. The link map should be examined to ensure that no unexpected library components are being incorporated into the target object code build.

Another valid approach is to examine the target object code build.

4.12.3 Conclusion

The purpose of the object code to Source Code traceability analysis is to detect any functionality added by the compiler. In some cases, the structure and content of the object code may differ slightly from the structure of the Source Code, as previously discussed. If comparisons performed on the representative code show that there are no functional differences, then it has been shown that the object code contains no added functionality.

This process is intensive and should be thorough. It should be approached logically and with full awareness that the outcome may be that coverage at the Source Code level is not sufficient. For this reason, the approach to structural coverage analysis should be addressed early during development, with the process outlined in this paper being a means to assess particular compilers for their suitability on the project.

Furthermore, this process should be discussed and agreed with the relevant certification authorities at an early stage, so that agreement can be achieved for compliance with ED-12C/ED-109A objectives.

4.13 DP #13: DISCUSSION OF STATEMENT COVERAGE, DECISION COVERAGE, AND MODIFIED CONDITION/DECISION COVERAGE (MC/DC)

Reference: ED-12C/ED-109A: Section 6.4.4.2 and Annex A Table A-7

Keywords: decision coverage; modified condition/decision coverage; MC/DC; statement coverage; structural coverage; verification

4.13.1 Introduction

ED-12C glossary defines structural coverage terms. This discussion paper describes the definitions and their implications for coverage analysis.

4.13.2 Descriptions and Implications

a. A **statement** is a programming construct defined by the programming language used for the Source Code.

For procedural languages like Ada, C, and C++, the language reference manuals describe the syntax of a statement. Programming languages are not consistent, and certain implementations of the same language may define and implement different types of statements. Model based languages complicate the definition even further. For most common languages, the definition of a statement is well understood and there is little argument. If the statement is different from the traditional definitions (such as, assignment, conditional, loop, procedure call, etc.), then the PSAC should define the interpretations as used by the programming language.

b. A **Boolean expression** is an expression that results in a TRUE or FALSE value.

Some languages interpret arithmetic values as Boolean values. If an expression provides an arithmetic value which is interpreted as Boolean values, then the expression is a Boolean expression. A Boolean expression may be a simple variable, or a language-dependent construct. Examples of Boolean expressions used in conditional statements are shown below:

- ***Example 1 – C Language:*** if Y then { … } else { … };

  In this example, “Y” is a Boolean expression

- ***Example 2 – Ada Language:*** if P in Pressure'Range then … else … endif;

  In this example, “P in Pressure'Range” is a Boolean expression

- ***Example 3 – Ada Language:*** An expression could be written as a predefined constant: if PPC603 then … ;

  In example 3, if the constant PPC603 were set to FALSE, the sequence of statements in the ‘then’ part would be eliminated. This type of construct is often used for conditional compilation. This is not considered a Boolean expression by our definition as it cannot result in both TRUE and FALSE.

c. An example of a decision is:

```
if [Boolean expression]
then
      [statement sequence];
else
      [statement sequence];
```

This decision chooses between two alternative statement sequences. In this example the same number of test cases is necessary to achieve both decision coverage and statement coverage.

A decision may select between several statement sequences. An example could be a case statement. An alternative in a case statement may choose one or more statement sequences. In the C programming language an example of this would be a case alternative followed by a second case alternative with no intervening break statement.

Another example is a language-required check followed by a conditional exception raise statement which will force a selection of two statement sequences. The check is a decision.

For **decision coverage**, two test cases are needed to cause the Boolean expression to be evaluated to both TRUE and FALSE and in turn cause both statement sequences to be executed.

```
if [Boolean expression]
then
    [statement sequence];
[statement sequence];
```

Statement coverage only requires the Boolean expression to be evaluated to TRUE, to force execution of the first statement sequence. For **decision coverage**, the Boolean expression will need to be evaluated to both TRUE and FALSE.

A case statement chooses between two or more alternative statement sequences. In some programming languages, one statement sequence can flow into another statement sequence unless a 'break' construct is used. In such situations, a single test case can achieve **statement coverage** of both statement sequences. Multiple test cases will be required to achieve **decision coverage**, since all values of the selecting expression necessary to choose between all of the case alternatives will need to be evaluated, including a value outside the range if a default is present.

The C language permits the use of 'ternary operators' as an alternative form of decision. For example:

```
(a < 2 ? … statement sequence : … statement sequence …);
```

This is equivalent to:

```
if (a < 2) then {… statement sequence…} else {… statement sequence …};
```

Each of the statement sequences yields the value of the last expression executed. The statement sequence could also simply be an expression. Therefore, the ternary operator may be treated as an expression and nested inside another expression. For example:

```
if (a > (b == 0 ? c + 2 : 27)) then …
```

The expression between the 'if' and 'then' is a Boolean expression. The expression preceding the '?' is also a Boolean expression because it also yields TRUE and FALSE. Both Boolean expressions are decisions because they both select between statement sequences. The ternary operator is therefore a nested decision.

d. A **Boolean operator** is used to combine two or more Boolean values to yield a Boolean value. In general this means the AND, OR, and XOR (exclusive OR) dyadic Boolean operators. The NOT operator operates on one value only, so is a monadic Boolean operator.

Other operators, although used heavily in hardware logic design are less common in programming languages. They may be interpreted by considering their basic elements. For example:

(x NAND y) may be interpreted as NOT (x AND y)

(w NOR z) may be interpreted as NOT (w OR z)

If the operands are not both Boolean values, then the operator is not a dyadic Boolean operator. For example:

if P in Pressure'Range then … else … end if; -- Ada language

'P' is a variable and *'Pressure'* is a type. The operation *'in Pressure'Range'* will result in TRUE or FALSE, but it does not combine two Boolean values.

Equality and inequality operators may operate on Boolean operands. For example:

if Y /= Z then … else … end if; -- Ada language

'Y' and *'Z'* are both Boolean variables. The not-equals relational operator may be treated as a dyadic Boolean operator (XOR).

In some programming languages the Boolean operator may be written as & instead of AND, and | instead of OR.

Some programming languages implement bitwise operators as well as logical operators. Bitwise operators are not, in general, Boolean operators.

Note that the short-circuit operators written as && or || in the C programming language and *'and then'* or *'or else'* in Ada, are not Boolean operators because they do not always require the second Boolean expression to possess a TRUE or FALSE value. In practice, this means that they do not always require the second expression to be evaluated, which may simplify testing requirements in some cases (as explained in the discussion below about Masking MC/DC).

e. A **condition** is a Boolean expression containing no Boolean operators except for the unary operator. It could be a Boolean variable; it could be the result of a relational operator, a function call which returns a Boolean value, and many others. Because a Boolean operator combines Boolean values (other than a NOT operator), a condition does not contain any Boolean operators. A NOT operator acts on a single operand, so it may be present in a condition. In the following programming construct (where *'a'* and *'b'* are integer variables, and 'x' is a Boolean variable):

if ((a < b) and f(2)) or (*not x*) then …

The three conditions are '(a < b)', 'f(2)', and '(not x)'. Each of them results in a Boolean value.

Short-circuit Boolean decisions are a sequence of conditions which are separated by short-circuit Boolean operators. For example:

if (((a > 0) && (b == 0)) || f(2)) then …

In any short-circuit sequence, irrespective of the brackets and the operators, the conditions are always evaluated left-to-right, and only evaluated as necessary. In the above example, if the first condition evaluates to FALSE, then the second is not even evaluated, and the execution continues with the evaluation of the third condition. If the first two conditions are evaluated to TRUE, then the third condition is not evaluated. Short-circuit forms are equivalent to sequenced if statements.

Short-circuit forms are popular, because they permit the user to control the order of evaluation of the conditions. A Boolean expression of the form

(((a > 0) & (b == 0)) | f(2)) is problematic if function 'f' has a side effect (say updating the global variable *'b'*). A compiler may choose the order in which the conditions are evaluated. The results of the Boolean expression will depend on the order of evaluation, and this may be different if the Source Code is instrumented by a coverage measurement tool.

f. **Modified Condition/Decision Coverage** may be accomplished by:

- varying just that condition while holding fixed all other possible conditions (known as unique cause MC/DC), or by
- varying just that condition while holding fixed all other possible conditions that could affect the outcome (known as masking MC/DC and short-circuit MC/DC).

Table 4-1 shows the truth table for the conditions and the decision outcome for this programming construct:

if A or (B and C) then … ; -- where A, B and C are Boolean variables

TABLE 4-1: TRUTH TABLE FOR DECISION (A OR (B AND C))

#	A	B	C	A or (B and C)
0	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE
2	FALSE	TRUE	FALSE	FALSE
3	FALSE	TRUE	TRUE	TRUE
4	TRUE	FALSE	FALSE	TRUE
5	TRUE	FALSE	TRUE	TRUE
6	TRUE	TRUE	FALSE	TRUE
7	TRUE	TRUE	TRUE	TRUE

g. **Unique Cause MC/DC:** One way that a condition can be shown to independently affect a decision's outcome is by varying just that condition while holding fixed all other possible conditions. From Table 4-1, the tests that can show how each condition independently affects the decision are:

For A – {0, 4}, {1, 5}, or {2, 6}

For B – {1, 3}

For C – {2, 3}

Given the above sets, the minimal sets of tests that show that every condition independently affects the decision is: {1, 2, 3, 5} or {1, 2, 3, 6}. This approach to MC/DC is referred to as "unique cause MC/DC".

h. **Masking MC/DC and short-circuit MC/DC** recognizes that to show A's independence, '(B and C)' must be FALSE. There are three ways that '(B and C)' can be FALSE, and any of them can be used for A's independent affect to be demonstrated.

Under masking MC/DC we can use:

For A – {0, 4}, {0, 5}, {0, 6}, {2, 4}, {2, 5}, {2, 6}, {1, 4}, {1, 5}, {1, 6}

For B – {1, 3}

For C – {2, 3}

So for masking MCDC the test vectors {1,2,3,4} would suffice.

Here are two examples of short-circuit MCDC:

1. The decision is written in C using the following notation:

   if (A || (B && C)) then …

   In this construct, if A is TRUE then the rest of the condition sequence is not evaluated, so we cannot verify the conditions B and C directly (although we can estimate what they are by reviewing the test cases and tests). If A is FALSE and B is FALSE then C is not evaluated because the decision can already be made. See Table 4-2 below.

TABLE 4-2: TRUTH TABLE FOR DECISIONS (A || (B && C))

#	A	B	C	A || (B && C)
1	FALSE	FALSE	X	FALSE
2	FALSE	TRUE	FALSE	FALSE
3	FALSE	TRUE	TRUE	TRUE
4	TRUE	X	X	TRUE

   Test cases which showed the condition values and decision outcomes would be sufficient to show MC/DC coverage, where X represents a don't care condition because the corresponding condition is not executed.

   As can be seen, the test vectors would demonstrate coverage of this construct.

2. The decision is written in C using the following notation:

   if ((A || B) && C) then …

   The truth table is shown in Table 4-3 for conditions and the decision outcomes.

TABLE 4-3: TRUTH TABLE FOR ((A || B) && C)

#	A	B	C	A || (B && C)
0	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE
2	FALSE	TRUE	FALSE	FALSE
3	FALSE	TRUE	TRUE	TRUE
4	TRUE	FALSE	FALSE	FALSE
5	TRUE	FALSE	TRUE	TRUE
6	TRUE	TRUE	FALSE	FALSE
7	TRUE	TRUE	TRUE	TRUE

   For MC/DC coverage, the test vectors {1, 3, 4, 5} could be used because they include the pairs of tests which verify each condition independently. The measurement of all the evaluated conditions would not be possible, as some conditions (shown shaded) are not evaluated during execution.

   As decision flow is affected by short-circuit operations, all of the decisions resulting from the conditions executed in their prescribed sequence could be covered using the test vectors {1,3,4}.

i. **Coupled Conditions:** If the same variable is used in multiple conditions of a Boolean expression, then the conditions may not be able to be executed with independence. Generally, this is due to one or more condition combinations not being possible. For example, if the code is verifying that a variable is outside the range 3 to 8, the code could be:

if (a < 3) or (a > 8) then …

The truth table is shown in Table 4-4.

TABLE 4-4: TRUTH TABLE FOR ((A < 3) OR (A > 8))

Test vector	(a < 3)	(a > 8)	Result
0	FALSE	FALSE	FALSE
1	FALSE	TRUE	TRUE
2	TRUE	FALSE	TRUE

Test vectors 0, 1, and 2 would provide MC/DC coverage. However, a value of 'a' could not be less than three and greater than eight at the same time. Therefore, test vector 3 is not possible and is not shown.

Given the expression:

(A or B) and (B or (A xor C))

The second occurrence of *'A'* has no independence pairs. To show the second *'A'*s independence, *'C'* must be held at a constant value, the second occurrence of *'B'* must be held FALSE, and *'(A or B)'* must be held TRUE. It is not possible to hold *'(A or B)'* TRUE when *'B'* must be held FALSE and *'A'* must vary between TRUE and FALSE.

4.14 DP #14: PARTITIONING ASPECTS IN ED-12C/ED-109A

Reference: ED-12C: Sections 2.1, 2.2.1, 2.2.2, 2.3, 2.4, 2.4.1, 2.5.1, 5.2.2, 5.2.3, 6.3.3, 6.4.3, 11.1, 11.3, 11.9, 11.10, 11.20, 12.3.2, and Annex A

ED-109A: Sections 2.1, 2.2.1, 2.2.2, 2.3, 2.4, 2.4.1, 2.6.1, 5.2.2, 5.2.3, 6.3.3, 6.4.3, 11.1, 11.3, 11.9, 11.10, 11.20, 12.3.2, 12.4.2, 12.4.4, 12.4.5.2, 12.4.11.2.4, and Annex A

Keywords: partitioning; protection; robust partitioning

4.14.1 What is partitioning?

ED-12C/ED-109A Section 2.4.1 states: "*Partitioning is a technique for providing isolation between software components to contain and/or isolate faults and potentially reduce the effort of the software verification process. Partitioning between software components may be achieved by allocating unique hardware resources to each component (that is, only one software component is executed on each hardware platform in a system). Alternatively, partitioning provisions may be made to allow multiple software components to run on the same hardware platform*."

ED-12C/ED-109A and other sources specify the expected or intended behavior for a partitioned system. For example, ARINC 653 specifies the requirements of partitioning in an operating system (OS) as the following: "*In order to isolate multiple partitions in a shared resource environment, the hardware should provide the OS with the ability to restrict memory spaces, processing time, and access to I/O (Input and Output) for each individual partition."* Shared (computer) resources are physical computation components that are used concurrently by several independent aircraft functions. These resources include the CPU (Central Processing Unit), Co-processors, FPU (Floating Point Processing Unit), MMU (Memory Management Unit), the physical memory, I/O-channels and, if available, file storage devices.

ED-12C/ED-109A addresses “software partitioning” in the following sections:

- Section 6.4.3.a mentions "Violations of software partitioning."
- Table A-4 of Annex A (Verification Of Outputs of Software Design Process) objective 13 states: *"Software partitioning integrity is confirmed."* [Note: Section 6.3.3.f is referenced in Table A-4 objective 13 and addresses *“partitioning integrity”.*]

NOTE 1: *Additional references to partitioning are included in the following ED-12C/ED-109A sections: 2.1, 2.2.1, 2.2.2, 2.3, 2.4, 2.4.1, 2.5.1, 2.5.5, 5.2.2, 6.3.3, 6.4.3, 11.1, 11.3, 11.9, 11.10, 11.20, and 12.3.2.*

NOTE 2: *Section 3.5 of ED-124 (entitled "Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations") provides guidance on partitioning for IMA systems. It has some particularly pertinent information regarding designing for robust partitioning and performing a partitioning analysis.*

The assumption in this discussion paper in regard to the use of the term “software partition” is not the mechanism of partitioning (section 4.14.3 of this DP), but rather the software component which has to be protected.

4.14.2 Where is partitioning used?

Partitioning provides, on the same resource: (1) independence of software components and processes, and (2) error containment. Partitioning enables the following development process goals:

- Supports the implementation of different software processes, components, or functions on the same processor.
- Supports the implementation of different software processes, components, or functions of different software/assurance levels on the same computation resources (for example, processors).
- Supports software updates or modifications without re-verification of unmodified software partitions.
- Isolates and protects a non-modifiable, partitioned software component from a user-modifiable software component that is not under the control of the original developer.
- Isolates and protects an approved software component from unapproved software components.
- Potentially reduces subsequent certification efforts for modified or additional software applications.

The partitioning strategy to be used should be described in the Plan for Software Aspects of Certification (PSAC) or Plan for Software Aspect of Approval (PSAA) and approved by the certification/approval authority.

4.14.3 Partitioning Rigor

Mechanisms to accommodate partitioning can range from pure software means to hardware protections. In some cases, there will be a combination of both mechanisms.

A less rigorous form of partitioning, for lower software/assurance levels, is to use only compiler and linker mechanisms to implement partitioning. This will lead to partitioning by design, but the user has to rely on the correctness of these tools.

A more rigorous form of partitioning is to use mutually exclusive hardware to achieve isolation and protection. Memory bank switching is an example for space protection of mutually exclusive hardware.

Acceptable techniques and criteria for partitioning strategy should be discussed by the applicant and the certification/approval authorities to reach an agreement on an acceptable level of partitioning before implementation of the chosen solution in a real application. The acceptability of selection of the partitioning solution should be determined relative to the software/assurance level (for example, a pure software means may be acceptable for lower software/assurance level applications).

4.14.4 Certification Background

Airborne system regulations contain requirements that may impact the design of the system architecture, including requiring that functions and components are physically separated, isolated, and protected from one another.

The following excerpts provide analogous system and equipment level regulations which illustrate how software partitioning might be addressed.

(1) CS 25.1309(a) states: "*The aeroplane equipment and systems must be designed and installed so that:*

(1) Those required for type certification or by operating rules, or whose improper functioning would reduce safety, perform as intended under the aeroplane operating and environmental conditions.

(2) Other equipment and systems are not a source of danger in themselves and do not adversely affect the proper functioning of those covered by sub-paragraph (a)(1) of this paragraph."

(2) AMC 25.1309 introduces the fail-safe design concept and the objectives which apply (for example, single failure should not be catastrophic regardless its probability and subsequent detected or latent failures should be assumed). AMC 25.1309 also introduces design principles or techniques that should be used to provide a fail-safe design. In addition, common cause failure analysis should include, when applicable, the fail-safe design concept.

(3) One of the design principles in AMC 25.1309 promotes: "Isolation (especially physical or spatial separation) and independence of systems, components, and elements so that the failure of one does not cause the failure of another."

NOTE 1: *The FAA regulations and advisory materials contain similar concepts.*

NOTE 2: *CNS/ATM systems also have safety requirements mandated by the approval authority.*

The requirement of isolation leads to the implementation of a robust partitioning mechanism in cases where shared computer resources are utilized for aircraft level functional applications.

4.14.5 Recommendations for Robust Partitioning Mechanisms

Partitioning for shared computer resources should consider the following (this is not an exhaustive list):

- For space partitioning:
  - Protection of code memory, data memory, registers, and input/output buffers.
  - Persistent storage locations (for example, data memory), assigned to a software partition, write-able only by that partition.
  - Context data (for example, processor registers, CPU-caches) used by a task preserved or flushed, as appropriate, when control is transferred to another partition.
  - Data flow and communications between partitions.

- For time partitioning:
  - Protection of the processing and communication assigned to a partition.
  - The consistent order of execution between communicating partitions.
  - Deterministic scheduling (processor and communication).
  - Guaranteed access for each software partition to a prescribed set of hardware resources for a prescribed period of time and at a prescribed rate and, if necessary, at a prescribed point in time.
- Operating system (if applicable):
  - The services provided by the operating system may be used by different tasks in different partitions. This may occur deliberately in the case where a task using a system service is interrupted (for example, software partition deadline expires). It may also occur when a system service is not released correctly. In short, partitions use system services without knowledge of other partitions and their service utilization. Therefore, there should be a means to protect and recover the system services or provide for degraded operations for all software partitions resident on a single processing resource.

To summarize, the recommendations for robust partitioning are:

- A software partition should not be allowed to contaminate another partition's code, I/O, or data storage areas.
- A software partition should be allowed to consume shared processor resources only during its period of execution.
- A software partition should be allowed to consume shared I/O resources only during its period of execution.
- Failures of hardware unique to a software partition should not cause adverse effects on other software partitions.
- Software providing partitioning should have at least the same software/assurance level as the highest level of the partitioned software applications.

4.14.6 Clarification of the Distinction of Protection Mechanisms

ED-12C/ED-109A section 2.4.1 states: *"The software life cycle processes should address the partitioning design considerations. These include the extent and scope of interactions permitted between the partitioned components, and whether the protection is* ***implemented by hardware or by a combination of hardware and software."*** [Bold text added for emphasis.]

ED-12C/ED-109A section 5.2.3.a states: *"... This protection can be enforced by hardware, by software, by the tools used to make the change, or by a combination of the three."*

Both statements of ED-12C/ED-109A contain recommended practices, but applicants are not limited to these. To define specific implementation rules or mechanisms is outside the scope of this paper.

4.14.7 Means of Verification

The main purpose of the verification of the partitioning mechanism is to demonstrate the following:

- That no errors of any software component can contribute to misbehavior or failure of any other outside the partition.
- That no violation of assignment (for example, overlapping) of the shared resources will be accepted.

Where partitions are used to isolate different software/approval levels, the partition mechanism should be verified to the objectives applicable to the highest software/approval level.

Elements of partitioning integrity can be verified by exercising the partitioning mechanisms using special test scenarios, simulations, and/or analysis techniques. Test scenarios should be written to stimulate the partitioning mechanism by injecting errors or violation attempts to bypass time and space constraints. An example of additional analysis would be the calculation of worst-case execution times to assess temporal performance.

The verification criteria for the software providing the partitioning means should comply with the ED-12C/ED-109A objectives. For level D (AL5) software, even though low-level requirements and Source Code are not required to be submitted as certification evidence, it is necessary to document all details defining the partitioning mechanism.

A verification test suite (containing normal range test cases and abnormal or out of range test cases for all requirements of the partitioning mechanism) should be established. Robustness of the partitioning mechanism may be demonstrated by use of a requirements-based test suite (satisfying requirements-based test coverage).

The output from the above process could provide one acceptable means to demonstrate partitioning integrity.

4.15 DP #15: RELATIONSHIP BETWEEN REGRESSION TESTING AND HARDWARE CHANGES

Reference: ED-12C: Sections 12.1.1, 12.1.3.e, 12.1.3.f, and 12.1.3.g

ED-109A: Sections 12.1.1, 12.1.3.e, 12.1.3.f, and 12.1.3.g

Keywords: change impact analysis; hardware; modification; previously developed software; PDS; regression analysis; regression testing; re-verification; traceability

4.15.1 Definition

Regression analysis or testing of software includes any repetition or additional execution of tests and/or analysis, which is intended to show that the software's behavior is unchanged, except as required by the change to the software or data.

4.15.2 Regression Analysis and Testing

Although hardware issues are beyond the scope of ED-12C/ED-109A, a change impact analysis should be conducted for any proposed hardware changes to determine the effect upon the system, other hardware components, and the software. This in itself is the catalyst for any hardware or software regression analysis or testing that may be needed. The level of analysis or testing required will vary depending upon the impact of the change. Hardware changes can result in a range of system and software impacts, including but not limited to the following:

- Modifications to the system requirements but not the software requirements, design, or the code.
- Modifications to the system requirements, software requirements, design, and/or code.
- Modifications to the code but not the system requirements or the software requirements.

These impacts are explained in the following examples. Scenario #1 addresses the first bulleted item and Scenario #2 addresses the second and third bullets above.

4.15.3 Discussion and Examples of Hardware Changes

These examples are based on the premise that the software development is finished and compliance with ED-12C/ED-109A has been demonstrated. Requirements management techniques were implemented on these example programs; that is, detailed requirements and verification traceability are in place, so the re-verification work is straightforward.

4.15.3.1 Scenario #1: Modification to the system requirements, but not the software requirements, design, or code.

If a system level requirement change is found to have no impact on the software requirements and subsequently have no effect on the Source Code, then the verification test cases and procedures established to verify the software requirements are still valid, as the Source Code has not changed. However, the system level verification cases and procedures established to verify the system level requirements should be reviewed for their validity due to the changes at the system level.

A regression analysis should be accomplished to identify the effects of the changes to system requirements and to the integration test cases and procedures. Compliance is found by repeating these verification activities with the modified and/or new test cases and procedures and the new hardware.

4.15.3.2 Scenario #2: Modifications to the system requirements, software requirements, software design, and code.

This scenario happens when the hardware change requires a change to software requirements, design, or code to accommodate it (for example, a serial communication device with a different baud rate). In this case, the hardware change may drive modifications to the low-level requirements, the code, and the associated Software Verification Cases and Procedures.

A regression analysis should be accomplished in accordance with ED-12C/ED-109A Section 12.1.1 for evaluating the impact of the changes to the software and to identify additional re-verification (that is, regression analysis and/or regression testing). This should show that there was no adverse impact on the unchanged software and that the changed software functions correctly. Compliance is found by performing these verification activities with the new and/or modified test cases and procedures and the new hardware.

4.15.4 Conclusion

Efficient re-verification using regression analysis and/or testing is highly dependent on implementing good requirements management and traceability. This is especially true when hardware changes affect other aspects of their related systems, particularly with respect to functional interfaces that should be controlled among aircraft, system, and software requirements. It should be recognized that modified configuration items should be controlled in compliance with section 7 of ED-12C/ED-109A. The resulting baselines should be consistent (for example, the documented design corresponds with the released code).

4.16 DP #16: CACHE MANAGEMENT

Reference: ED-12C/ED-109A: Section 6.3.4.f

Keywords: cache; memory; partitioning; Source Code; verification

4.16.1 Purpose

Many applicants utilize cache memory in microprocessors to improve performance of multiple memory accesses, thus improving processor throughput. However, using cache memory increases the complexity and may decrease the accuracy of the worst-case execution time (WCET) analysis and introduces challenges for coherency, deterministic execution, correct memory management, and partitioning protection. This discussion paper considers some of the concerns and current approaches for addressing cache memory usage in airborne systems and equipment.

4.16.2 Technical Considerations

4.16.2.1 Introduction

Cache memory is a fast memory used as temporary storage for data, instructions, and address information by the central processing unit (CPU) of microprocessors. A hardware memory management unit (MMU) is typically used to optimize the contents of the cache memory for maximum throughput.

4.16.2.2 Vulnerabilities

There are several vulnerabilities regarding the usage of cache memory. In many airborne systems containing multiple software functions of different software levels, cache memory is a shared resource used by all the software functions executing on that microprocessor. Therefore, providing protection mechanisms and/or partitioning between those functions can be challenging and necessitates a thorough robust partitioning analysis and testing of the approach and implementation of cache memory management.

Some of the specific vulnerabilities are described below (not an exhaustive list):

- Difficult to establish accurate worst-case execution timing of all applications in order to ensure a predictable timing evaluation (for example, no deadlines are missed). It may become more difficult when the execution order is disrupted (for example, when asynchronous interrupts are used).
- Difficult to establish an accurate cache model (when used due to complexity or when proprietary data is unavailable) in order to ensure a predictable timing evaluation.
- Difficult to establish assurance of robust partitioning (including jitter) for real-time operating system using cache memory in accordance with caching policies (for example, to ensure data integrity and availability).
- Difficult to establish coherency of data exchange between cache and RAM in order to ensure data integrity and availability.
- Difficult to establish cache failure scenarios in order to ensure data integrity and availability.
- Difficult to establish protection against single event upset (at cache level) as cache memory may not have error correction codes (ECC).
- There may be several levels of cache memory with different caching policies for data, instruction, pages table addresses, etc.

4.16.2.3 Recommendations for dealing with cache

Various approaches are used by developers regarding the use of cache memory, including the following:

- No-cache approach.
- Use of development tools (at the processor level) to provide accuracy, reliability, integrity, and determinism.
- System and software architectural features independent from cache memory effects.
- Addressing cache coherency to solve any data exchange issues.
- Addressing application jitter due to cache effects, to ensure correct implementation of timing requirements when partitioning is used.
- Cache flushing or invalidation to ensure determinism in partition switching or task execution.
- Cautious selection of caching policies.

4.17 DP #17: USAGE OF FLOATING-POINT ARITHMETIC

Reference: ED-12C/ED-109A: Sections 6.3.4.f

Keywords: Source Code; verification

4.17.1 Purpose

Software developers use floating-point arithmetic and numbers as a possible representation of digital data used during computations. The floating-point arithmetic number representation should guarantee computational accuracy appropriate to the data usage, particularly when using specific algorithms.

4.17.2 Technical Considerations

When using floating-point arithmetic and number representation, specific considerations to ensure correct computations within the embedded software should include the following:

a. The standard used to represent floating-point numbers (for example, IEEE754, IEEE854, etc.).

b. The format used (for example, simple accuracy, double accuracy, etc.).

c. Specific design and coding constraints applied to the use of the floating-point arithmetic and numbers to ensure correct computation particularly:

   1. Cancellation issue: Subtraction between two members of nearly equal value.
   2. Absorption issue: Addition between two members with different scales.
   3. Inconsistencies linked to “not a number” and “infinite”.
   4. Floating-point exceptions (for example, invalid operation, division by zero, overflow, inexact result, denormalized number, underflow, etc.).
   5. Consistency with mathematic/arithmetic rules.
   6. Management of accuracy for multiple cascading computations (for example, rounding mode).
   7. Specific verification means linked to floating-point arithmetic use and pass/fail criteria establishment.

NOTE: *It is recognized that some of the above considerations are also applicable to fixed-point arithmetic.*

4.18 DP #18: SERVICE EXPERIENCE RATIONALE FOR ED-109A

Reference: ED-109A: Section 12.3.4

Keywords: service experience

This paper clarifies guidance from ED-109A on considerations in using service experience as an alternative method of compliance for embedded software. Commentary is provided for each topic in ED-109A section 12.3.4. The ED-109A headings are italicized and underlined; the commentary is not. The full text of ED-109A section 12.3.4 is not repeated – only the section titles and numbers. Therefore, the DP should be read in conjunction with ED-109A section 12.3.4.

NOTE: *For clarification of service history relevant to ED-12C, see DP #4 (section 4.4).*

12.3.4 Service Experience

Service experience is under ED-109A section 12.3, "Alternative Methods." Section 12.3.a states: "*An alternative method should be shown to satisfy the objectives of this document or the applicable supplement*." This is the definition of "equivalent safety" for this discussion paper. "Approval credit" in this instance refers to using service experience towards satisfying the intent of the objectives in ED-109A Annex A.

The proponent for the software product is responsible for the collection of data justifying the requested approval credit based on the software's service experience.

Necessary conditions for this justification are:

- The service period duration is sufficient.
- The new operational environment is the same or similar (with additional verification needed if not the same).
- The product is stable and mature (that is, few Problem Reports and/or modifications occurred during the service period).

12.3.4.1 Relevance of Service Experience

a. *Type of experience.* Service experience data may be available through the typical practice of running new systems in parallel with operational systems in the operational environment (also known as shadow mode operations), long duration of simulation of new systems, and multiple shadow operations executing in parallel at many locations. Relevant service experience data may be available from reuse of software from in-service systems, or system verification and pre-operational activities, such as training. For software with no precedence or CNS/ATM applications, many processes may be used to collect service experience; examples include the system integration process, the validation process, the operator training process, the system qualification testing, the system operational evaluation, and field demonstrations.

Service experience is different from standard testing for approval because valid service experience requires evidence about the makeup of test cases (for example, operational workload) that other types of testing often do not contain. Testing may be used as service experience only if the criteria in this section are met.

b. *Known configuration.* Configuration management allows effective assessment of the product's service experience in the presence of changes, such as additions to functionality or correction of errors.

c. *Operating time collection process.* For collection of operating time or event data, the following needs to be considered:

- Method by which operating time or number of events was measured.
- Reliability of the means of measuring operating time or number of events.
- Method by which operating time or number of events was reported.
- Reliability of the means of reporting operating time or number of events.
- Impact of portions of software unused during normal operations on service experience credit. For example, credit may not be granted for software components that were not active or used during the service period.

Service experience should be collected for all operating modes or states (for example, a changing number of controller positions for an air traffic management center).

d. *Changes to the software.* The applicant should have configuration records of all changes made to configuration items. The configuration history should record discrete changes made to add functionality or correct errors. It should provide the evidence to substantiate the integrity of the processes applied to make such changes. Records should exist to identify all changes made to the Executable Object Code or Adaptation Data Items, regardless of the source of the change (for example, changes introduced through the use of different compiler options). The following should be considered when assessing the adequacy of configuration management and control of the integrated software system (all components for which approval credit is sought using service experience):

- Completeness and reliability of the evidence that all components of the software have been change controlled throughout the service period.
- Number and significance of the modifications.
- Method by which the validity of modified software for service experience credit is established.

Software or hardware changes could affect the applicability of service experience collected prior to the change.

e. *Usage and Environment.*

(1), (2) The applicant should provide an analysis which shows that the software will be performing the same function in the proposed new application as it performed during the service experience period. For example, in real-time control and related software, basic performance and accuracy requirements, as well as more detailed issues such as input ranges, output ranges, and data rates, should be examined. In operating systems and protocol stacks, relevant execution characteristics, such as event sequences which might drive the software operation should also be considered. This analysis may be contained in the Plan for Software Aspects of Approval or other document.

No credit for service experience can be claimed for software functions that were not exercised in the previous application. These software functions may need to be evaluated in the proposed new application to ensure they do not present a hazard, even if they are still not intended to be exercised.

(3) The service experience environment should be assessed to determine whether problems may have been avoided due to technology-related usage differences (for example, manual versus automatic operations), system level recovery mechanisms, or the level of operator training. If so, the intended environment needs to be assessed to determine whether these same mechanisms will exist.

(4) Software service experience will often be part of the service experience of a larger system. Therefore operating environment is an important consideration in determining relevance. Considerations in assessing the relevance of the operating environment include:

- Adequacy of data available to support analysis of similarity of operating environment. If environment changes over time, data may need to support differentiation between relevant and irrelevant portions of the service experience duration.
- Similarity of the hardware environment of service experience to the target hardware environment. Resource differences (for example, time, memory, precision, communication services) between the two environments should be analyzed. Changes needed for the software to be compatible with the new environment may invalidate the service experience or may preclude service experience credit for hardware compatibility objectives.

If there were fewer hazards or less severe hazards assigned to the software under consideration in the previous application, the assurance level assigned to it during the system assessment process would likely be higher than that of the previous application. Additional work may be required to assure compliance with the new safety objectives if it cannot be shown that the service usage regularly exercised the aspects of the software which support the management of the newly allocated hazards. For any application of service experience, the applicant should identify whether there are different requirements of the new installation or environment that make any demands on the system that may affect the safety objectives. The applicant should assure that the implementation of these requirements have been exercised in service or are subjected to verification to the appropriate level as defined by ED-109A. See ED-109A sections 12.1.1 through 12.1.3 for additional guidance.

(5) The relationship between the service experience environment and the intended environment includes, but is not limited to, processor and memory utilization, accuracy, precision, communication services, built-in-tests, fault tolerance, channels and ports, queuing modes, priorities, and error recovery actions. If there were changes to hardware during the service experience duration, analysis should be conducted to assess whether it is still appropriate to consider the service experience duration before the modifications.

f. *Deactivated code*. Functions that did not cause a problem may not have left any indications regarding their usage; absence of problems may not mean that the function was free from errors. Additional verification may be required to support compliance with the safety objectives for the subset components, if they are found to be used differently than in the prior service due to the fact that the software package is not reused as a whole.

If the analysis of the software's use in its previous environment identifies code that is rarely executed, but which may be executed in the new environment, then verification evidence is required for the code and the collected service experience may not be applicable. Even if the unused functionality is unlikely to be invoked in the new environment, supporting evidence is needed to show that differences in operational usage will not cause an invocation of the unused functionality.

g. *Recovery from failures.* The number of software anomalies or Problem Reports may be related to the extent of fault tolerance, error trapping, or built-in-testing in the computing environment in which the service experience was collected. On the other hand, the fault tolerance properties of the computing environment may make the service experience data look better than it should. The quality of data assumed in any service experience argument will therefore be affected. The recovery mechanisms used in the operational environment should be shown to be similar or superior to those used in the service experience environment. It also needs to be shown that those recovery mechanisms will continue to work reliably in the intended environment (possibly requiring separate testing of the recovery mechanisms themselves).

12.3.4.2 Sufficiency of Accumulated Service Experience

a. The amount of service experience necessary for approval credit is usually derived from safety objectives. Safety objectives are usually related to failure severity or hazard classification and are identified during the system safety assessment. The system safety assessment process will thus need to provide the safety objectives as inputs to the determination of sufficient service experience.

In some cases, operating time is not as relevant to service experience as using the number of events such as alarms/warnings or the number of times an operator queried the software for specific information.

b. The amount of service experience needed will also depend on the required confidence the safety objective has been achieved (for example, 90% or 95% confidence that the safety objective has been met). More service experience is needed for higher confidence. Methods for calculating the amount of service experience needed as a function of confidence level should be based on a documented and justified method accepted by the approval authority. Such methods can be found in international industrial and military reliability standards.

c. Differences between the service experience environment and the system operational environment may necessitate additional service experience, and may mean that techniques other than service experience need to be used.

d. Some objectives require more service experience than others to attain the confidence required. The rationale for why the amount of service experience proposed is sufficient to address the objectives proposed needs to be documented.

e. Coupling service experience with other forms of evidence may reduce the amount of service experience needed in order to meet some objectives. The extent of the reduction depends on the type and completeness of the other means of approval being used.

12.3.4.3 Collection, Reporting, and Analysis of Problems found during Service Experience

a. *Problem reporting process.* Service experience application requires collection of problem data in order to substantiate assertions regarding the integrity of the software. The intent of this item is that users are able to report anomalous behavior and that any such reports are recorded. The system should contain information necessary to support assessment of problems as described in paragraphs b and c, below. Issues to consider in assessing the problem reporting process include:

- Method and reliability of detecting in-service problems.
- Method and reliability of recording in-service problems.
- Method for determining problem severity.
- Method for differentiating software-related problems from other problems.

- Method for assessing relevance of and (if appropriate) collecting non-service problems (for example, problems detected during rehearsals or preliminary testing).
- Method for assessing impact of product improvements or unresolved software Problem Reports (if any) on service experience credit.

Common-cause problems will occur together in time and may have a greater impact than problems with separate causes. Furthermore, common-cause problems may defeat system redundancy strategies and so could have safety implications. Cascading problems will generally occur closer together in time than other problems, so they may have a greater impact than the same number of non-cascading problems. Common-cause and cascading problems violate the assumptions usually made when measuring software error and repair rates, and so should be treated with great caution if quantitative analysis techniques are used at the system level to determine the amount of service experience needed.

b. *Process-related problems.* System design errors, hardware design errors, and hardware failures are outside the scope of ED-109A. While they are still a cause for concern and may invalidate the product service experience application, they do not indicate an inadequate software process.

The presence of multiple errors, particularly systemic errors that should have been detected during software development, may indicate an inadequate software development process. These errors should be examined to assess whether they may be attributed to a weak development process. This assessment should consider trends and characteristics of the errors such as:

- Large number of defects: A large number of defects could indicate an inadequate development process. The size of the product should be considered to determine whether the density of defects is indicative of a process issue.
- Failures caused by off-nominal inputs: Failures of this nature could indicate a lack of robustness in the testing process or an inadequate requirements process.
- Partially fixed defects: This could indicate poor requirements, design, or regression testing processes.
- Reintroduction of errors: This could indicate a poor configuration management process.
- Failures caused by borderline values: Failures of this nature could be caused by improper algorithm or interface design – particularly with respect to numeric processing.

In the case of inadequacies which are clearly traceable to an identifiable process deficiency or omission, additional analysis, inspection, or other verification should be performed. Service experience may continue to be used, provided the identified process deficiency or omission and its effects are addressed. In this case, the software should also be analyzed to find any other latent errors introduced by the inadequate process.

c. *Safety-related problems.* The existence of safety-related problems may be indicative of an inadequate development process (that is, a development process that does not conform to all the objectives of ED-109A). If an inadequate software process is indicated, three actions need to be taken:

1. Identify the gaps in the development process.
2. Analyze the product to determine if other safety related errors exist.
3. Provide assurance for the missing or incomplete elements of the development process, generally via additional verification activity to meet ED-109A objectives.

Credit for service experience may not be possible if safety-related problems indicate an inadequate development process, and the process of reverifying the software and correcting defects could introduce new defects or change the results of any quantitative analysis.

Service experience may be used even if a safety-related problem has been found if evidence can be presented showing that the problem has been resolved, and it was not the result of a systemic cause (for example, there was a problem with the test environment that would not be seen in operation).

12.3.4.4 Information to be Included in the Plan for Software Aspects of Approval

a-b. Service experience is unlikely to be applicable to AL1, because of the impact of any differences in operational environment on assurance and because of the amount of service experience that would be needed.

c. The rationale for calculating number of hours in service should require collection and analysis of any discrepancies in:

- total operating time and workload by operating mode (if there are multiple modes);
- workload by function for each operating mode; and
- proportion of failures in software functionality not involved in proposed use.

d-f. Service experience from systems with long service lives may include system failures. These system failures do not necessarily render the software invalid for its intended application, particularly at lower assurance levels. At higher assurance levels, any system failures may invalidate the use of the service experience, particularly if those failures were safety-related. There should not have been any failures observed in the service experience data collected for AL1 or AL2. These details should be agreed in advance based on the software's contribution to system safety.

In cases where the plan called out a number of system failures that on later analysis was exceeded, this section allows for re-negotiation of the applicability of the service experience to the intended assurance level.

4.19 DP #19: INDEPENDENCE IN ED-12C/ED-109A

Reference: ED-12C/ED-109A: Sections 4.6, 6.2.e, 8.2, 11.3, 11.5, and 12.3.2.1; and Annex B

Keyword: independence; verification

Some of the verification objectives in ED-12C/ED-109A should be accomplished with independence for some software/assurance levels. Different means (for example, personnel or tools) should be used to accomplish the verification than the means used to accomplish the development or verification whose outputs are to be verified. In some cases, this may be achieved by independent personnel reviewing the outputs of the verification activity. The purpose of independence is to avoid having a misinterpretation of requirements by a single means carry through both the design and the verification of a function. A different reporting organizational structure or company is not needed.

ED-12C/ED-109A Annex A defines which objectives should be satisfied with independence, by software/assurance level.

Independence is defined in the ED-12C/ED-109A Annex B glossary as: *"Separation of responsibilities which ensures the accomplishment of objective evaluation. (1) For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve an equivalence to the human verification activity. (2) For the software quality assurance process, independence also includes the authority to ensure corrective action."*

ED-12C/ED-109A section 6.2.e states: *"Verification independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified. A tool may be used to achieve equivalence to the human verification activity. For independence, the person who created a set of low-level requirements-based test cases should not be the same person who developed the associated Source Code from those low-level requirements."*

The term independence is also used in other contexts in ED-12C/ED-109A. ED-12C/ED-109A section 8.2.a indicates that software quality assurance (SQA) functions should be performed by personnel with the "*authority, responsibility, and independence to ensure that the SQA process objectives are satisfied*." This allows quality assurance considerations, including corrective action assurance, to be independent of development considerations.

Independence is also used in the context of independent development of multiple-version dissimilar software. ED-12C/ED-109A section 12.3.2.1(a) states: *"The applicant should demonstrate that different teams with limited interaction developed each software version's software requirements, software design and Source Code."* This minimizes the possibility of a common design error in the multiple software versions.

Some of the verification objectives that should be satisfied with independence for Level A (AL 1) software are shown in Figure 4-2 and Table 4-5 below. Note that Table 4-5 only describes the situation when the verification activity is performed by a person(s). In each case, the verification activity could also be performed by a tool. Either approach would satisfy the objective with independence.

NOTE 1: *ED-12C/ED-109A Table A-7 objective 9 is not discussed in this DP because "additional code that cannot be traced to Source Code" is usually generated by a tool such as a compiler; therefore, any manual verification of that additional code would be independent of the development of that additional code.*

NOTE 2: *ED-12C/ED-109A Table A-5 objectives 8 and 9 pertain when parameter data items (PDI) are used. For this reason these objectives are not addressed in Figure 4-2. However, independence for objective 8 should be achieved by having a person other than the person who was involved in the production of the PDI File from its HLRs (that is, development of any intermediate representation or creation of the PDI File) to perform the verification of it. Independence for objective 9 should be achieved by having a person other than the person who verified the PDI File to ensure that the verification was achieved.*

NOTE 3: *Per ED-12C/ED-109A Table A-6 (objectives 3 and 4) and section 6.2.e the developer of low-level requirements-based tests should not also be the developer of Source Code. However, the developer of high-level requirements-based tests may also be the developer of the Software Design or Source Code. ED-12C/ED-109A (Table A-6 objectives 1 and 2) does not specify that the development of high-level requirements-based tests be done with independence because the development of high-level requirements is typically a more collaborative effort than is the development of low-level requirements.*

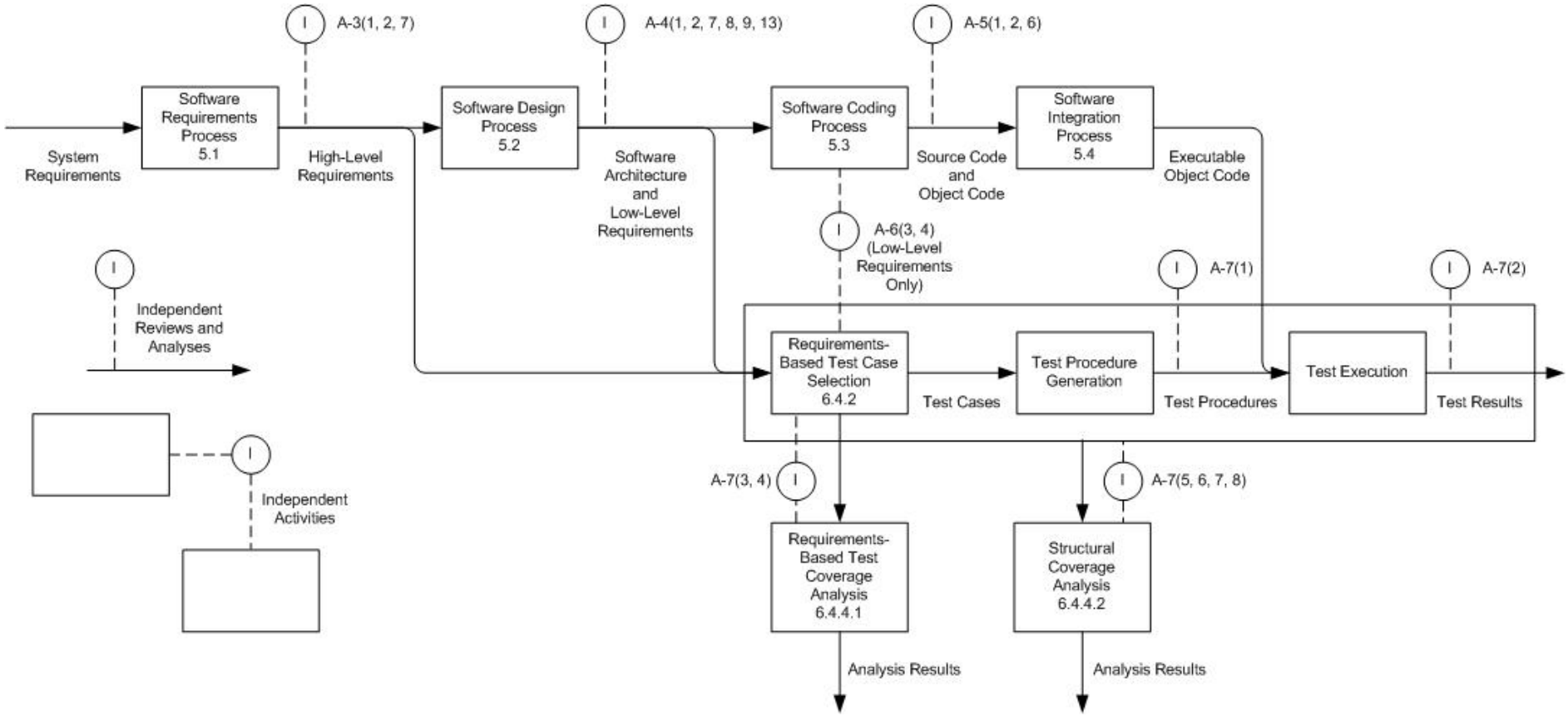


FIGURE 4-2: INDEPENDENCE AND VERIFICATION OBJECTIVES ILLUSTRATED

TABLE 4-5: INDEPENDENCE INTERPRETATION FOR VERIFICATION OBJECTIVES

	Objective	Verification Activity	Item Being Verified	Interpretation
A-3(1)	High-level requirements comply with system requirements.	Reviews and Analyses of the High-Level Requirements	High-level requirements	The reviews and analyses of the high-level requirements should be performed by a person(s) other than the developer of the high-level requirements.
A-3(2)	High-level requirements are accurate and consistent.			
A-3(7)	Algorithms are accurate.			
A-4(1)	Low-level requirements comply with high-level requirements.	Reviews and Analyses of the Low-Level Requirements	Low-level requirements	The reviews and analyses of the low-level requirements should be performed by a person(s) other than the developer of the low-level requirements.
A-4(2)	Low-level requirements are accurate and consistent.			
A-4(7)	Algorithms are accurate.			
A-4(8)	Software architecture is compatible with high-level requirements.	Reviews and Analyses of the Software Architecture	Software architecture	The reviews and analyses of the software architecture should be performed by a person(s) other than the developer of the software architecture.
A-4(9)	Software architecture is consistent.			
A-4(13)	Software partitioning integrity is confirmed.			
A-5(1)	Source Code complies with low-level requirements.	Reviews and Analyses of the Source Code	Source Code	The reviews and analyses of the Source Code should be performed by a person(s) other than the developer of the Source Code.
A-5(2)	Source Code complies with software architecture.			
A-5(6)	Source Code is accurate and consistent.			

	Objective	Verification Activity	Item Being Verified	Interpretation
A-6(3)	Executable Object Code complies with low-level requirements.	Requirements-Based Testing	Executable Object Code	The person(s) who created a set of low-level requirements-based test cases should not be the same person(s) who developed the associated Source Code from those low-level requirements. It follows that: 1. The same person(s) could develop the low-level requirements and the Source Code, provided another person(s) develops the test cases from those low-level requirements, or 2. The same person(s) could develop the low-level requirements and their associated test cases, provided another person(s) develops the Source Code.
A-6(4)	Executable Object Code is robust with low-level requirements.			
A-7(1)	Test procedures are correct.	Reviews and Analyses of the Test Procedures	Test procedures	The reviews and analyses of the test procedures should be performed by a person(s) other than the developer of the test procedures.
A-7(2)	Test results are correct and discrepancies explained.	Reviews and Analyses of the Test Results	Test results	The reviews and analyses of the test results should be performed by a person(s) other than the person(s) who performed the tests.
A-7(3)	Test coverage of high-level requirements is achieved.	Requirements-Based Test Coverage Analysis	Test cases	The requirements-based test coverage analysis should be performed by a person(s) other than the developer of the test cases.
A-7(4)	Test coverage of low-level requirements is achieved.			

	Objective	Verification Activity	Item Being Verified	Interpretation
A-7(5)	Test coverage of software structure (modified condition/decision coverage) is achieved.	Structural Coverage Analysis	Test cases, test procedures, and/or test results	The exact independence required depends on how the structural coverage analysis is carried out. When the structural coverage analysis is performed on the test cases, then the structural coverage analysis should be performed by a person(s) other than the developer of the test cases. Similarly, if the structural coverage analysis is performed on the test procedures and test results, then the structural coverage analysis should be performed by a person(s) other than the developer of the test procedures and test results.
A-7(6)	Test coverage of software structure (decision coverage) is achieved.			
A-7(7)	Test coverage of software structure (statement coverage) is achieved.			
A-7(8)	Test coverage of software structure (data coupling and control coupling) is achieved.			

4.20 DP #20: PARAMETER DATA ITEMS AND ADAPTATION DATA ITEMS

Reference: ED-12C: Section 2.3.4, 2.5.1, 2.5.5, 4.2.j, 5.1.2.i, 6.3.1.b, 6.6, 7.4.b, 11.16.j

ED-109A: Section 2.3.4, 2.6.1, 2.6.5, 4.2.j, 5.1.2.i, 6.3.1.b, 6.6, 7.4.b, 11.16.j

Keywords: adaptation data item; compatibility; field-loadable software; parameter data item; PDI File; software load control

4.20.1 Introduction

This discussion paper answers several questions related to parameter data items and adaptation data items. In this discussion paper only the term “parameter data item” (PDI) is used; however, this should be interpreted as meaning adaptation data item as well. Questions answered by this discussion paper include:

- What is a parameter data item? (see 4.20.2)
- What considerations apply with respect to compatibility between Executable Object Code and Parameter Data Item Files? (see 4.20.3)
- What does the fourth bullet of section ED-12C/ED-109A section 6.6 mean? (see 4.20.4)
- How is a separately verifiable Parameter Data Item File verified? (see 4.20.5)
- How is PDI related to option-selectable software, user-modifiable software, and field-loadable software? (see 4.20.6)
- What is the purpose of the normal range testing and the “robustness of the Executable Object Code (EOC) defined in ED-12C/ED-109A section 6.6? (see 4.20.7)
- What is the rationale for assigning the PDI the same software/assurance level as the component using it and not using the assignment rules in ED-12C/ED-109A section 2.3.4? (see 4.20.8)

4.20.2 What is a parameter data item (PDI)?

A PDI is a software component that contains only data (and no EOC). A PDI is also a configuration item (CI). Software may consist of one or more CIs of EOC and one or more PDI CIs.

The representation of the PDI that is directly usable by the processing unit of the target computer is known as the PDI File. A PDI File is an instantiation of the PDI containing defined values for each data element.

The PDI File may be used by the EOC to govern software functionality. This is illustrated in Figure 4-3.

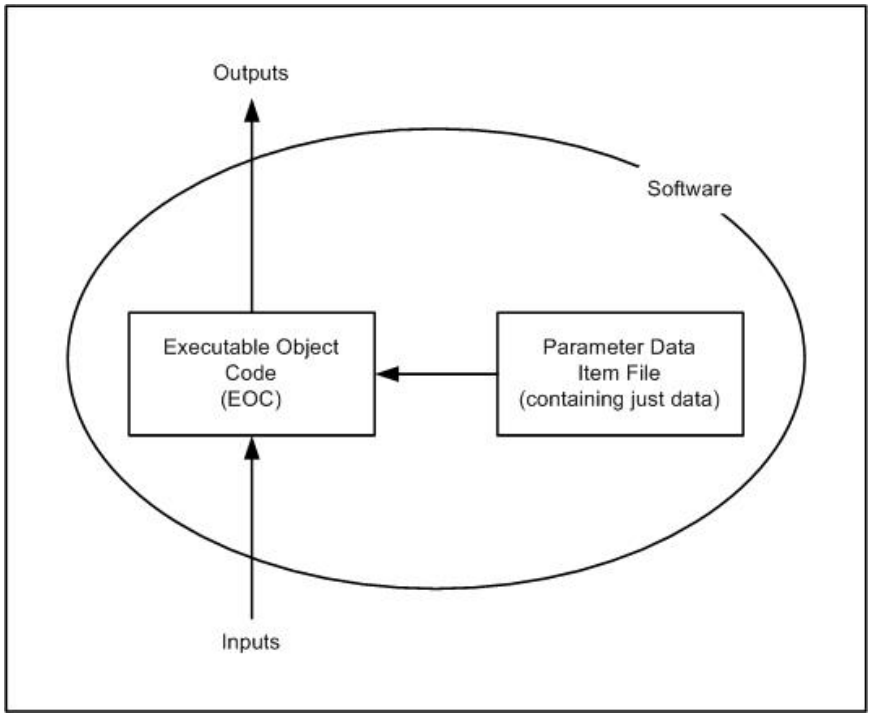


FIGURE 4-3: PARAMETER DATA ITEM RELATIONSHIP WITH EXECUTABLE OBJECT CODE

When ED-12C and ED-109A were defined, the following view was taken with respect to PDI:

- The high-level requirements should define how the PDI affects the function of the software.
- The PDI structure, attributes, and values should be defined in requirements to allow requirements-based verification of the PDI File. Since high-level requirements are required for all software/assurance levels, the decision was made to define structure, attributes, and values in the high-level requirements.
- The actual values in the PDI are considered high-level requirements that are allocated to the PDI only. These may trace directly to system level requirements or may be considered derived high-level requirements. This is illustrated in Figure 4-4.

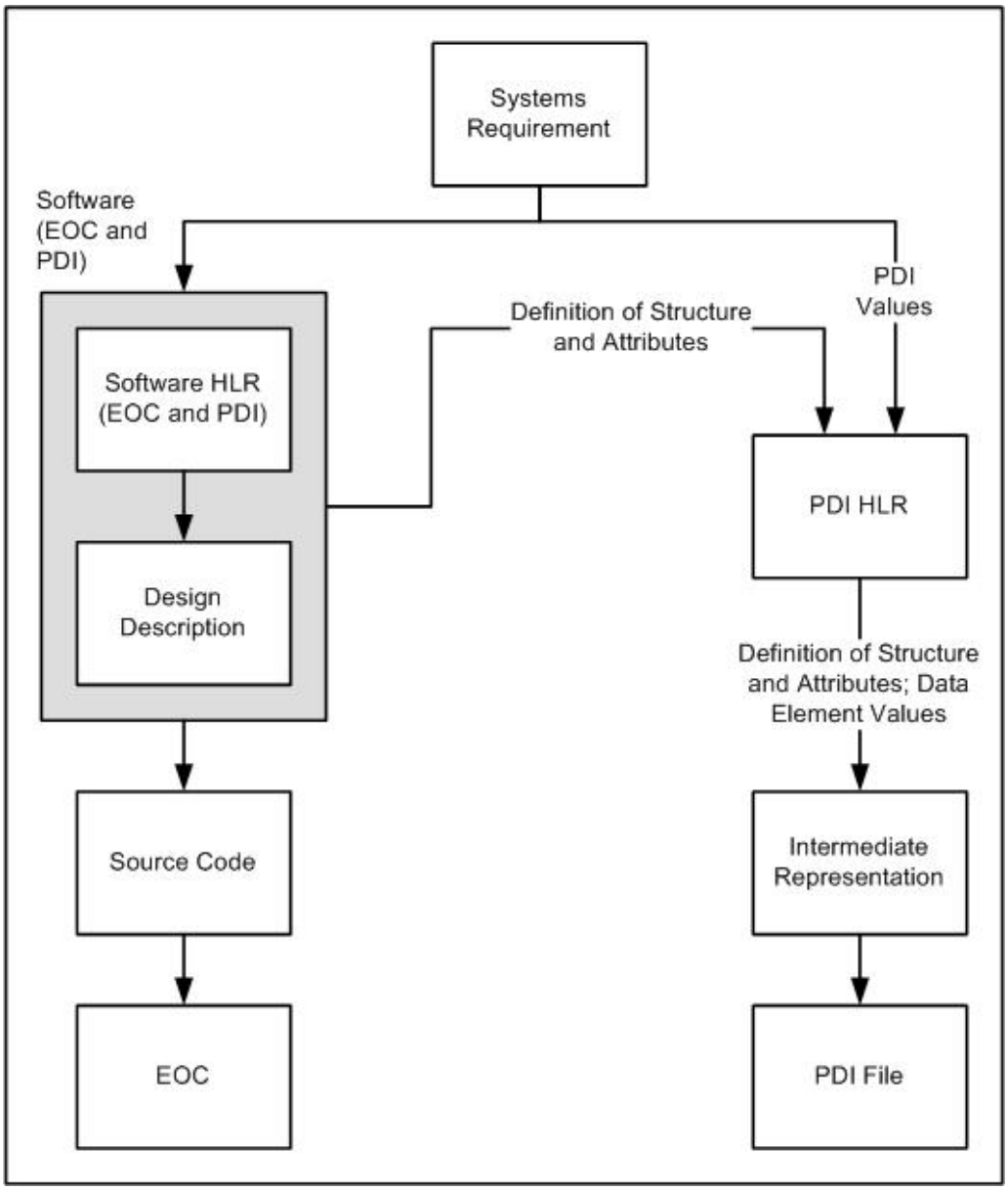


FIGURE 4-4: PDI VIEW IN ED-12C AND ED-109A

The “intermediate representation” referred to in Figure 4-4 refers to an implementation (for example, an XML file) of the PDI data element values that is used to generate the PDI File. The PDI File may also be generated directly from the PDI HLRs without such an intermediate implementation being produced.

Figure 4-5 shows a very simple example of life cycle data for software consisting of an EOC configuration item (Sensor A) and a PDI configuration item (B1).

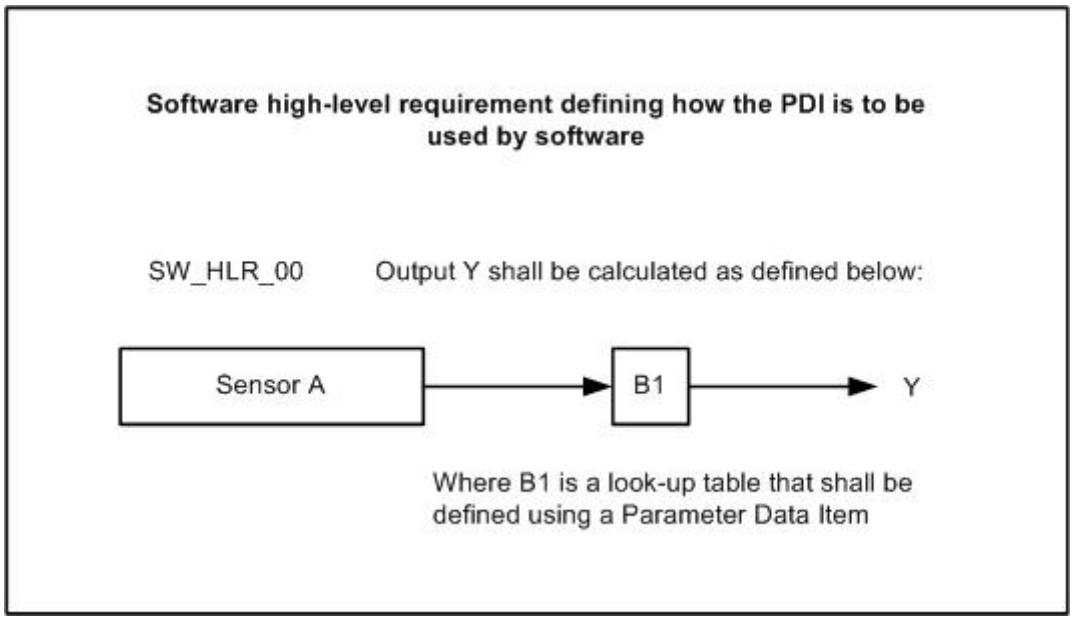


FIGURE 4-5: EXAMPLE OF PDI HIGH-LEVEL REQUIREMENT

Software high-level requirement defining how the PDI is to be used by software

SW_HLR_01

Before the PDI is used its CRC shall be verified. If the CRC is not correct the following default values shall be used :

Output value for Sensor A signal < -10 : 12.0
Output value for -10 ≤ Sensor A signal < 10 : 10.0
Output value for Sensor A signal ≥ 10 : 11.0

SW_HLR_02

If the value looked up in the PDI falls outside the range 0.0 – 12.0, the default values of SW_HLR_01 shall be used.

SW_HLR_03

Y shall be set to the value looked up in the PDI table based on the signal from Sensor A (with the exception of the cases defined by SW_HLR_01 and SW_HLR_02).

Software (EOC+PDI) high-level requirements defining structure and attributes of the PDI

SW_HLR_04

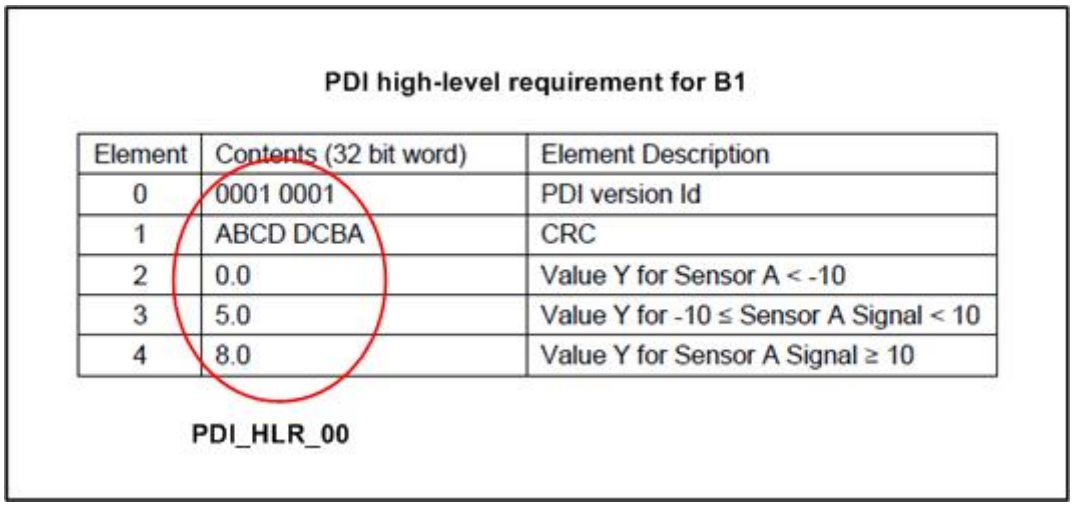
The PDI shall have the following

Word	Contents (32 bit word)
0	PDI version Id
1	CRC
2	Value Y for Sensor A < -10
3	Value Y for -10 ≤ Sensor A Signal < 10
4	Value Y for Sensor A Signal ≥ 10

SW_HLR_05

Values in words 2 through 4 shall be represented as floating points. Their permissible range is 0.0 – 12.0.

FIGURE 4-6: EXAMPLE OF PDI HIGH-LEVEL REQUIREMENTS



PDI high-level requirement for B1

Element	Contents (32 bit word)	Element Description
0	0001 0001	PDI version Id
1	ABCD DCBA	CRC
2	0.0	Value Y for Sensor A < -10
3	5.0	Value Y for -10 ≤ Sensor A Signal < 10
4	8.0	Value Y for Sensor A Signal ≥ 10

PDI_HLR_00

FIGURE 4-7: EXAMPLE OF HIGH-LEVEL REQUIREMENT DEFINING PDI VALUES

In the example above (in Figures 4-6 and 4-7) the structure of the PDI is comprised of five elements, ordered by element numbers, consisting of 32-bit words, and with assignments as shown in the element description column in Figure 4-7.

The attributes (type and range) and values for the five elements are shown in Table 4-6 (based on Figures 4-6 and 4-7):

TABLE 4-6: DATA TYPE ATTRIBUTES

Word	Type	Range	Value
0	unsigned integer (32 bits)	0_{hex} – FFFF $FFFF_{hex}$	0001 0001_{hex}
1	unsigned integer (32 bits)	0_{hex} – FFFF $FFFF_{hex}$	ABCD $DCBA_{hex}$
2	IEEE754 floating point (32 bits)	0.0 - 12.0	0.0
3	IEEE754 floating point (32 bits)	0.0 - 12.0	5.0
4	IEEE754 floating point (32 bits)	0.0 - 12.0	8.0

4.20.3 What considerations apply with respect to compatibility between Executable Object Code and Parameter Data Item Files?

In ED-12C, considerations with respect to compatibility are addressed by sections 2.5.1, 2.5.5, and 7.4. In ED-109A, see sections 2.6.5 and 7.4. Compatibility is a concern for both field-loadable and non-field-loadable software. For this reason, the clarification provided by this DP applies to both types of software.

Both the PDI File and the EOC item may evolve over time and be issued in multiple versions, which may be combined in a number of ways: but only some of those combinations may be compatible (compatibility implies that both the PDI file and the EOC have been found compliant in the context of the other). To ensure compatibility all of the following need to be established:

1. Compatibility of combinations of EOC and PDI File(s) is verified.
2. Compatible configurations of EOC and PDI File(s) are clearly identified.
3. Only compatible configurations are used in the target system during operation.

During the development of the EOC, the compliance between the EOC and a given type of PDI File is demonstrated by the verification as defined in ED-12C/ED-109A section 6.6.

When a PDI File is modified, the verification is to be performed as discussed in section 4.20.5 of this DP, to demonstrate either that the compliance is maintained with the EOC, or a new compliance is established with a new EOC. Therefore, configuration control is essential to define which combinations of existing versions of PDI Files and existing versions of EOC are compatible. The life cycle data should contain information defining the combinations of compatible versions of EOC and PDI Files that may be used.

Item 3 above needs to be achieved as part of the initial development but also when new versions of the PDI File or of the software component that uses it are introduced.

The process used to develop, verify, and load new versions of a PDI File needs to be established in the software plans.

ED-12C section 7.4.b states that "*records should be kept that confirm software compatibility with the airborne system or equipment hardware*." (ED-109A section 7.4.b has same statement except it's for CNS/ATM system or equipment hardware.) The "airborne system" or "CNS/ATM system" is considered to include several software configuration items (including PDI Files). Therefore, this implies that activities should be performed to ensure compatibility between those different software components. In performing these activities, the guidance for field-loadable software (section 2.5.5 in ED-12C and section 2.6.5 in ED-109A) may be helpful even for non-field-loadable software.

In order to further ensure compatibility, the methods used to load PDI Files may be complemented by mechanisms in the EOC to detect corrupted data in compatible loads.

4.20.4 What does the fourth bullet of section ED-12C/ED-109A section 6.6 mean?

In section 6.6 of ED-12C, there are four criteria identified that need to be fulfilled in order to be able to verify the PDI separately from the EOC. The fourth criterion states that the "*structure of the life cycle data allows the parameter data item to be managed separately.*"

This criterion was added to highlight the need to organize the life cycle data to support separate verification of the PDI File. For instance, this may mean that a separate SAS, SCI, and SECI are issued for the PDI. Alternatively, a common set of documents (covering all software components) could be used but structured in such a way that the PDI information is clearly separated. A separate SCI may be necessary especially if the PDI is user-modifiable as defined by section 11.16.j.

NOTE: *ED-109A section 6.6 includes the same guidance for adaptation data items.*

4.20.5 How is a separately verifiable Parameter Data Item File verified?

The structure of the PDI and the attributes of each element are defined as high-level requirements applicable to both the PDI and the software component using it. The values of the PDI's elements are defined as high-level requirements applicable to the PDI only. Change to structure, attributes, or values of a PDI are changes to high-level requirements and all applicable activities of ED-12C/ED-109A apply. In particular:

- The high-level requirements should be shown to not conflict as defined by ED-12C/ED-109A section 6.3.1.b. Compliance with this objective implies that the values of the data elements should be consistent with each other.
- When the structure or an attribute of the PDI is changed, the software component and the PDI File have to be re-verified.
- When the values of PDI's elements are changed, and the criteria of separately verifiable PDI File are fulfilled as defined by ED-12C/ED-109A section 6.6, only the PDI File has to be re-verified.

Separate re-verification of a PDI File can be done by performing the following activities:

1. Establishing that neither the structure of the PDI File, nor the attributes of its elements have been modified.
2. Establishing that the new values comply with the defined attributes.
3. Establishing that the values themselves are correct.
4. Establishing that all the elements, and only those elements, intended to be modified have been modified.

Figure 4-8 depicts the above verification activities.

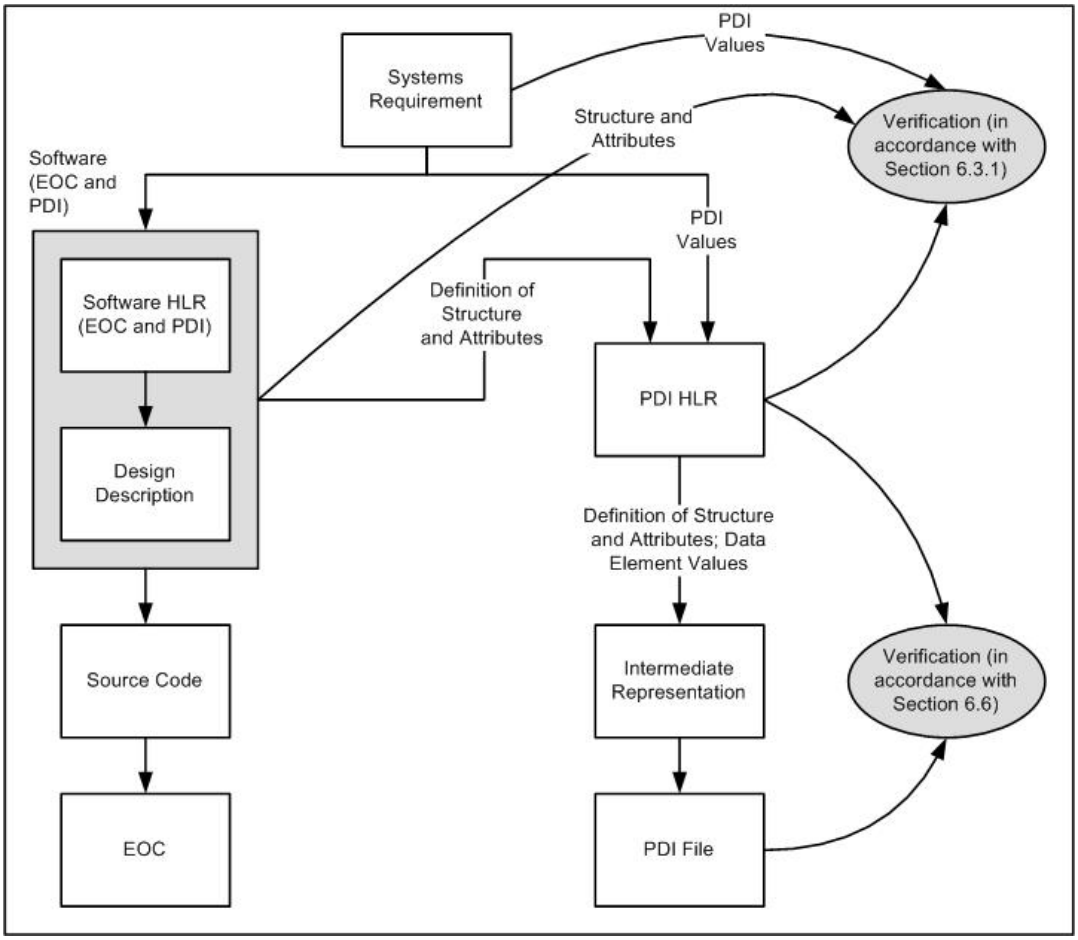


FIGURE 4-8: VERIFICATION OF PDI FILE

The PDI File shown in Figure 4-8 represents the actual binary that is loaded into the target system; as stated in ED-12C/ED-109A sections 6.6.a and 6.6.b, this PDI File should be verified. An applicant may also perform some additional verification on the intermediate representation (for example, to verify that the structure, attributes, and values contained in the intermediate representation are correct before the PDI File is generated).

4.20.6 How is PDI related to option-selectable software, user-modifiable software, and field-loadable software?

ED-12C section 2.5.1 (section 2.6.1 in ED-109A) states that depending on how the PDI is used in the system, the topics of optional-selectable, user-modifiable, and field-loadable software should be considered. This statement is included in ED-12C and ED-109A because these topics may be closely related to the use of PDI as discussed below:

- Optional-selectable software: A PDI may be used to govern the activation and deactivation of functions in the Executable Object Code. In such cases, consideration should be given to the guidance for optional-selectable software when developing the Executable Object Code.
- User-modifiable software: A user may be allowed to create and verify their own PDI Files and load them into the system. In such cases the development organization should provide "*Procedures, methods, and tools for making modifications to the user-modifiable software*" (see ED-12C/ED-109A 11.16.j).
- Field-loadable software: As was discussed above, a PDI may be loaded into the system without first removing the system software from its installation.

4.20.7 What is the purpose of the normal range testing and the robustness of the EOC defined in ED-12C/ED-109A section 6.6?

When the verification of a PDI is conducted separately from the verification of the EOC, the EOC should be able to handle all correct PDI Files as well as all corrupted ones. "Normal range testing" should establish that the EOC can correctly manage any PDI File which complies with its structure, and whose element values comply with their attributes.

When the PDI File is separately verified, in accordance with ED-12C/ED-109A sections 6.6.a and 6.6.b, it will be demonstrated that it is compliant with its structure, and it contains element values that comply with their attributes and are correct. However, errors may be introduced between the separate verification of the PDI File and its use by the EOC. Examples of such errors are corruption during loading, corruption of the PDI File in memory, etc. Appropriate mitigation means should be established to manage these errors. These means may be external to the EOC (for example, detection of a partial PDI File load by the loading function/procedure, or hardware protection against single event upset), or internal to the EOC (for example, a cyclic redundancy check of the PDI File).

The second bullet in ED-12C/ED-109A section 6.6 which discusses robustness should be satisfied by establishing that the mitigation means developed in the EOC to address corrupted PDI Files have been verified to be appropriate and sufficient.

4.20.8 What is the rationale for assigning the PDI the same software/assurance level as the component using it and not using the assignment rules in ED-12C/ED-109A section 2.3.4?

ED-12C section 2.5.1 (section 2.6.1 in ED-109A) states that the PDI should be assigned the same software level (assurance level in ED-109A) as the components using it.

The rationale for this statement is that in many cases it may not be feasible to show that errors within the PDI File can be managed by protection mechanisms in the EOC using it. Also, in cases where demonstration is feasible, the additional work to verify the PDI File to the same software/assurance level as the component using it will be negligible; therefore, there would be no benefit in lowering the software level of the PDI.

However, if the applicant feels that it is beneficial, and can show that a different software/assurance level is appropriate (based on ED-12C/ED-109A section 2.3.4), this proposal may be negotiated with the certification/approval authority.

4.21 DP #21: CLARIFICATION ON SINGLE EVENT UPSET (SEU) AS IT RELATES TO SOFTWARE

Reference: ED-12C/ED-109A: Sections 4.5 and 11.1.b

Keywords: single event upset; SEU; SEU mitigation; system safety assessment

This discussion paper examines several questions regarding the relationship of single event upset (SEU) to software. The following questions are considered:

- What is a SEU? (see 4.21.1)
- When is it necessary to apply SEU mitigation measures in software? (see 4.21.2)
- What are some of the SEU mitigation techniques that can be implemented in software? (see 4.21.3)
- What other options are available for SEU mitigation? (see 4.21.4)
- What other information is available on single event upset (SEU) mitigation for avionics? (see 4.21.5)

4.21.1 What is a Single Event Upset (SEU)?

Single event effects (SEE) caused by atmospheric radiation have been recognized in recent years as a design issue for avionics systems, as well as high reliability ground-based systems. There are various types of these individual events which are the result of a single particle depositing sufficient energy to cause a disturbance in an electronic device. Single Event Upsets (SEUs) are random bit flip events. SEUs are transient soft errors, and are non-destructive. A reset or rewriting of the affected locations should result in normal device behavior.

4.21.2 When is it necessary to apply SEU mitigation measures in software?

The problem of SEU susceptibility is not a software issue; its origin is in hardware. As with many hardware shortcomings, it would be best if protection could be afforded in hardware. However, mitigation for its effects may be made in software or hardware. SEU susceptibility should be identified at system level taking into consideration the type of devices used, the probability of exposure to radiation/bit flip events, and the criticality level of the function. Some mitigation should be used where hardware protection is not available or is insufficient to meet the level of reliability required for the system. However, in deciding on software mitigation, it should be noted that this can be expensive in terms of performance and extent of verification needed. There may also be limits to the extent software mitigation can reasonably protect the system from the effects of SEU (for example, only memory containing key aircraft parameters). The need for SEU mitigation in software should be identified in the system safety assessment process.

4.21.3 What are some of the SEU mitigation techniques that can be implemented in software?

a. Where hardware detection or correction is unavailable, the following list of examples offer software solutions to the single event upset problems that may be experienced both by ground and avionics equipment. This list indicates some solutions available at the time of this discussion paper's publication and is not intended to be all inclusive. These solutions are also not guaranteed to work in all designs or to cover all SEU effects; specific optimum solutions should be sought out for a given architecture. The most common protection mechanisms against SEUs are error detection and correction mechanisms, such as:

   1. Parity, capable of detecting single bit errors.
   2. Cyclic redundancy code, capable of detecting errors.

3. Hamming code, normally capable of detecting two and correcting one bit error.
4. Reed-Solomon code, capable of correcting multiple symbol errors.
5. Convolutional codes, capable of correcting bursts of errors.
6. Watchdog timers, capable of detecting timing and scheduling errors.
7. Voting, capable of detecting and selecting most probable correct value.
8. Minimizing the use of unprotected cache memory.
9. Periodically flushing the cache so it has to be refilled.
10. Storing triple versions of critical values to use for voting.
11. Minimizing the use of random access memory (RAM) on the processor that has no protection mechanism or eliminating it altogether.
12. Minimizing stack and heap usage since dynamically changing memory space is difficult to protect in software (that is, utilize protection techniques such as those that protect function return addresses).
13. Using stack and heap overrun protection, use protection mechanisms built into the language (such as variable range checks in Ada) if possible (there are throughput penalties for this type of protection).
14. Minimizing the use of variables (including pointers) whose corruption could significantly impact the correct or expected behavior of the software (such as the storage of configuration pin data, or a pointer to the data, in unprotected RAM that could unexpectedly change system behavior at run-time).
15. Periodically performing checksums on critical areas of memory like non-volatile memory.
16. Performing checksums on permanently stored data before it is stored to prevent an SEU from becoming an unrecoverable failure.
17. Providing a mechanism whereby the application can be reset from the outside in the event that the application acts inappropriately.

b. There are additional practices that can be used where appropriate, including:

1. Repeated calculations in order to overcome transient errors.
2. Define constants in read only memory (ROM) locations, if possible.
3. Do not rely on RAM data to be accurate especially when that data has to be used for critical decisions/calculations. To address this, one may recalculate data every frame or periodically as needed by the criticality. Many parameters may be only determined one time, such as at power on. When stored in RAM, these parameters are vulnerable; therefore, such parameter usage should be minimized.
4. Write output discretes to hardware latches every frame. The hardware latch is susceptible to SEU. Don’t rely on the contents of the hardware latch being unchanged.

5. Consider using “>” or ”<” instead of "=", wherever possible. An example of this is in counter usage. Suppose a process is required to be executed every 30 frames and this is done with the following code:

```
begin module
    if count = 30
      then
          turn all outputs on for 15uS
          set count = 0
      end if
    inc count
end module
```

   If the RAM location that stored the 8-bit variable "count" had an SEU in bit 5 (or 6 or 7), then the value of count would be greater than 30. This would continue until "count" reached 255 at which point the “inc” operation would cause a wrap back to 0. With a frame time of 100mS, turning on of all outputs could take an additional 23.3 seconds (that is, instead of every 2 seconds, the output would occur at 25.3 seconds once and then recover). This could result in a nuisance fault that would cause the unit to be returned to the manufacturer where no fault would be found.

6. Continuously check the configuration state of devices that have been initialized by software. The data written to these devices defining the configuration could be changed by an SEU. If the configuration state of a hardware device cannot be checked continuously, then reset the device and re-write the configuration state, if a continuous monitor detects a failure with the device and declares the data failed. The device may get "healthy".

7. Filter input data. LRUs should not use data from the bus until the same value has been received twice in a row (2 transmit intervals from the source). Examples: 1) since some sources only update the data they send once every second, it will take at least ten 100 msec frames to filter that data. 2) when reading discrete inputs from hardware use a 2 out of 3 filter algorithm.

8. Whenever a BITE (built-in test equipment) or BIT (built-in test) detects a failure, rerun the test a second time to confirm the failure. An SEU could have caused the test to fail or changed the RAM location containing the pass-fail flag. SEUs in a hardware register may linger if the register is updated slower than the BIT test is run. Persistence algorithms can be designed based on how long the application can be in the faulted condition without detrimental effects. Default values can also be used while the failure is detected.

9. When using bi-directional I/O ports (an I/O port that can be programmed to be used as an input port or an output port), re-assert the configuration of the I/O port often. For example, some microprocessors require that a bi-directional I/O port have the state of the port's output buffer all 1's when the I/O port is to be used as an input. Typically the buffer powers up in the default of all 1's. If the design uses that port strictly as an input port and nowhere in the design is that state changed, then the design should never need to write 1's to the output buffer. This works well in the lab, but in the airplane an SEU can disable the ports ability to read inputs. (Note: Don’t trust the default power-up state. Always assert the state you need.)

10. Where registers are used to define the CPU configuration, the configuration should be re-asserted often. Use a "robust" watchdog timer (WDT) scheme. This could require hardware to be involved. A "smart" WDT could expect a rotating 8-bit pattern, which could be written from several different "software modules". Improper execution order would be detected and a microprocessor reset would cause recovery.

11. Pointers should be range checked when used so that if corruption has occurred, the error may be detected and dealt with appropriately. Similarly, integrator state values many need to be bounded and checked upon use.

c. Note that the example methods above are secondary to a system level design approach that uses a full set of methods to reduce the effects of SEUs on equipment. Hardware methods, software methods, and redundancy can contribute to robust system designs. As processors have progressed in speed and complexity, and as the use of high-level languages has increased, it has become more difficult to structure software that will reduce functional failures caused by SEUs. Consider the following:

1. The aforementioned solutions could be applied to scenarios like these:

- The programmer structures the software as a “main” program with a group of subroutines that are called to carry out much of the logic of the program. Each time a subroutine is called, the return address is written to memory. If the image of the return address is corrupted by an SEU while the subroutine is executing, then the software will not return to the correct point in the instruction sequence. Instead, it will transfer program control to some unpredictable point and the software will crash.

- If software runs as multiple tasks (or threads, or processes, under the control of an operating system) it is necessary that the operating system record the “state” of all tasks that are not currently executing in memory resident structures (for example, task control blocks). In the task control blocks, the operating system records many things, including the address where the task is to resume execution and the values of all the processor’s registers relevant to the task. If any of this information is corrupted by a SEU while the task is suspended, then when task next resumes execution it is likely to fail.

2. These techniques are not appropriate for general purpose RAM for the following reasons:

- Software techniques to detect and correct SEU in otherwise SEU protected hardware constitutes unnecessary code that appears to be contributing to robustness. Unnecessary code potentially leads to overly complex software with little or no gain in safety.

- The stack is the weak link in RAM. For example, the very functions that may be doing a double computation to mitigate SEU may not return properly due to SEU corruption of the return address on the stack.

- Sometimes programs are loaded into RAM due to more efficient memory accesses. SEU mitigation techniques may interfere with such techniques.

3. Some techniques may create an illusion of absolute safety while only reducing the probability of data corruption.

4. Some of these techniques may apply to special memory such as memory mapped registers which may be difficult to protect with a hardware circuit but don't affect the execution or control flow of the software in the presence of SEU events.

d. In general, these software mitigation measures work through refreshing and monitoring hardware status, configuration, and stored data. Software-implemented SEU mitigation may result in software that executes slower, requires more CPU cycles, requires more memory, and costs more to design and verify; but may reduce the effects of SEU and other transient effects. Hardware with integrated SEU detection or correction (for example, parity and error correction code (ECC)) is also becoming increasingly available, and careful consideration should be made in hardware selection and software architecture to accommodate the preferred SEU mitigation strategy while still meeting functional and performance requirements.

4.21.4 What other options are available for SEU mitigation?

Modern microprocessors are increasingly providing features that greatly reduce their susceptibility to SEUs. On-processor memory controllers frequently include the logic required to perform error detection and correction of data contained in the main system RAM. Some current processors include level-1 and level-2 caches on-processor with the processor logic; processors are available with parity error detection for the level-1 cache and ECC logic that provides error detection and correction for the level-2 cache. These measures reduce the incidence of function failure caused by SEUs by a large factor.

SEUs can be architecturally mitigated either by using redundant hardware or by using radiation-hardened parts. However, if the appliance contains software or if there are weight, volume, and cost constraints, hardware mitigation alone may not be practical. An option may be to put the entire processor through radiation bombardment in laboratory settings and observe processor behavior. However, testing such as this is potentially as costly and as impractical as the hardware only solution. Certain processors are marketed with known radiation susceptibility effects published in their data logs. Positioning of the processor to have the least surface area exposed to the radiation direction, or controlling the use of memory are some of the techniques used to reduce SEU effects. However, some of these practices may not be feasible on certain applications.

4.21.5 What other information is available on single event upset (SEU) mitigation for avionics?

First, section 3.1.20 of International Electrotechnical Commission Technical Specification (IEC TS) 62239 "Process management for avionics – Preparation of an electronic components management plan" defines single event effects as: "*response of a component caused by the impact of galactic cosmic rays, solar enhanced particles and/or energetic neutrons and protons. The range of responses can include both non-destructive (for example upset) and destructive (for example latch-up or gate rupture) phenomena.*"

Destructive single event effects can only be mitigated by avoidance using selected hardware that is less prone to latch-up or hardware mitigation methods such as radiation hardening or by an architectural design that allows correct operation even when hard failures have occurred.

According to IEC TS 62239 (section 4.2.6 Avionics radiation environment): "*The documented processes shall verify that the component will operate successfully in the application with regard to the effects of atmospheric radiation. These include single event upset (SEU), single event latch-up (SEL), single event burnout (SEB) and total dose radiation for any identified application where it is a concern. If radiation effects are accommodated by the equipment design, then the method of accommodation shall be documented in the equipment design records*."

If an avionics manufacturer has developed an Electronic Component Management Plan, it is expected that the plan would address any hardware mitigation measures for avionics.

Additionally, IEC TS 62396 aimed at Single Event Effects (which include SEU, SEL, and SEB) consists of the following parts, under the general title “Process management for avionics – Atmospheric radiation effects”:

- Part 1: Accommodation of atmospheric radiation effects via single event effects within avionics electronic equipment.
- Part 2: Guidelines for single event effects testing for avionics systems.
- Part 3: Guidelines to optimize avionics system design to reduce single event effects rates.
- Part 4: Guidelines for designing with high voltage aircraft electronics and potential single event effects.
- Part 5: Guidelines for assessing thermal neutron fluxes and effects in avionics systems.

Part 3 provides guidance relative to systems and software design.

CHAPTER 5

RATIONALE FOR ED-12C/ED-109A

5.1 INTRODUCTION

5.1.1 Overview

This section provides background information and rationale for the objectives contained in ED-12C/ED-109A and the ED-12C/ED-109A supplements. Since ED-12C/ED-109A heavily borrows from standard software engineering principles that are well understood, rationale is only provided for those elements within the document that are specific to aircraft certification (or CNS/ATM system approval). The reader is directed to the public literature for rationale for items not covered in this section. ED-12C/ED-109A adds objectives and required independence as a function of software level which is not addressed in generally accepted software engineering practices. This allowed ED-12C/ED-109A to be consistent with the safety categories defined in safety analysis.

The following areas of the document were reviewed and considered for the need for rationale.

- Sections of ED-12C/ED-109A.
- Associated tables in Annex A, including the references to the relevant section(s) of the document and supplements or applicable cross-references of ED-94C.

It should be noted that both ED-12C and ED-109A are considered in this rationale (as they are throughout this ED-94C document). The terminology relationship for the ED-12C airborne software and the ED-109A CNS/ATM software are shown below and are used throughout this rationale.

ED-12C Terminology	ED-109A Terminology
Airborne	CNS/ATM
Certification	Approval
Airworthiness requirements	Applicable approval requirements
Certification liaison process	Approval liaison process
Parameter Data Item Files	Adaptation Data Item Files
Plan for Software Aspects of Certification (PSAC)	Plan for Software Aspects of Approval (PSAA)

5.1.2 Rationale for ED-12C/ED-109A Development

The content of ED-12C/ED-109A contains objectives that reflect the generally accepted principles of software engineering documented in the literature and other standards. It also provides additional objectives where traditional methods were not deemed sufficient for airborne or CNS/ATM software. The continued software technology advances determined the need for improved guidance and resulted in an effort to revise ED-12B/ED-109 to leverage technology for use in the aviation industry. The revised guidance is intended to provide an acceptable means of compliance to achieve approval of the software aspects of airborne and equipment or CNS/ATM systems.

The primary purposes of the revision to ED-12B/ED-109 are:

- To continue to promote safe implementation of aviation software.
- To provide clearer and more consistent relationships to the systems development processes and safety assessment processes.
- To address emerging trends and technologies in software development.
- To provide an approach that is flexible and allows changes in technology.

5.2 RATIONALE FOR ED-12C/ED-109A SECTION 2: SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT

Software development is part of the system development process. A high-level definition of the interfaces between the system development and software development processes is necessary to establish context. Software/assurance levels and allocated system requirements are a result of the system development and safety assessment processes. The system safety assessment process requires that software be associated with a failure condition category and be assigned a software/assurance level. Software/assurance levels are associated with an appropriate design assurance rigor needed to ensure an appropriate software design, achievable intended function, and freedom from anomalous behavior.

Since software associated with lower hazard categories does not require the same rigor as higher hazard categories, ED-12C/ED-109A provides varying levels of rigor for each of the objectives and evidence. The system development process also provides constraints on the software development process that should be communicated from the system and safety process to the developers. Because software always executes as part of a system, an overview of the system aspects related to software development is provided in ED-12C/ED-109A section 2.

5.3 RATIONALE FOR ED-12C/ED-109A SECTION 3: SOFTWARE LIFE CYCLE

The committee wanted to avoid prescribing any specific development methodology. This section allows for a software life cycle to be defined with any suitable life cycle model(s) to be chosen for software development. This is further supported by the introduction of “transition criteria”. Specific transition criteria between one process and the next are not prescribed, rather ED-12C/ED-109A states that transition criteria should be defined and adhered to throughout the development life cycle(s) selected. The approach documented in section 3 allows the accommodation of different life cycles.

5.4 RATIONALE FOR ED-12C/ED-109A SECTION 4: SOFTWARE PLANNING PROCESS

The majority of ED-12C/ED-109A section 4 summarizes the standard software engineering principles of planning. Additions were needed to address items that were unique to avionics or CNS/ATM software as identified below:

- Requirements were added to ensure that the special topics of multiple-version dissimilar software, deactivated code, dead code, user-modifiable code, parameter data item, and additional considerations related to the regulatory environment are addressed.
- Provisions were included for the creation of documents used to negotiate acceptable approaches at the start of a project (the PSAC or PSAA), as well as a document that summarizes the development at the end of the project and can be used by the certification/approval authorities to accept the software as satisfying the objectives of ED-12C/ED-109A (the SAS).
- Because of the lack of uniformity in industry of defining the configuration of the product, data descriptions for how to describe the life cycle environment and the configuration of the delivered product were introduced.

- Since the treatment of compilers in non-aviation programs did not have the necessary rigor needed for safety-critical systems, a section was created to ensure that all compiler functionality was fully verified in the product.
- ED-12C/ED-109A section 4 also introduced categorization of requirements into high-level, low-level, and derived requirements to avoid the widely differing definitions of requirements and design.
- The relationship of the requirements development process to the safety process was defined to ensure that the safety analysis would not be compromised by either the improper implementation of safety-related requirements or the introduction of new behavior (that is, derived requirements) that was not envisioned in the original safety analysis.

ED-12C/ED-109A section 4 describes objectives and activities which ensure that planning is part of the software life cycle processes. These are summarized in ED-12C/ED-109A Annex A Table A-1. Many of the objectives reflect generally accepted software engineering principles that are not unique to aviation software. These do not require any additional rationale. The aviation community needed to ensure that the applicant has reviewed their plans for compliance with ED-12B/ED-109A and that they were consistent with each other. As this was not consistently done in normal software engineering practice the objectives 6 and 7 in ED-12C/ED-109A Table A-1 were created. Likewise, objective 4 in ED-12C/ED-109A Table A-1 ensured that the applicant addressed those items unique to certification (or CNS/ATM system approval) and highlighted any non-standard approaches so their risks to safety could be assessed.

5.5 RATIONALE FOR ED-12C/ED-109A SECTION 5: SOFTWARE DEVELOPMENT PROCESSES

ED-12C/ED-109A section 5 addresses the standard engineering practices for establishing the processes and their execution as defined in the development plans for requirements, design, code, and integration.

5.5.1 Rationale for Table A-2 Objectives

The corresponding objectives 1, 2, 3, 4, 5, 6, and 7 in ED-12C/ED-109A Table A-2 correspond closely with standard engineering practices for systematic requirements and design decomposition and the associated activities for assuring that all derived requirements, both high level and low level are fully captured. The only clarification necessary for these objectives was to align the terminology as follows:

- High-level requirements terminology corresponds roughly with terms like software requirements, software specifications, etc., and low-level requirements terminology corresponds roughly with terms like software design, detailed design, etc.
- While software architecture description does have some correspondence to the same terminology in standard software engineering practices, other terms such as high-level design are also used.
- Derived requirements are those additional requirements which may not be directly traceable to higher level requirements and which result from the software development processes.

5.6 RATIONALE FOR ED-12C/ED-109A SECTION 6: SOFTWARE VERIFICATION PROCESS

ED-12C/ED-109A section 6 addresses the overall verification strategy of layers of assurance employed to address full requirement and design incorporation and also to assure error detection and removal early in the execution of the development processes.

5.6.1 Rationale for ED-12C/ED-109A Table A-3, A-4, and A-5 Objectives

ED-12C/ED-109A section 6 and the corresponding objectives in Tables A-3, A-4, and A-5 correspond closely with standard engineering practices for review and analysis verification activities for requirements, design, and code.

The following objectives were added or modified to address the unique aspects of aircraft certification or CNS/ATM system approval. The rationales for these are shown below.

- The objective for algorithm accuracy found in ED-12C/ED-109A Tables A-3, A-4, and A-5 may have some counterparts in standard software engineering practices; however, additional clarification was provided in ED-12C/ED-109A to highlight the heavy use of computation and effect of computational error on safety.
- The objectives 8 and 9 in ED-12C/ED-109A Table A-5 were added to address the additional concerns for ensuring the use of Parameter Data Item Files (Adaptation Data Item Files in ED-109A) and to provide for fully verifying this type of architecture.
- There are no universally accepted standard software engineering practices that would ensure that deterministic architectures will be produced. In order to satisfy the safety criteria for predictability objectives 11 and 13 in ED-12C/ED-109A Table A-4 were added. By requiring architectures to be verifiable, Objective 11 excludes architectures which cannot be shown to be correct or result in non-predictable behaviors.
- While there are many approaches to achieving partitioning, not all of the approaches can be shown to satisfy the safety analysis properties required for equipment certification or CNS/ATM systems approval. By requiring the applicant to demonstrate that partitioning schemes can be fully verified through analysis, review, and test, objective 13 excludes those partitioning architectures that cannot substantiate claims of non-interference. This objective also ensures that there is documented evidence available which will allow the certification or approval authorities to validate an applicant's claim for partitioning.

5.6.2 Rationale for ED-12C/ED-109A Table A-6 Objectives

There is an extensive body of work including books, research, standards, and published articles regarding testing of software. Of the many techniques available, ED-12C/ED-109A adopts a strategy called equivalence class testing. In this approach, the inputs and outputs are divided into sets wherein the results of testing of members of the set can be allowed to act as proxy for any member in the set. The rationale for the testing objectives are summarized below:

- Standard software engineering practices contain extensive information on the testing of software, but this information provides confusing conclusions on what credit can be taken for test cases based on the Source Code. The objectives 1, 2, 3, and 4 in ED-12C/ED-109A Table A-6 provide clarification that testing credit can only be obtained by testing of the high-level and low-level requirements.
- The non-aviation community does allow credit for testing in a non-target environment, however, this could allow certain errors that are target-related, as well as compiler target-specific errors, to escape detection. Therefore, the objective 5 in ED-12C/ED-109A Table A-6 specifies that credit is obtained by testing on the target or showing that any other testing is equivalent to target-level testing.
- While standard software engineering practices have guidance on stress testing, ED-12C/ED-109A placed special emphasis on this based on field experience with earlier versions of the document. Therefore, objectives 2 and 4 in Table A-6 were added in ED-12B/ED-109A to provide this emphasis.

5.6.3 Rationale for ED-12C/ED-109A Table A-7 Objectives

The standard engineering practices contain comprehensive information on how test cases, procedures, and results need to be evaluated. However, these same practices do not provide any consensus guidance on testing completion criteria.

While ED-12C/ED-109A echoes much of the standard software engineering testing processes, it emphasizes key issues critical to safety as well as adopting coverage criteria for Level A (or Assurance Level 1) that wasn't generally identified in the standard software engineering practices for testing. The rationale for the additional emphasis and addition of the coverage criteria are described in the following sections:

- It was important to establish that any software installed on the aircraft (or in the CNS/ATM system) did not contain any unexplored behavior. The depth of analysis to ensure this has to be a function of the software/assurance level. As a consequence objectives 3 through 9 in ED-12C/ED-109A Table A-7 were developed to ensure that the behaviors instantiated in the Executable Object Code developed from high-level requirements, low-level requirements, Source Code, and architecture were fully explored. It also assured that any behaviors not represented by these elements would be detected.
- Objectives 3 and 4 in ED-12C/ED-109A Table A-7 follow generally accepted software engineering practices. In some cases informal and non-documented activities were sufficient. The objectives provide requirements for creating well-defined data that can be validated by certification/approval authorities.
- Traceability between high-level requirements, low-level requirements and Source Code provides some assurance that no unspecified functionality exists; however, traceability does not provide assurance:
    - That all of the functionality was tested or that the Executable Object Code is correct since there are no reviews or traceability that ties the Executable Object Code to the Source Code.
    - That the compiler did not inject some functionality that was not specified in the Source Code.

      Therefore, objectives 5, 6, 7, and 9 in ED-12C/ED-109A Table A-7 were created to address these issues.
- Objectives 5, 6, and 7 in ED-12C/ED-109A Table A-7 ensure that test cases written for requirements explore the Source Code with the degree of rigor required by the software/assurance level. For level C (AL 3), it was deemed satisfactory that demonstrating that all statements in the Source Code were explored by the set of test cases. For level B (AL 2) the addition of the requirement that all decision paths in the source was considered sufficient to address the increase in the associated hazard category.

  However for level A (AL 1), the committee established that all logic expressions in the Source Code should be explored. The use of techniques such as multiple condition decision coverage, or exhaustive truth table evaluation to fully explore all of the logic was considered impractical.

  A compromise was achieved based on experience gained from three aircraft programs, where an approach derived from hardware logic testing that concentrated on showing that each term in a Boolean expression can be shown to affect the result, was applied to software. The term for this type of coverage was Modified Condition/Decision Coverage (MC/DC).

- Objective 8 in ED-12C/ED-109A Table A-7 ensures test coverage with respect to the software architecture, specifically, the data flow between software components and the control of software component execution. The intent behind this objective is to ensure that applicants do a sufficient amount of hardware/software integration testing and/or software integration testing to verify that the software architecture is correctly implemented with respect to the requirements.
- The above objectives did not address the issue of additional functionality added by the compiler. Objective 9 in ED-12C/ED-109A Table A-7 ensures that the Executable Object Code or its proxy is evaluated for any functionality added by the compiler and ensure that such functionality is verified or analyzed to ensure that it has no safety impact.

5.7 RATIONALE FOR ED-12C/ED-109A SECTION 7: SOFTWARE CONFIGURATION MANAGEMENT PROCESS

The content of ED-12C/ED-109A section 7 and the associated objectives in Table A-8 reflect generally accepted software engineering practices. Two levels of configuration control are identified and employed to allow reduction of configuration control activities for artifacts that do not require rigorous configuration control for items such as Software Verification Results data.

5.8 RATIONALE FOR ED-12C/ED-109A SECTION 8: SOFTWARE QUALITY ASSURANCE PROCESS

The generally accepted software engineering practices for quality assurance contain activities for process improvement, technical evaluation, and other criteria that were not deemed necessary by the committee based on the nature of certification (or CNS/ATM systems approval).

When a system is certified/approved, it should be shown to meet the guidelines and requirements at the time of certification/approval and cannot rely on a continuous improvement process. As a result, only the essential elements from the generally accepted software quality assurance practices that were unique to embedded avionics (or CNS/ATM) systems were adopted. These resulted in the objectives in in ED-12C/ED-109A Table A-9.

5.9 RATIONALE FOR ED-12C/ED-109A SECTION 9: CERTIFICATION/APPROVAL LIAISON PROCESS

Under ED-12A there was considerable confusion over which life cycle data had to be delivered to and available for the certification authority. To address these issues, ED-12C section 9 provides clarification of which of the software life cycle data that is considered part of type design used for regulatory purposes and provides the means to establish the critical relationships and agreements with the certification authority at the beginning of a certification program that will be used for the basis for the approval of the software aspects of the system at the end of the program. These constitute the objectives in Table A-10.

ED-109A section 9 is entitled “Approval Liaison Process”. Like ED-12C section 9, it explains what software life cycle data is considered part of the software approval process and provides the means to establish the critical relationships and agreements with the approval authority at the beginning of the CNS/ATM system project. The objectives for the approval liaison process are included in Table A-10 of ED-109A.

5.10 RATIONALE FOR ED-12C/ED-109A SECTION 10: OVERVIEW OF CERTIFICATION (CNS/ATM SYSTEM APPROVAL) PROCESS

While there are regulations and guidance material for aircraft and engine certification, during the development of ED-12B, the committee decided that there was no description on how software related to this material. Therefore, a need was identified to have a summary of the aircraft and engine certification process. Section 10 of ED-12C provides a high-level description of the interface and coordination between the developer, applicants, and the certification authority. Likewise, ED-109A section 10 provides a high-level overview of the approval process with respect to software aspects of CNS/ATM systems and equipment.

5.11 RATIONALE FOR ED-12C/ED-109A SECTION 11: SOFTWARE LIFE CYCLE DATA

The standard engineering practices contain comprehensive information on software life cycle data. However, these needed to be tailored to the specific requirements of certification/approval and to be consistent with the terminology of the rest of the document. This required the inclusion of ED-12C/ED-109A sections 11.1 through 11.22 to specify the content requirements for the various Software Life Cycle Data (compliance evidence) needed to demonstrate satisfaction of the various objectives specified within ED-12C/ED-109A.

5.12 RATIONALE FOR ED-12C/ED-109A SECTION 12: ADDITIONAL CONSIDERATIONS

ED-12C/ED-109A section 12 addresses additional topics that may need to be considered in addition to sections 3 through 11 in order to provide consistent guidance for these focused topics, for example, multi-version dissimilar software, service history, etc.

5.13 RATIONALE FOR TOOL QUALIFICATION DOCUMENT AND ED-12C/ED-109A SUPPLEMENTS

Supplements were implemented to provide guidelines for technology advances in software engineering. When these technologies are used, it was not always clear how the objectives of the core ED-12C/ED-109A document are mapped to the terminology and approach used by a specific technology. Each Supplement adds, modifies, or deletes the ED-12C/ED-109A objectives to make the guidance more clear.

5.13.1 Rationale for “SOFTWARE TOOL QUALIFICATION CONSIDERATIONS”

Tools are widely used to develop, verify, configure, and control software. A tool is a computer program or a functional part thereof, used to help develop, transform, test, analyze, produce, or modify another program, data, or its documentation. Examples are an automated code generator, a compiler, test tools, and modification management tools. The document ED-215, entitled “Software Tool Qualification Considerations,” explains the process and objectives for qualifying tools. ED-12C/ED-109A, section 12.2 provides the criteria to determine if a tool needs to be qualified. If a tool does require qualification, ED-215 provides the guidance and objectives for how to obtain tool qualification.

ED-215 was developed for the following reasons.

a. Tools are different from the software using the tools and form a unique domain; therefore, tool-specific guidance for both tool developers and tool users is needed.

b. Tools are often developed by teams other than those who use the tools to develop software. These tool development teams frequently do not have software guidance background (examples of guidance include ED-12B, ED-12C, or ED-109A). The tool-specific document benefits tool development teams and helps them avoid confusion and misinterpretation.

c. ED-215 provides tool-specific guidance for airborne and ground-based software. It may also be used by other domains, such as automotive, space, systems, electronic hardware, aeronautical databases, and safety processes.

5.13.2 Rationale for "FORMAL METHODS SUPPLEMENT TO ED-12C AND ED-109A"

Formal methods are mathematically based techniques for the specification, development, and verification of computer systems. The use of formal methods is motivated by the fact that performing rigorous mathematical analyses can contribute to establishing the correctness and robustness of software aspects of safety-critical systems. Such techniques are also highly applicable to complex hardware development assurance. The formal logic, discrete mathematics, and computer-readable languages underpinning formal methods, provide a solid and defensible foundation for many of the development and verification activities required for avionics software. Nonetheless, the avionics industry at large has been hesitant to adopt formal methods, despite growing evidence of their benefits.

Since the introduction of formal methods as an alternative method in section 12.3.1 of ED-12B, advances and practical experience have been gained in techniques and tools supporting formal methods, to the extent that they have become sufficiently mature for application on today's avionics products. The Formal Methods Supplement was authorized, consequently, to provide guidance for applicants and certification/approval authorities to facilitate the use of formal methods. The supplement is based on the following key principles:

- A formal method is the application of a formal analysis to a formal model.
- A formal model must be in a notation with mathematically defined syntax and semantics.
- Formal methods may be used at different verification steps in the software life cycle, for all or part of a step and for all or part of the system being developed.
- A formal method must never produce a result which may not be true (that is, the formal analysis must be sound).
- It is possible to apply the results of formal analysis of Source Code to the corresponding object code by understanding the compilation, link, and load processes in sufficient detail.
- Test is always required to ensure compatibility of the software with target hardware and to fully verify the understanding of the relationship between source and object code.

5.13.3 Rationale for "OBJECT-ORIENTED TECHNOLOGY AND RELATED TECHNIQUES SUPPLEMENT TO ED-12C AND ED-109A"

Object-oriented technologies (OOT) have been widely adopted in non-critical software development projects. The use of these technologies for critical software applications in avionics has increased, but a number of issues need to be considered to ensure the safety and integrity goals are met.

These issues are directly related to language features and to complications encountered with meeting well-established safety objectives. There are a number of additional language features that are part of OOT that need to be considered as well. Clarifying each issue will ease the application of object-oriented technology and related techniques (OOT&RT).

The OOT&RT Supplement was authorized, consequently, to provide guidance for applicants and certification/approval authorities to facilitate the use of OOT in the development process. The supplement addresses both object-oriented technology and related techniques and includes the following key elements:

- Basic concepts of OOT include classes and objects, types and type safety, hierarchical encapsulation, polymorphism, function passing and closures, and method dispatch.
- Related techniques include parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, virtualization, and component-based development.
- Key features include inheritance, parametric polymorphism, overloading, type conversion, software exceptions and exception handling, and dynamic memory management.
- An annex is included to assist in vulnerability analysis for OOT&RT.
- Frequently asked questions are included in an appendix.

5.13.4 Rationale for "MODEL-BASED DEVELOPMENT AND VERIFICATION SUPPLEMENT TO ED-12C AND ED-109A"

Model-based development and verification technology involves methods and techniques to represent requirements in the form of a model, typically a graphical model, to facilitate the development and/or verification of software. The model-based techniques are rarely, if ever, used as the sole means to develop or verify software since model-based techniques may not be the optimum choice for all requirements.

The use of model-based development and verification technology in safety-critical airborne applications pre-dates ED-12B and no special guidance was included in ED-12B to address the use of models. As tools and techniques have evolved with the use of models, auto-code generation, simulation, and test automation, there has not been a consistent understanding and application of how the ED-12B (and hence, ED-109) objectives mapped to the systems and software aspects of model-based requirements and associated verification activity.

The Model-based Development and Verification Supplement was authorized, consequently, to provide guidance for applicants and certification/approval authorities to facilitate the use of models in the development and verification processes. The supplement is based on the following key principles:

- Models can represent high-level and/or low-level requirements.
- More than one type of model may be used within a development or verification process.
- Models are derived from, traced to, and verified against higher level requirements.
- Simulation may utilize models to meet development and verification objectives.
- Test is always required to ensure compatibility of the software with target hardware and to fully verify the understanding of the relationship between source and object code.

APPENDIX A

ACRONYMS

Acronym	Meaning
14CFR	Title 14 of the Code of Federal Regulations
AL	(ED-109A) Assurance Level
AMC	Acceptable Means of Compliance
ARINC	Aeronautical Radio, Incorporated
ARP	Aerospace Recommended Practice
ARP4754	SAE document entitled "Guidelines for Development of Civil Aircraft and Systems"
BIT	Built-In Testing
BITE	Built-In Test Equipment
CC1	Control Category 1
CC2	Control Category 2
CFR	Code of Federal Regulations
CI	Configuration Item
CM	Configuration Management
CMMI	Capability Maturity Model Integration
CNS/ATM	Communication, Navigation, Surveillance, and Air Traffic Management
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CS	Certification Specifications
DO-178	RTCA document entitled "Software Considerations in Airborne Systems and Equipment Certification"
DP	Discussion Paper
EASA	European Aviation Safety Agency
ECC	Error Correction Code
ED-12	EUROCAE document entitled "Software Considerations in Airborne Systems and Equipment Certification"
ED-79	EUROCAE document entitled "Guidelines for Development of Civil Aircraft and Systems"
EMI	Electromagnetic Interference
EOC	Executable Object Code
ETSO	European Technical Standard Order
EUROCAE	European Organisation for Civil Aviation Equipment
FAA	Federal Aviation Administration
FAQ	Frequently Asked Question
FPU	Floating-point Processing Unit
HLR	High-Level Requirement
IEC	International Electrotechnical Commission
IEC TS	International Electrotechnical Commission Technical Specification
IEEE	Institute of Electrical and Electronic Engineers
IMA	Integrated Modular Avionics
I/O	Input and/or Output
LRU	Line Replaceable Unit
LLR	Low-Level Requirement
MC/DC	Modified Condition Decision Coverage
MMU	Memory Management Unit
NDI	Non-Developmental Item
OCC	Object Code Coverage
OOT	Object-Oriented Technology
OOT&RT	Object-Oriented Technology and Related Techniques
OPR	Open Problem Report
OS	Operating System
PDI	Parameter Data Item
PDS	Previously Developed Software
PMC	Program Management Committee
PSAA	Plan for Software Aspects of Approval
PSAC	Plan for Software Aspects of Certification
RAM	Random Access Memory
ROM	Read Only Memory
RTCA	RTCA, Inc. is an association of aeronautical organizations in the United States
SAE	Society of Automotive Engineers
SAS	Software Accomplishment Summary
SCI	Software Configuration Index
SC-205	RTCA Special Committee #205
SCM	Software Configuration Management
SDP	Software Development Plan

Acronym	Meaning
SECI	Software Life Cycle Environmental Configuration Index
SEE	Single Event Effects
SEI	Software Engineering Institute
SEB	Single Even Burst
SEL	Single Event Latch-up
SEU	Single Event Upset
SPICE	Software Process Improvement Capability Evaluation
SQA	Software Quality Assurance
SSA	System Safety Assessment
TS	Technical Specification
TSO	Technical Standard Order
TSOA	Technical Standard Order Authorization
WCET	Watchdog Timer
WDT	Worst Case Execution Timing
WG-71	EUROCAE Working Group #71
XML	Extensible Markup Language

APPENDIX B

COMMITTEE MEMBERSHIP

EXECUTIVE COMMITTEE MEMBERS

Jim Krodel, Pratt & Whitney	SC-205 Chair
Gérard Ladier, Airbus/Aerospace Valley	WG-71 Chair
Mike DeWalt, Certification Services, Inc./FAA	SC-205 Secretary (until March 2008)
Leslie Alford, Boeing Company	SC-205 Secretary (from March 2008)
Ross Hannan, Sigma Associates (Aerospace)	WG-71 Secretary
Barbara Lingberg, FAA	FAA Representative/CAST Chair
Jean-Luc Delamaide, EASA	EASA Representative
John Coleman, Dawson Consulting	Sub-group Liaison
Matt Jaffe, Embry-Riddle Aeronautical University	Web Site Liaison
Todd R. White, L-3 Communications/Qualtech	Collaborative Technology Software Liaison

SUB-GROUP LEADERSHIP

SG-1 – Document Integration

Ron Ashpole, SILVER ATENA	SG-1 Co-chair
Tom Ferrell, Ferrell and Associates Consulting	SG-1 Co-chair (until March 2008)
Marty Gasiorowski, Worldwide Certification Services	SG-1 Co-chair (from March 2008)
Tom Roth, Airborne Software Certification Consulting	SG-1 Secretary

SG-2 – Issues and Rationale

Ross Hannan, Sigma Associates (Aerospace)	SG-2 Co-chair
Mike DeWalt, Certification Services, Inc./FAA	SG-2 Co-chair (until March 2008)
Will Struck, FAA	SG-2 Co-chair (until March 2009)
Fred Moyer, Rockwell Collins	SG-2 Co-chair (from April 2009)
John Angermayer, Mitre	SG-2 Secretary

SG-3 – Tool Qualification

Frédéric Pothon, ACG Solutions	SG-3 Co-chair
Leanna Rierson, Digital Safety Consulting	SG-3 Co-chair
Bernard Dion, Esterel Technologies	SG-3 Co-secretary
Gene Kelly, CertTech	SG-3 Co-secretary (until May 2009)
Mo Piper, Boeing Company	SG-3 Co-secretary (from May 2009)

SG-4 – Model-Based Development and Verification

Pierre Lionne, EADS APSYS	SG-4 Co-chair
Mark Lillis, Goodrich GPECS	SG-4 Co-chair
Hervé Delseny, Airbus	SG-4 Co-chair
Martha Blankenberger, Rolls-Royce	SG-4 Secretary

SG-5 – Object-Oriented Technology

Peter Heller, Airbus Operations GmbH	SG-5 Co-chair (until February 2009)
Jan-Hendrik Boelens, Eurocopter	SG-5 Co-chair (Feb 2009 to August 2010)
James Hunt, aicas	SG-5 Co-chair (from August 2010)
Jim Chelini, Verocel	SG-5 Co-chair (until Oct 2009)
Greg Millican, Honeywell	SG-5 Co-chair (Oct 2009 to August 2011)
Jim Chelini, Verocel	SG-5 Co-chair (from August 2011)

SG-6 – Formal Methods

Duncan Brown, Aero Engine Controls (Rolls-Royce)	SG-6 Co-chair
Kelly Hayhurst, NASA	SG-6 Co-chair

SG-7 – Special Considerations and CNS/ATM

David Hawken, NATS	SG-7 Co-chair (until June 2010)
Jim Stewart, NATS	SG-7 Co-chair (from June 2010)
Don Heck, Boeing Company	SG-7 Co-chair
Leslie Alford, Boeing Company	SG-7 Secretary (until March 2008)
Marguerite Baier, Honeywell	SG-7 Secretary (from March 2008)

EUROCAE Representative:

Gilbert Amato	EUROCAE (until September 2009)
Roland Mallwitz	EUROCAE (from October 2009)

RTCA Representative:

Rudy Ruana	RTCA Inc. (until September 2009)
Ray Glennon	RTCA Inc. (until March 2010)
Hal Moses	RTCA Inc. (until August 2010)
Cyndy Brown	RTCA Inc. (until August 2011)
Hal Moses	RTCA Inc. (from August 2011)

EDITORIAL COMMITTEE

Leanna Rierson, Digital Safety Consulting	Editorial Committee Chair
Ron Ashpole, SILVER ATENA	Editorial Committee
Alex Ayzenberg, Boeing Company	Editorial Committee
Patty (Bartels) Bath, Esterline AVISTA	Editorial Committee
Dewi Daniels, Verocel	Editorial Committee
Hervé Delseny, Airbus	Editorial Committee
Andrew Elliott, Design Assurance	Editorial Committee
Kelly Hayhurst, NASA	Editorial Committee
Barbara Lingberg, FAA	Editorial Committee
Steven C. Martz, Garmin	Editorial Committee
Steve Morton, TBV Associates	Editorial Committee
Marge Sonnek, Honeywell	Editorial Committee

COMMITTEE MEMBERSHIP

Kyle Achenbach	Rolls-Royce
Dana E. Adkins	Kidde Aerospace
Leslie Alford	Boeing Company
Carlo Amalfitano	Certon Software, Inc
Gilbert Amato	EUROCAE
Peter Amey	Praxis High Integrity Systems
Allan Gilmour Anderson	Embraer
Håkan Anderwall	Saab AB
Joseph Angelo	NovAtel Inc, Canada
John Charles Angermayer	Mitre Corp
Robert Annis	GE Aviation
Ron Ashpole	SILVER ATENA
Alex Ayzenberg	Boeing Company
Marguerite Baier	Honeywell
Fred Barber	Avidyne
Clay Barber	Garmin International
Gerald F. Barofsky	L-3 Communications
Patty (Bartels) Bath	Esterline AVISTA
Brigitte Bauer	Thales
Phillipe Baufreton	SAGEM DS Safran Group
Connie Beane	ENEA Embedded Technology Inc
Bernard Beaudouin	EADS APSYS
Germain Beaulieu	Independent Consultant
Martin Beeby	Seaweed Systems
Scott Beecher	Pratt & Whitney
Haik Biglari	Fairchild Controls
Peter Billing	Aviya Technologies Inc
Denise Black	Embedded Plus Engineering
Brad Blackhurst	Independent Consultant
Craig Bladow	Woodward
Martha Blankenberger	Rolls-Royce
Holger Blasum	SYSGO
Thomas Bleichner	Rohde & Schwarz
Don Bockenfeld	CMC Electronics
Jan-Hendrik Boelens	Eurocopter
Eric Bonnafous	CommunicationSys
Jean-Christophe Bonnet	CEAT
Hugues Bonnin	Cap Gemini
Matteo Bordin	AdaCore
Feliks Bortkiewicz	Boeing
Julien Bourdeau	DND (Canada)
Paul Bousquet	Volpe National Transportation Systems Center
David Bowen	EUROCAE
Elizabeth Brandli	FAA
Andrew Bridge	EASA
Paul Brook	Thales
Daryl Brooke	Universal Avionics Systems Corporation
Duncan Brown	Aero Engine Controls (Rolls-Royce)
Thomas Buchberger	Siemens AG
Brett Burgeles	Consultant

Bernard Buscail	Airbus
Bob Busser	Systems and Software Consortium
Christopher Caines	QinetiQ
Cristiano Campos Almeida De Freitas	Embraer
Jean-Louis Camus	Esterel Technologies
Richard Canis	EASA
Yann Carlier	DGAC
Luc Casagrande	EADS Apsys
Mark Chapman	Hamilton Sundstrand
Scott Chapman	FAA
Jim Chelini	Verocel
Daniel Chevallier	Thales
John Chilenski	Boeing Company
Subbiah Chockalingam	HCL Technologies
Chris Clark	Sysgo
Darren Cofer	Rockwell Collins
Keith Coffman	Goodrich
John Coleman	Dawson Consulting
Cyrille Comar	AdaCore
Ray Conrad	Lockheed Martin
Mirko Conrad	The MathWorks, Inc.
Nathalie Corbovianu	DGAC
Ana Costanti	Embraer
Dewi Daniels	Verocel
Eric Danielson	Rockwell Collins
Henri De La Vallée Poussin	SABCA
Michael Deitz	Gentex Corporation
Jean-Luc Delamaide	EASA
Hervé Delseny	Airbus
Patrick Desbiens	Transport Canada
Mike DeWalt	Certification Services, Inc./FAA
Mansur Dewshi	Ultra Electronics Controls
Bernard Dion	Esterel Technologies
Antonio Jose Vitorio Domiciano	Embraer
Kurt Doppelbauer	TTTech
Cheryl Dorsey	Digital Flight
Rick Dorsey	Digital Flight
John Doughty	Garmin International
Vincent Dovydaitis III	Foliage Software Systems, Inc.
Georges Duchein	DGA
Branimir Dulic	Transport Canada
Gilles Dulon	SAGEM DS Safran Group
Paul Dunn	Northrop Grumman Corporation
Andrew Eaton	UK CAA
Brian Eckmann	Universal Avionics Systems Corporation
Vladimir Eliseev	Sukhoi Civil Aircraft Company (SCAC)
Andrew Elliott	Design Assurance
Mike Elliott	Boeing Company
Joao Esteves	Critical Software
Rowland Evans	Pratt & Whitney Canada
Louis Fabre	Eurocopter

Martin Fassl	Siemens AG
Michael Fee	Aero Engine Controls (Rolls-Royce)
Tom Ferrell	Ferrell and Associates Consulting
Uma Ferrell	Ferrell and Associates Consulting
Lou Fisk	GE Aviation
Ade Fountain	Penny and Giles
Claude Fournier	Liebherr
Pierre Francine	Thales
Timothy Frey	Honeywell
Stephen J. Fridrick	GE Aviation
Leonard Fulcher	TTTech
Randall Fulton	Seaweed Systems
Francoise Gachet	Dassault-Aviation
Victor Galushkin	GosNIIAS
Marty Gasiorowski	Worldwide Certification Services
Stephanie Gaudan	Thales
Jean-Louis Gebel	Airbus
Dries Geldof	BARCO
Dimitri Gianesini	Airbus
Jim Gibbons	Boeing Company
Dara Gibson	FAA
Greg Gicca	AdaCore
Steven Gitelis	Lumina Engineering
Ian Glazebrook	WS Atkins
Santiago Golmayo	GMV SA
Ben Gorry	British Aerospace Systems
Florian Gouleau	DGA Techniques Aéronautiques
Olivier Graff	Intertechnique - Zodiac
Russell DeLoy Graham	Garmin International
Robert Green	BAE Systems
Mark Grindle	Systems Enginuity
Peter Grossinger	Pilatus Aircraft
Mark Gulick	Solers, Inc.
Pierre Guyot	Dassault Aviation
Ibrahim Habli	University of York
Ross Hannan	Sigma Associates (Aerospace) Limited
Christopher H. Hansen	Rockwell Collins
Wue Hao Wen	Civil Aviation Administration of China (CAAC)
Keith Harrison	HVR Consulting Services Ltd
Bjorn Hasselqvist	Saab AB
Kevin Hathaway	Aero Engine Controls (Goodrich)
David Hawken	NATS
Kelly Hayhurst	NASA
Peter Heath	Securaplane Technologies
Myron Hecht	Aerospace Corporation
Don Heck	Boeing Company
Peter Heller	Airbus Operations GmbH
Barry Hendrix	Lockheed Martin
Michael Hennell	LDRA
Michael Herring	Rockwell Collins
Ruth Hirt	FAA

Kent Hollinger	Mitre Corp
C. Michael Holloway	NASA
Ian Hopkins	Aero Engine Controls (Rolls-Royce)
Gary Horan	FAA
Chris Hote	PolySpace Inc.
Susan Houston	FAA
James Hummell	Embedded Plus
Dr. James J. Hunt	aicas
Rebecca L. Hunt	Boeing Company
Stuart Hutchesson	Aero Engine Controls (Rolls-Royce)
Rex Hyde	Moog Inc. Aircraft Group
Mario Iacobelli	Mannarino Systems
Melissa Isaacs	FAA
Vladimir Istomin	Sukhoi Civil Aircraft Company (SCAC)
Stephen A. Jacklin	NASA
Matt Jaffe	Embry-Riddle Aeronautical University
Marek Jaglarz	Pilatus Aircraft
Myles Jalalian	FAA
Merlin James	Garmin International
Tomas Jansson	Saab AB
Eric Jenn	Thales
Lars Johannknecht	EADS
Rikard Johansson	Saab AB
John Jorgensen	Universal Avionics Systems
Jeffrey Joyce	Critical Systems Labs
Chris Karis	Ensco
Gene Kelly	CertTech
Anne-Cécile Kerbrat	Aeroconseil
Randy Key	FAA
Charles W. Kilgore II	FAA
Wayne King	Honeywell
Daniel Kinney	Boeing Company
Judith Klein	Lockheed Martin
Joachim Klichert	Diehl Avionik Systeme
Jeff Knickerbocker	Sunrise Certification & Consulting, Inc.
John Knight	University of Virginia
Rainer Kollner	Verocel
Andrew Kornecki	Embry-Riddle Aeronautical University
Igor Koverninskiy	Gos NIIAS
Jim Krodel	Pratt & Whitney
Paramesh Kunda	Pratt & Whitney Canada
Sylvie Lacabanne	AIRBUS
Gérard Ladier	Airbus/Aerospace Valley
Ron Lambalot	Boeing Company
Boris Langer	Diehl Aerospace
Susanne Lanzerstorfer	APAC GesmbH
Gilles Laplane	SAGEM DS Safran Group
Jeanne Larsen	Hamilton Sundstrand
Emmanuel Ledinot	Dassault Aviation
Stephane Leriche	Thales
Hong Leung	Bell Helicopter Textron

John Lewis	FAA
John Li	Thales
Mark Lillis	Goodrich GPECS
Barbara Lingberg	FAA
Pierre Lionne	EADS APSYS
Hoyt Lougee	Foliage Software Systems
Howard Lowe	GE Aviation
Hauke Luethje	NewTec GmbH
Jonathan Lynch	Honeywell
Françoise Magliozzi	Atos Origin
Veronique Magnier	EASA
Kristine Maine	Aerospace Corporation
Didier Malescot	DSNA/DTI
Varun Malik	Hamilton Sundstrand
Patrick Mana	EUROCONTROL
Joseph Mangan	Coanda Aerospace Software
Ghilaine Martinez	DGA Techniques Aéronautiques
Steven C. Martz	Garmin International
Peter Matthews	Independent Consultant
Frank McCormick	Certification Services Inc
Scott McCoy	Harris Corporation
Thomas McHugh	FAA
William McMinn	Lockheed Martin
Josh McNeil	US Army AMCOM SED
Kevin Meier	Cessna Aircraft Company
Amanda Melles	Bombardier
Marc Meltzer	Belcan Engineering
Steven Miller	Rockwell Collins
Gregory Millican	Honeywell
John Minihan	Resource Group
Martin Momberg	Cassidian Air Systems
Pippa Moore	UK CAA
Emilio Mora-Castro	EASA
Endrich Moritz	Technical University
Robert Morris	CDL Systems Ltd.
Allan Terry Morris	NASA
Steve Morton	TBV Associates
Nadir Mostefat	Mannarino Systems
Fred B. Moyer	Rockwell Collins
Robert D. Mumme	Embedded Plus Engineering
Arun Murthi	AERO&SPACE USA
Armen Nahapetian	Teledyne Controls
Gerry Ngu	EASA
Elisabeth Nguyen	Aerospace Corporation
Robert Noel	Mitre Corp
Sven Nordhoff	SQS AG
Paula Obeid	Embedded Plus Engineering
Eric Oberle	Becker Avionics
Brenda Ocker	FAA
Torsten Ostermeier	Bundeswehr
Frederic Painchaud	Defence Research and Development Canada

Sean Parkinson	Resource Group
Dennis Patrick Penza	AVISTA
Jean-Phillipe Perrot	Turbomeca
Robin Perry	GE Aviation
David Petesch	Hamilton Sundstrand
John Philbin	Northrop Grumman Integrated Systems
Christophe Piala	Thales Avionics
Cyril Picard	EADS APSYS
Francine Pierre	Thales Avionics
Patrick Pierre	Thales Avionics
Gerald Pilj	FAA
Benoit Pinta	Intertechnique - Zodiac
Mo Piper	Boeing Company
Andreas Pistek	ITK Engineering AG
Laurent Plateaux	DGA
Laurent Pomies	Independent Consultant
Jennifer Popovich	Jeppesen Inc.
Clifford Porter	Aircell LLC
Frédéric Pothon	ACG Solutions
Bill Potter	The MathWorks Inc
Sunil Prasad	HCL Technologies, Chennai, India
Paul J. Prisaznuk	ARINC-AEEC
Naim Rahmani	Transport Canada
Angela Rapaccini	ENAC
Lucas Redding	Silver-Atena
David Redman	Aerospace Vehicle Systems Institute (AVSI)
Tammy Reeve	Patmos Engineering Services, Inc.
Guy Renault	SAGEM DS Safran Group
Leanna Rierson	Digital Safety Consulting
George Romanski	Verocel
Cyrille Rosay	EASA
Edward Rosenbloom	Kollsman, Inc
Tom Roth	Airborne Software Certification Consulting
Jamel Rouahi	CEAT
Marielle Roux	Rockwell Collins France
Rudy Ruana	RTCA, Inc.
Benedito Massayuki Sakugawa	ANAC Brazil
Almudena Sanchez	GMV SA
Vdot Santhanam	Boeing Company
Laurence Scales	Thales
Deidre Schilling	Hamilton Sundstrand
Ernst Schmidt	Bundeswehr
Peter Schmitt	Universität Karlsruhe
Dr. Achim Schoenhoff	EADS Military Aircraft
Martin Schwarz	TT Technologies
Gabriel Scolan	SAGEM DS Safran Group
Christel Seguin	ONERA
Beatrice Sereno	Teuchos SAFRAN
Phillip L. Shaffer	GE Aviation
Jagdish Shah	Parker
Vadim Shapiro	TetraTech/AMT

Jean François Sicard	DGA Techniques Aéronautiques
Marten Sjoestedt	Saab AB
Peter Skaves	FAA
Greg Slater	Rockwell Collins
Claudine Sokoloff	Atos Origin
Marge Sonnek	Honeywell
Guillaume Soudain	EASA
Roger Souter	FAA
Robin L. Sova	FAA
Richard Spencer	FAA
Thomas Sperling	The Mathworks
William StClair	LDRA
Roland Stalford	Galileo Industries Spa
Jerry Stamatopoulous	Aircell LLC
Tom Starnes	Cessna Aircraft Company
Jim Stewart	NATS
Tim Stockton	Certon
Victor Strachan	Northrop-Grumman
John Strasburger	FAA
Margarita Strelnikova	Sukhoi Civil Aircraft Company (SCAC)
Ronald Stroup	FAA
Will Struck	FAA
Wladimir Terzic	SAGEM DS Safran Group
Wolfgang Theurer	C-S SI
Joel Thornton	Tier5 Inc
Mikael Thorvaldsson	KnowIT Technowledge
Bozena Brygida Thrower	Hamilton Sundstrand
Christophe Travers	Dassault Aviation
Fay Trowbridge	Honeywell
Nick Tudor	Tudor Associates
Silpa Uppalapati	FAA
Marie-Line Valentin	Airbus
Jozef Van Baal	Civil Aviation Authorities Netherlands
John Van Leeuwen	Sikorsky Aircraft
Aulis Viik	NAV Canada
Bertrand Voisin	Dassault Aviation
Katherine Volk	L-3 Communications
Dennis Wallace	FAA
Andy Wallington	Bell Helicopter
Yunming Wang	Esterel Technologies
Don Ward	AVSI
Steve Ward	Rockwell Collins
Patricia Warner	Software Engineering
Michael Warren	Rockwell Collins
Rob Weaver	NATS
Yu Wei	CAA China
Terri Weinstein	Parker Hannifin
Marcus Weiskirchner	EADS Military Aircraft
Daniel Weisz	Sandel Avionics, Inc.
Rich Wendlandt	Quantum3D
Michael Whalen	Rockwell Collins

Paul Whiston	High Integrity Solutions Ltd
Todd R. White	L-3 Communications/Qualtech
Virginie Wiels	ONERA
ElRoy Wiens	Cessna Aircraft Company
Terrance Williamson	Jeppesen Inc.
Graham Wisdom	BAE Systems
Patricia Wojnarowski	Boeing Commercial Airplanes
Joerg Wolfrum	Diehl Aerospace
Kurt Woodham	NASA
Cai Yong	CAAC (Civil Aviation Administration of China)
Edward Yoon	Curtiss-Wright Controls, Inc
Robert Young	Rolls-Royce
William Yu	CAAC China
Erhan Yuceer	Savunma Teknolojileri Muhendislik ve Ticaret
Uli Zanker	Liebherr

APPENDIX C

INDEX OF KEYWORDS

APPENDIX D

CORRELATION BETWEEN ED-12C, ED-109A, AND ED-94C

ED-12C Section	ED-109A Section	ED-94C Product	ED-94C Section
1.2	1.2	FAQ #59	3.59
1.3	1.3	FAQ #22	3.22
1.4	1.4	FAQ #20, FAQ #32, FAQ #84	3.20, 3.32, 3.84
2	2	FAQ #24, Rationale	3.24, 5.2
2.1	2.1	DP #14	4.14
2.2.1	2.2.1	FAQ #37, DP #14	3.37, 4.14
2.2.2	2.2.2	FAQ #37, DP #9, DP #14	3.37, 4.9, 4.14
2.2.3	2.2.3	FAQ #37	3.37
2.3	2.3	FAQ #23, DP #14	3.23, 4.14
2.3.4	2.3.4	DP #20	4.20
2.4	2.4	FAQ #25, DP #14	3.25, 4.14
2.4.1	2.4.1	DP #14	4.14
2.5.1	2.6.1	DP #14, DP #20	4.14, 4.20
2.5.4	2.6.4	FAQ #4, FAQ #8	3.4, 3.8
2.5.5	2.6.5	FAQ #5, FAQ #29, DP #20	3.5, 3.29, 4.20
3	3	DP #6, Rationale	4.6, 5.3
3.2	3.2	FAQ #68	3.68
3.3	3.3	DP #6	4.6
4	4	Rationale	5.4
4.1	4.1	DP #6	4.6
4.2	4.2	FAQ #4, FAQ #8, DP #6, DP #20	3.4, 3.8, 4.6, 4.20
4.3	4.3	DP #6	4.6
4.4.1	4.4.1	FAQ #83	3.83
4.4.2	4.4.2	FAQ #83	3.83
4.5	4.5	FAQ #32, FAQ #42, DP #21	3.32, 3.42, 4.21
4.6	4.6	DP #19	4.19
5	5	FAQ #35, FAQ #36, FAQ #81, FAQ #82, Rationale	3.35, 3.36, 3.81, 3.82, 5.5
5.1.1	5.1.1	FAQ #37	3.37
5.1.2	5.1.2	FAQ #37, DP #6, DP #20	3.37, 4.6, 4.20
5.2	5.2	FAQ #82	3.82
5.2.1	5.2.1	FAQ #37	3.37
5.2.2	5.2.2	FAQ #37, FAQ #69, DP #6, DP #14	3.37, 3.69, 4.6, 4.14
5.2.3	5.2.3	FAQ #7, DP #14	3.7, 4.14
5.2.4	5.2.4	FAQ #4, FAQ #8, FAQ #70	3.4, 3.8, 3.70
5.3.2	5.3.2	DP #6	4.6
5.4.2	5.4.2	DP #6	4.6
5.5	5.5	FAQ #35	3.35
6	6	FAQ #35, FAQ #75, Rationale	3.35, 3.75, 5.6
6.2	6.2	FAQ #72, DP #19	3.72, 4.19
6.3	6.3	FAQ #73	3.73
6.3.1	6.3.1	FAQ #73, DP #20	3.73, 4.20
6.3.2	6.3.2	FAQ #73	3.73
6.3.3	6.3.3	FAQ #39, FAQ #67, DP #14	3.39, 3.67, 4.14
6.3.4	6.3.4	FAQ #39, FAQ #40, FAQ #67, FAQ #73, FAQ #80, FAQ #82, DP #16, DP #17	3.39, 3.40, 3.67, 3.73, 3.80, 3.82, 4.16, 4.17
6.4	6.4	FAQ #9	3.9
6.4.1	6.4.1	FAQ #9	3.9
6.4.2.1	6.4.2.1	FAQ #78	3.78

ED-12C Section	ED-109A Section	ED-94C Product	ED-94C Section
6.4.2.2	6.4.2.2	FAQ #73	3.73
6.4.3	6.4.3	FAQ #9, FAQ #73, DP #14	3.9, 3.73, 4.14
6.4.4	6.4.4	FAQ #67, DP #8	3.67, 4.8
6.4.4.2	6.4.4.2	FAQ #42, FAQ #43, FAQ #44, FAQ #74, FAQ #82, DP #8, DP #12, DP #13	3.42, 3.43, 3.44, 3.74, 3.82, 4.8, 4.12, 4.13
6.4.4.3	6.4.4.3	FAQ #4, FAQ #8, FAQ #43, FAQ #44, FAQ #82, DP #12	3.4, 3.8, 3.43, 3.44, 3.82, 4.12
6.5	6.5	FAQ #35	3.35
6.6	6.6	DP #20	4.20
7	7	Rationale	5.7
7.2.1	7.2.1	DP #9	4.9
7.2.2	7.2.2	FAQ #17, FAQ #46	3.17, 3.46
7.2.3	7.2.3	FAQ #72, DP #9	3.72, 4.9
7.2.4	7.2.4	FAQ #17, FAQ #72	3.17, 3.72
7.2.5	7.2.5	FAQ #72, DP #9	3.72, 4.9
7.2.6	7.2.6	DP #9	4.9
7.3	7.3	FAQ #13, FAQ #14, FAQ #55	3.13, 3.14, 3.55
7.4	7.4	DP #20	4.20
8	8	FAQ #75, Rationale	3.75, 5.8
8.1	8.1	DP #6	4.6
8.2	8.2	DP #6, DP #19	4.6
8.3	8.3	FAQ #47, DP #9	3.47, 4.9
9	9	FAQ #48, Rationale	3.48, 5.9
9.1	9.1	FAQ #22	3.22
9.3	9.3	FAQ #50, FAQ #60	3.50, 3.60
10	10	FAQ #48, FAQ #79, Rationale	3.48, 3.79, 5.10
10.3	10.3	FAQ #52	3.52
11	11	FAQ #22, FAQ #50, FAQ #54, FAQ #55, FAQ #56, Rationale	3.22, 3.50, 3.54, 3.55, 3.56, 5.11
11.1	11.1	FAQ #57, DP #14, DP #21	3.57, 4.14, 4.21
11.2	11.2	DP #6	4.6
11.3	11.3	FAQ #56, FAQ #58, DP #6, DP #14, DP #19	3.56, 3.58, 4.6, 4.14, 4.19
11.4	11.4	DP #6, DP #9	4.6, 4.9
11.5	11.5	DP #6, DP #19	4.6, 4.19
11.6	11.6	FAQ #37	3.37
11.8	11.8	FAQ #75	3.75
11.9	11.9	FAQ #73, FAQ #77, FAQ #81, DP #14	3.73, 3.77, 3.81, 4.14
11.10	11.10	FAQ #73, FAQ #81, DP #14	3.73, 3.81, 4.14
11.13	11.13	FAQ #56	3.56
11.14	11.14	FAQ #56	3.56
11.16	11.16	DP #20	4.20
11.17	11.17	FAQ #72, DP #9	3.72, 4.9
11.20	11.20	FAQ #73, DP #9, DP #14	3.73, 4.9, 4.14
12	12	FAQ #57, Rationale	3.57, 5.12
12.1	12.1	FAQ #16, FAQ #17, FAQ #68, DP #9, DP #10	3.16, 3.17, 3.68, 4.9, 4.10
12.1.1	12.1.1	FAQ #18, FAQ #58, FAQ #60, FAQ #73, DP #15	3.18, 3.58, 3.60, 3.73, 4.15
12.1.2	12.1.2	FAQ #18, FAQ #76	3.18, 3.76
12.1.3	12.1.3	FAQ #76, DP #15	3.76, 4.15
12.1.5	12.1.5	FAQ #76	3.76
12.2	12.2	FAQ #42, FAQ #59, FAQ #62, Rationale	3.42, 3.59, 3.62, 5.13.1

ED-12C Section	ED-109A Section	ED-94C Product	ED-94C Section
12.2.1	12.2.1	FAQ #65	3.65
12.3	12.3	FAQ #75, DP #5	3.75, 4.5
12.3.1	12.3.1	FAQ #63	3.63
12.3.2	12.3.2	FAQ #64, FAQ #65, DP #14	3.64, 3.65, 4.14
12.3.2.1	12.3.2.1	DP #19	4.19
12.3.2.3	12.3.2.3	DP #12	4.12
12.3.2.4	12.3.2.4	FAQ #65	3.65
12.3.2.5	12.3.2.5	FAQ #65	3.65
12.3.3	12.3.3	FAQ #23	3.23
12.3.4	12.3.4	DP #4, DP #5, DP #10, DP #18	4.4, 4.5, 4.10, 4.18
12.3.4.3	12.3.4.3	DP #9	4.9
N/A	12.4	DP #10	4.10
N/A	12.4.1.3	DP #6	4.6
N/A	12.4.2	DP #14	4.14
N/A	12.4.3	DP #6	4.6
N/A	12.4.4	DP #14	4.14
N/A	12.4.5.2	DP #14	4.14
N/A	12.4.6	DP #9	4.9
N/A	12.4.9	DP #9	4.9
N/A	12.4.11	FAQ #39, DP #9, DP #14	3.39, 4.9, 4.14
Annex A	Annex A	FAQ #14, FAQ #22, FAQ #35, FAQ #43, FAQ #55, FAQ #67, FAQ #73, FAQ #74, FAQ #75, FAQ #84, DP #5, DP #6, DP #8, DP #12, DP #13, DP #14, Rationale	3.14, 3.22, 3.35, 3.43, 3.55, 3.67, 3.73, 3.74, 3.75, 3.84, 4.5, 4.6, 4.8, 4.12, 4.13, 4.14, 5.5.1, 5.6.1, 5.6.2, 5.6.3
Annex A Table A-1	Annex A Table A-1	DP #6, Rationale	4.6, 5.4
Annex A Table A-2	Annex A Table A-2	FAQ #84, Rationale	3.84, 5.5.1
Annex A Table A-3	Annex A Table A-3	Rationale	5.6.1
Annex A Table A-4	Annex A Table A-4	FAQ #84, Rationale	3.84, 5.6.1
Annex A Table A-5	Annex A Table A-5	FAQ #73, FAQ #75, Rationale	3.73, 3.75, 5.6.1
Annex A Table A-6	Annex A Table A-6	Rationale	5.6.2
Annex A Table A-7	Annex A Table A-7	FAQ #43, FAQ #67, DP #8, DP #12, DP #13, Rationale	3.43, 3.67, 4.8, 4.12, 4.13, 5.6.3
Annex A Table A-9	Annex A Table A-9	DP #6, Rationale	4.6, 5.8
Annex B	Annex B	FAQ #20, FAQ #21, FAQ #36, FAQ #39, FAQ #47, FAQ #67, FAQ #75, DP #5, DP #6, DP #19	3.20, 3.21, 3.36, 3.39, 3.47, 3.67, 3.75, 4.5, 4.6, 4.19
Figure 6-1	Figure 6-1	FAQ #43	3.43