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

SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION

ED-12C

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

SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION

ED-12C

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

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

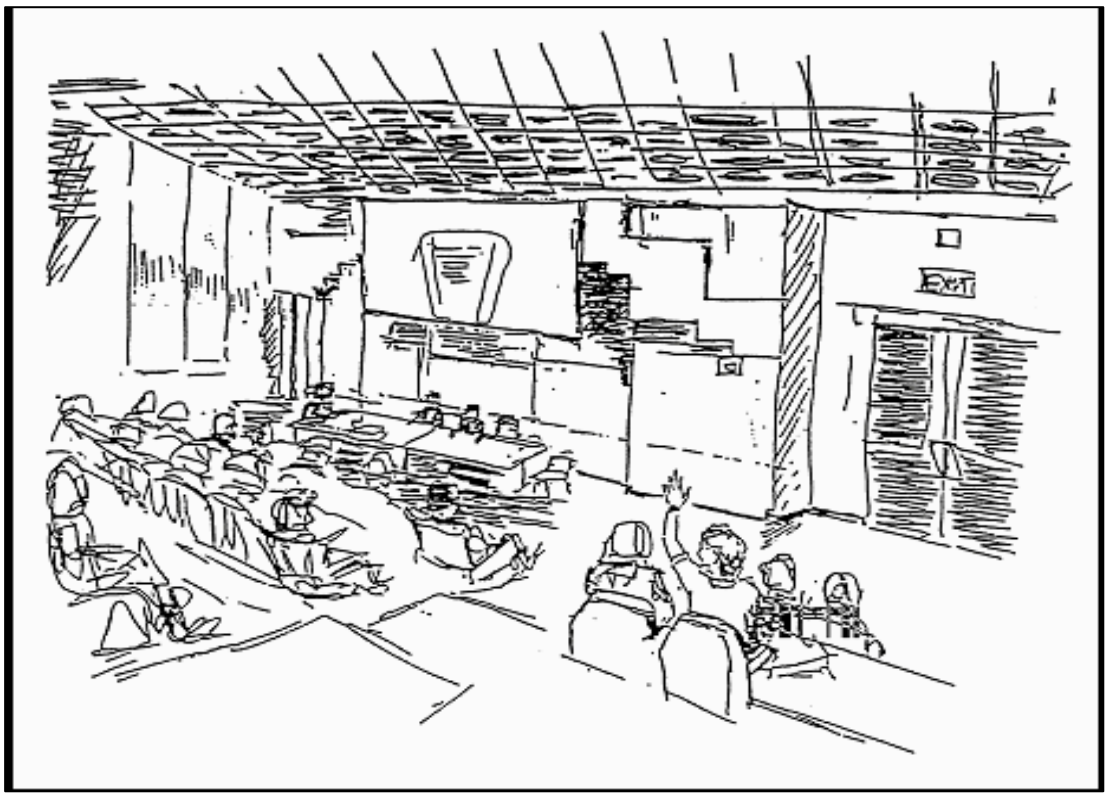

Illustration provided by Pat Neilan, UK CAA

CONSENSUS n. Collective opinion or concord; general agreement or accord. [Latin, from consentire, to agree]

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The rapid increase in the use of software in airborne systems and equipment used on aircraft and engines in the early 1980s resulted in a need for industry-accepted guidance for satisfying airworthiness requirements. ED-12, "Software Considerations in Airborne Systems and Equipment Certification", was written to satisfy this need.

This document, now revised in the light of experience, provides the aviation community with guidance for determining, in a consistent manner and with an acceptable level of confidence, that the software aspects of airborne systems and equipment comply with airworthiness requirements. As software use increases, technology evolves, and experience is gained in the application of this document, this document will be reviewed and revised. Appendix A provides the background of this document.

1.1 PURPOSE

The purpose of this document is to provide guidance 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. This guidance includes:

- Objectives for software life cycle processes.
- Activities that provide a means for satisfying those objectives.
- Descriptions of the evidence in the form of software life cycle data that indicate that the objectives have been satisfied.
- Variations in the objectives, independence, software life cycle data, and control categories by software level.
- Additional considerations (for example, previously developed software) that are applicable to certain applications.
- Definition of terms provided in the glossary.

In addition to guidance, supporting information is provided to assist the reader’s understanding.

1.2 SCOPE

This document discusses those aspects of certification that pertain to the production of software for airborne systems and equipment used on aircraft, engines, propellers and, by region, auxiliary power units. In discussing those aspects, the system life cycle and its relationship with the software life cycle is described to aid in the understanding of the certification process. A complete description of the system life cycle processes, including the system safety assessment and validation processes, or the certification process is not intended.

The guidance contained in this document does not define or imply the level of involvement of a certification authority in a certification process. To understand certification authority involvement, the applicant should refer to applicable regulations and guidance material issued by the relevant certification authority.

Since certification issues are discussed only in relation to the software life cycle, the operational aspects of the resulting software are not discussed. For example, the certification aspects of user-modifiable data are beyond the scope of this document.

This document does not attempt to define firmware. Firmware should be classified as hardware or software and addressed by the applicable processes. This document assumes that during the system definition, functions have been allocated to either software or hardware. Other documents exist that provide guidance for development assurance for functions that are allocated to implementation in hardware. This document provides guidance for functions that are allocated to software.

NOTE: *This allows an efficient method of implementation and development assurance to be determined at the time the system is specified and functions allocated. All parties should agree with this system decision at the time the allocation is made.*

Matters concerning the structure of the applicant's organization, the commercial relationships between the applicant and its suppliers, and personnel qualification criteria are beyond the scope of this document.

1.3 RELATIONSHIP TO OTHER DOCUMENTS

In addition to the airworthiness requirements, various national and international standards for software are available. In some communities, compliance with these standards may be required. However, it is outside the scope of this document to invoke specific national or international standards, or to propose a means by which these standards might be used as an alternative or in addition to this document.

It is recognized that projects may be obliged, through contract or other means, to comply with additional standards as applied by, for example, the engine or aircraft manufacturer. Such standards may be derived from general standards produced or adopted by the manufacturer for its activities. Such standards should be considered by the planning process and considered, as appropriate, when applying supplier oversight.

1.4 HOW TO USE THIS DOCUMENT

The following points should be noted when using this document:

a. This document is intended to be used by the international aviation community. To aid such use, references to specific national regulations and procedures are minimized. Instead, generic terms are used. For example, the term "certification authority" is used to denote the organization or person granting approval on behalf of the country responsible for certification of the product (for example, an aircraft, engine). Where a second country or group of countries validates or participates in this certification, this document may be used with due recognition given to bilateral agreements or memoranda of understanding between the countries involved.

b. This document recognizes that the guidance herein is not mandated by law, but represents a consensus of the aviation community. It also recognizes that alternative methods to the methods described herein may be available to the applicant. For these reasons, the use of words such as "shall" and "must" is avoided.

c. If an applicant adopts this document as a means of compliance, the applicant should satisfy all applicable objectives. This document should apply to the applicant and any of its suppliers, who are involved with any of the software life cycle processes or the outputs of those processes described herein. The applicant is responsible for oversight of all of its suppliers.

d. The applicant should plan a set of activities that satisfy the objectives. This document describes activities for achieving those objectives. The applicant may plan and, subject to the approval of the certification authority, adopt alternative activities to those described in this document. The applicant may also plan and conduct additional activities that are determined to be necessary.

e. The applicant should address any additional considerations in its software plans and standards.

f. The applicant should perform the planned activities and provide evidence as indicated in section 11 to substantiate that the objectives have been satisfied.

g. Explanatory text is included to aid the reader in understanding the topic under discussion. For example, section 2 provides information necessary to understand the interaction between the system life cycle and software life cycle. Similarly, section 3 provides a description of the software life cycle and section 10 an overview of the certification process.

h. Section 11 contains the data generally produced to aid the software aspects of the certification process. The names of the data are denoted in the text by capitalization of the first letter of each word in the name (for example, Source Code).

i. Section 12 discusses additional considerations including guidance for the use of previously developed software, tool qualification, and the use of alternative methods to those described in sections 2 through 11. Section 12 may not apply to every project.

j. Annex A specifies the applicability of the objectives, activities, and software life cycle data for each software level as well as the variation in the independence and control categories for each software level. In order to fully understand the guidance, the full body of this document should be considered.

k. In cases where examples are used to indicate how the guidance might be applied, either graphically or through narrative, the examples are not to be interpreted as the preferred method. In these cases, the examples are considered supporting information.

l. A list of items does not imply the list is all-inclusive.

m. Notes in this document are supporting information used to provide explanatory material, emphasize a point, or draw attention to related items which are not entirely within context.

n. 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”.

o. One or more supplements to this document exist and extend the guidance in this document to a specific technique. Supplements are used in conjunction with this document and may be used in conjunction with one another. Unless alternatives are used (see 1.4.i), if a supplement exists for a specific technique, the supplement should be used to add, delete, or otherwise modify objectives, activities, explanatory text, and software life cycle data in this document to address that technique, as defined appropriately in each supplement. It is the responsibility of the applicant to ensure that the supplement’s use is acceptable to the appropriate certification authority. As part of the software planning process, the applicant should review all potentially relevant supplements and identify those that will be used. The information in supplements should be used with and in the same way as this document. Annex A of each supplement identifies how the objectives of this document are revised relative to the specific technique addressed by the supplement.

p. Compliance is achieved when all applicable objectives have been satisfied by performing all planned activities and capturing the related evidence.

1.5 DOCUMENT OVERVIEW

Figure 1-1 is a pictorial overview of this document's sections and their relationship to each other.

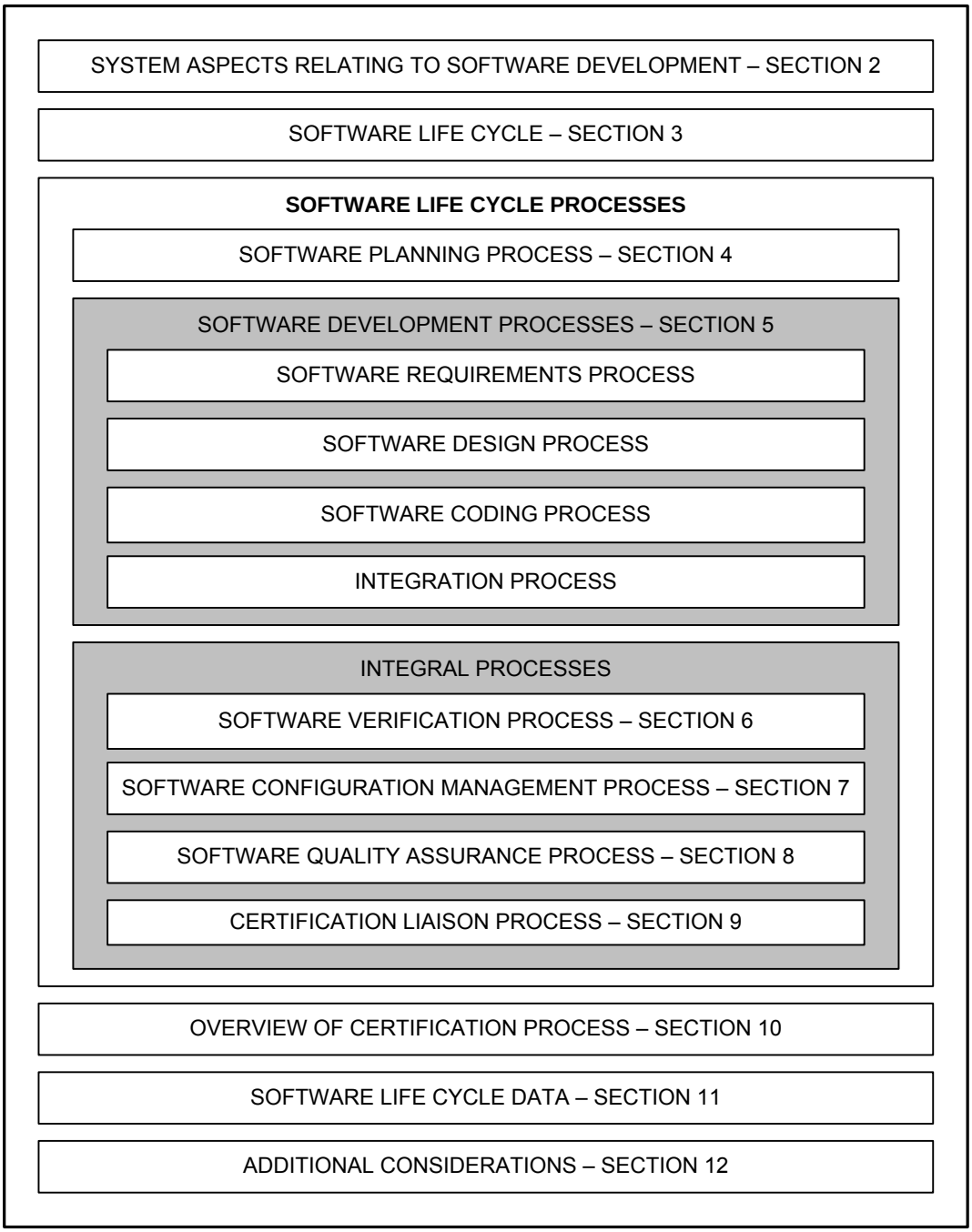

FIGURE 1-1: DOCUMENT OVERVIEW

CHAPTER 2

SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT

This section discusses those aspects of the system life cycle processes necessary to understand the software life cycle processes. System life cycle processes can be found in other industry documents (for example, SAE ARP4754A).

Discussed in this section are:

- System requirements allocation to software (see 2.1).
- The information flow between the system and software life cycle processes and between the software and hardware life cycle processes (see 2.2).
- The system safety assessment process, failure conditions, software level definitions, and software level determination (see 2.3).
- Architectural considerations (see 2.4).
- Software considerations in system life cycle processes (see 2.5).
- System considerations in software life cycle processes (see 2.6).

The term “system” in the context of this document refers to the airborne system and equipment only, not to the wider definition of a system that might include operators, operational procedures, etc.

2.1 SYSTEM REQUIREMENTS ALLOCATION TO SOFTWARE

As part of the system life cycle processes, system requirements are developed from the system operational requirements and other considerations such as safety-related, security, and performance requirements. The safety-related requirements result from the system safety assessment process, and may include functional, integrity, and reliability requirements, as well as design constraints.

The system safety assessment process determines and categorizes the failure conditions of the system. Within the safety assessment process, safety-related requirements are defined to ensure the integrity of the system by specifying the desired immunity from, and system responses to, these failure conditions. These requirements are identified for hardware and software to preclude or limit the effects of faults, and may provide fault detection, fault tolerance, fault removal, and fault avoidance.

The system processes are responsible for the refinement and allocation of system requirements to hardware and/or software as determined by the system architecture.

System requirements allocated to software, including safety-related requirements, are developed and refined into software requirements that are verified by the software verification process activities. These requirements and the associated verification should establish that the software performs its intended functions under any foreseeable operating condition. System requirements allocated to software may include:

a. Functional and operational requirements.

b. Interface requirements.

c. Performance requirements.

d. Safety-related requirements, including safety strategies, design constraints and design methods, such as, partitioning, dissimilarity, redundancy, or safety monitoring. In cases where the system is a component of another system, the requirements and failure conditions for that other system may also form part of the system requirements allocated to software.

e. Security requirements.

f. Maintenance requirements.

g. Certification requirements, including any applicable certification authority regulations, issue papers, etc.

h. Additional requirements needed to aid the system life cycle processes.

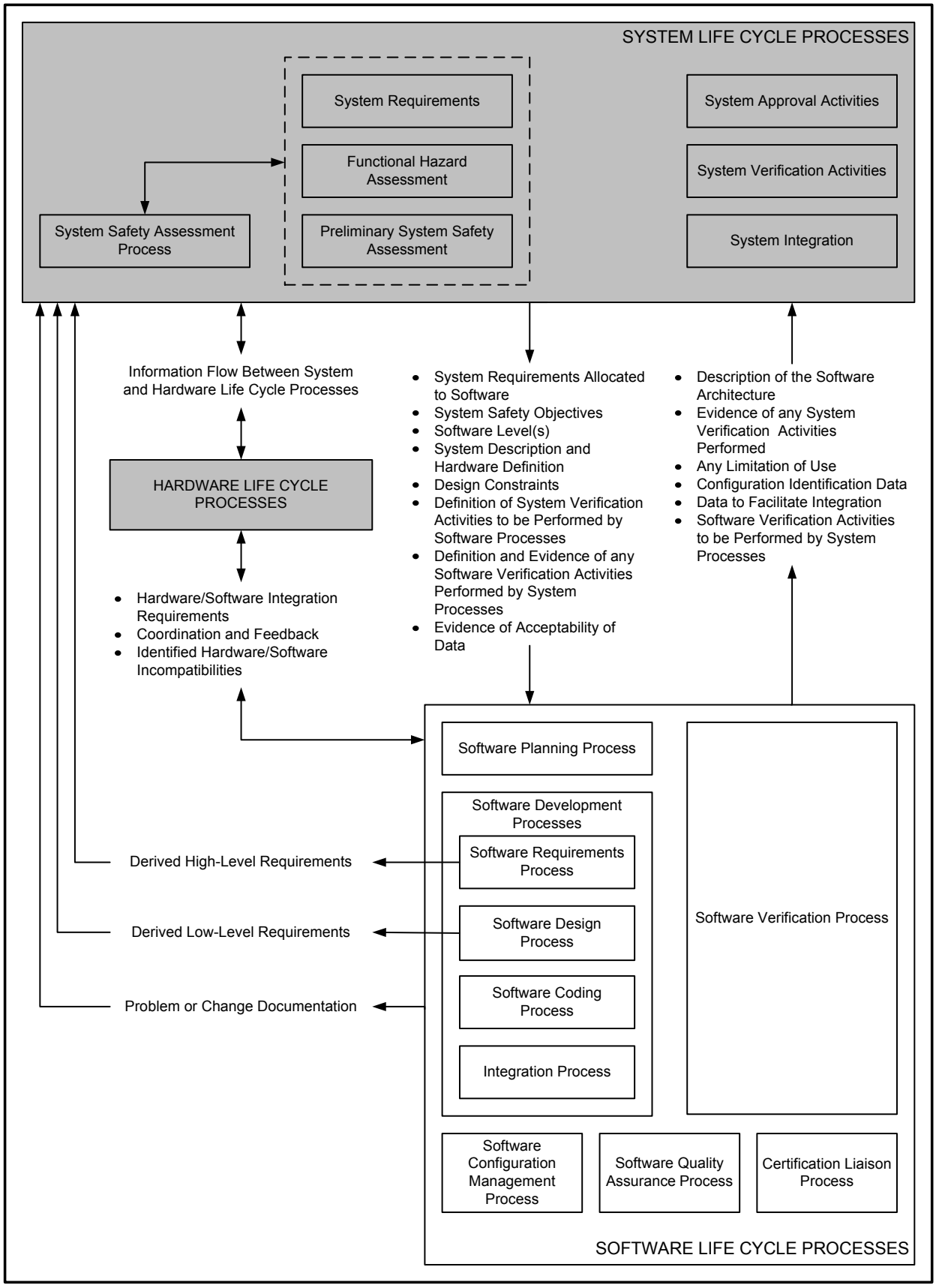

FIGURE 2-1: INFORMATION FLOW BETWEEN SYSTEM AND SOFTWARE LIFE CYCLE PROCESSES

2.2 INFORMATION FLOW BETWEEN SYSTEM AND SOFTWARE LIFE CYCLE PROCESSES

Figure 2-1 is an overview of the information flow between system life cycle processes and the software life cycle processes. This information flow includes the system safety aspects. Due to interdependence of the system safety assessment process and the system design process, the flow of information described in these sections is iterative.

2.2.1 Information Flow from System Processes to Software Processes

The following data is passed to the software life cycle processes by the system processes either as part of the requirements allocation or during the development life cycle:

a. System requirements allocated to software.

b. System safety objectives.

c. Software level for software components and a description of associated failure condition(s), if applicable.

d. System description and hardware definition.

e. Design constraints, including external interfaces, partitioning requirements, etc.

f. Details of any system activities proposed to be performed as part of the software life cycle. Note that system requirement validation is not usually part of the software life cycle processes. The system life cycle processes are responsible for assuring any system activities proposed to be performed as part of the software life cycle.

g. Evidence of the acceptability, or otherwise, of any data provided by the software processes to the system processes on which any activity has been conducted by the system processes. Examples of such activity are the system processes' evaluations of:

    1. Derived requirements provided by the software processes to determine if there is any impact on the system safety assessment and system requirements.

    2. Issues raised by the software processes with respect to the clarification or correction of system requirements allocated to software.

h. Evidence of software verification activities performed by the system life cycle processes, if any.

Any evidence provided by the system processes (see 2.2.1.f and 2.2.1.g) should be considered by the software processes to be Software Verification Results (see 11.14).

2.2.2 Information Flow from Software Processes to System Processes

The software life cycle processes analyze the system requirements allocated to software as part of the software requirements process. If such an analysis identifies any system requirements as inadequate or incorrect, the software life cycle processes should capture the issues and refer them to the system processes for resolution. Furthermore, as the software design and implementation evolves, details are added and modifications made that may affect system safety assessment and system requirements.

To aid the evaluation of the evolving design and changes to the design, the software life cycle processes should make data available to the system processes including the system safety assessment process. This data will facilitate analyses and evaluations to establish the effect on the system safety assessment and system requirements. It may be advantageous for such analyses and evaluations to be performed jointly by the systems and software processes. Such data includes:

a. Details of derived requirements created during the software life cycle processes.

b. A description of the software architecture, including software partitioning.

c. Evidence of system activities performed by the software life cycle processes, if any.

d. Problem or change documentation, including problems identified in the system requirements allocated to software and identified incompatibilities between the hardware and the software.

e. Any limitations of use.

f. Configuration identification and any configuration status constraints.

g. Performance, timing, and accuracy characteristics.

h. Data to facilitate integration of the software into the system.

i. Details of software verification activities proposed to be performed during system verification, if any.

2.2.3 Information Flow between Software Processes and Hardware Processes

Data is passed between the software life cycle process and the hardware life cycle process either as part of the system requirements allocation or during the development life cycles. Such data includes:

a. All requirements, including derived requirements, needed for hardware/software integration, such as definition of protocols, timing constraints, and addressing schemes for the interface between hardware and software.

b. Instances where hardware and software verification activities require coordination.

c. Identified incompatibilities between the hardware and the software.

2.3 SYSTEM SAFETY ASSESSMENT PROCESS AND SOFTWARE LEVEL

This section provides a brief introduction to how the software level for software components is determined and how architectural considerations may influence the allocation of a software level. It is not the intent of this document to prescribe how these activities are performed; this has to be established and performed as part of the system life cycle processes.

The software level of a software component is based upon the contribution of software to potential failure conditions as determined by the system safety assessment process by establishing how an error in a software component relates to the system failure condition(s) and the severity of that failure condition(s). The software level establishes the rigor necessary to demonstrate compliance with this document.

Development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software reliability rates based on software levels cannot be used by the system safety assessment process in the same way as hardware failure rates.

Only partitioned software components (see 2.4.1) can be assigned individual software levels by the system safety assessment process. If partitioning between software components cannot be demonstrated, the software components should be viewed as a single component when assigning software levels (that is, all components are assigned the software level associated with the most severe failure condition to which the software can contribute).

The applicant should establish the system safety assessment process to be used based on certification authority guidance. The software level for each software component of the system should then be assigned based on this process and agreed with the certification authorities.

2.3.1 Relationship between Software Errors and Failure Conditions

Figure 2-2 shows a sequence of events in which a software error leads to the failure condition at aircraft level. A software error may be latent and, thus, not immediately produce a failure. This model is intentionally a simple, linear representation. In actual operation, the sequence of events that leads from a software error to a failure condition may be complex and not easily represented by a sequence of events, as shown in Figure 2-2.

It is important to realize that the likelihood that the software contains an error cannot be quantified in the same way as for random hardware failures.

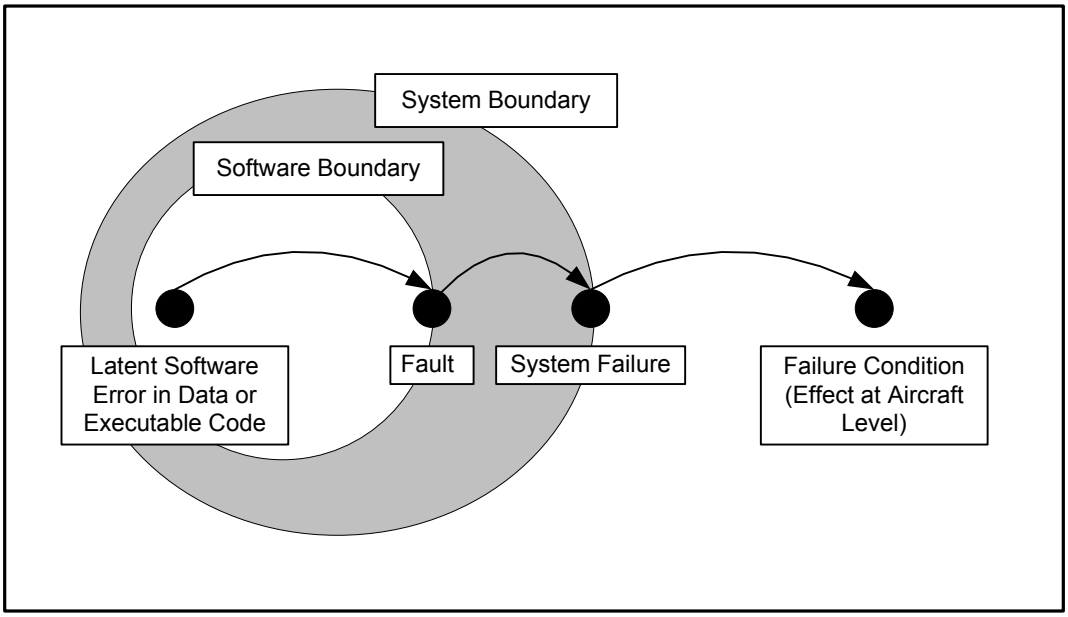

FIGURE 2-2: SEQUENCE OF EVENTS FOR SOFTWARE ERROR LEADING TO A FAILURE CONDITION

Architectural considerations (see 2.4) and/or system external factors may also be considered as part of the system safety assessment process when identifying the failure condition categories and assigning the software level to each software component.

2.3.2 Failure Condition Categorization

For a complete definition of failure condition categories, the applicant should refer to applicable regulations and guidance material issued by the relevant certification authority. The failure condition categories listed in Table 2-1 are valid for large transport aircraft based on established advisory material for the system safety assessment process, and are included to assist in the use of this document.

TABLE 2-1: FAILURE CONDITION CATEGORY DESCRIPTIONS

Category	Description
Catastrophic	Failure Conditions, which would result in multiple fatalities, usually with the loss of the aeroplane.
Hazardous	Failure Conditions, which would reduce the capability of the aeroplane or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be: • A large reduction in safety margins or functional capabilities; • Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely, or • Serious or fatal injury to a relatively small number of the occupants other than the flight crew.
Major	Failure Conditions which would reduce the capability of the aeroplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to the flight crew, or physical distress to passengers or cabin crew, possibly including injuries.
Minor	Failure Conditions which would not significantly reduce aeroplane safety, and which involve crew actions that are well within their capabilities. Minor Failure Conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some physical discomfort to passengers or cabin crew.
No Safety Effect	Failure Conditions that would have no effect on safety; for example, Failure Conditions that would not affect the operational capability of the aeroplane or increase crew workload.

2.3.3 Software Level Definition

This document recognizes five software levels, Level A to Level E. For the example failure condition categories listed in section 2.3.2, the relationships between these software levels and failure conditions are:

a. Level A: Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft.

b. Level B: Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a hazardous failure condition for the aircraft.

c. Level C: Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a major failure condition for the aircraft.

d. Level D: Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft.

e. Level E: Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function with no effect on aircraft operational capability or pilot workload. If a software component is determined to be Level E and this is confirmed by the certification authority, no further guidance contained in this document applies.

The applicant should always consider the appropriate certification guidance and system development considerations for categorizing the failure condition severity and the software level.

2.3.4 Software Level Determination

The system safety assessment process determines the software level(s) appropriate to the software components of a particular system based upon the failure condition which may result from anomalous behavior of the software. The impact of, both loss of function and malfunction should be analyzed. External factors such as adverse environmental conditions as well as architectural strategies (as described in section 2.4) may be considered when identifying the failure condition categories and determining the software level.

NOTE 1: *The applicant may want to consider planned functionality to be added during future developments, as well as potential changes in system requirements allocated to software that may result in a more severe failure condition category and higher software level. It may be desirable to develop the software to a level higher than that determined by the system safety assessment process of the original application, since later development of software life cycle data for substantiating a higher software level application may be difficult.*

NOTE 2: *For airborne systems and equipment mandated by operating regulations, but which do not affect the airworthiness of the aircraft, for example, a flight data recorder, the software level needs to be commensurate with the intended function. In some cases, the software level may be specified in equipment minimum performance standards.*

If the anomalous behavior of a software component contributes to more than one failure condition, then the software component should be assigned the software level associated with the most severe failure condition to which the software can contribute, including combined failure conditions.

2.4 ARCHITECTURAL CONSIDERATIONS

This section provides information on several architectural strategies that may limit the impact of failures, or detect failures and provide acceptable system responses to contain them. These architectural techniques are typically identified during system design and should not be interpreted as the preferred or required solutions.

A serial implementation is one in which multiple software components are used for a system function such that anomalous behavior of any of the components could produce the failure condition. In this implementation, the software components will have the software level associated with the most severe failure condition category of the system function.

In implementations where anomalous behavior of two or more partitioned software components is needed in order to cause the failure condition, this may be taken into consideration by the system safety assessment process when assigning the software level for these software components.

The system safety assessment process needs to establish that sufficient independence exists between software components with respect to both function (that is, high-level requirements) and design (for example, common design elements, languages, and tools).

If partitioning and independence between software components cannot be demonstrated, the software components should be viewed as a single software component when assigning software levels (that is, all components are assigned the software level associated with the most severe failure condition to which the software can contribute).

2.4.1 Partitioning

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. Regardless of the method, the following should be ensured for partitioned software components:

a. A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output (I/O), or data storage areas.

b. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.

c. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components.

d. Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components.

e. Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety.

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.

2.4.2 Multiple-Version Dissimilar Software

Multiple-version dissimilar software is a system design technique that involves producing two or more components of software that provide the same function in a way that may avoid some sources of common errors between the components. Multiple-version dissimilar software is also referred to as multi-version software, multi-version independent software, dissimilar software, N-version programming, or software diversity.

Software life cycle processes completed or activated before dissimilarity is introduced into a development, remain potential error sources. System requirements may specify a hardware configuration that provides for the execution of multiple-version dissimilar software.

The degree of dissimilarity, and hence the degree of protection, is not usually measurable. Probability of loss of system function will increase to the extent that the safety monitoring associated with dissimilar software versions detects actual errors using comparator differences greater than threshold limits. Dissimilar software versions are usually used, therefore, as a means of providing additional protection after the software verification process objectives for the software level, as described in section 6, have been satisfied.

Dissimilar software verification methods may be reduced from those used to verify single version software if it can be shown that the resulting potential loss of system function is acceptable as determined by the system safety assessment process.

Verification of multiple-version dissimilar software is discussed in section 12.3.2.

2.4.3 Safety Monitoring

Safety monitoring is a means of protecting against specific failure conditions by directly monitoring a function for failures that would result in a failure condition. Monitoring functions may be implemented in hardware, software, or a combination of hardware and software.

Through the use of monitoring techniques, the software level of the monitored software may be assigned a software level associated with the loss of its related system function. To allow this assignment, there are three important attributes of the monitor that should be determined:

a. Software level: Safety monitoring software is assigned the software level associated with the most severe failure condition category for the monitored function.

b. System fault coverage: Assessment of the system fault coverage of a monitor ensures that the monitor's design and implementation are such that the faults which it is intended to detect will be detected under all necessary conditions.

c. Independence of function and monitor: The monitor and protective mechanism are not rendered inoperative by the same failure that causes the failure condition.

2.5 SOFTWARE CONSIDERATIONS IN SYSTEM LIFE CYCLE PROCESSES

This section provides an overview of those software-related issues (not necessarily mutually exclusive) that should be considered, as appropriate, by the system life cycle processes:

a. Parameter data items.

b. User-modifiable software.

c. Commercial-Off-The-Shelf (COTS) software.

d. Option-selectable software.

e. Field-loadable software.

f. Software considerations in system verification.

2.5.1 Parameter Data Items

Software consists of Executable Object Code and/or data, and can comprise one or more configuration items. A data set that influences the behavior of the software without modifying the Executable Object Code and is managed as a separate configuration item is called a parameter data item. That is, when discussing parameter data items and Executable Object Code, it is implied that the parameter data items are not part of the Executable Object Code.

A parameter data item comprises a structure of individual elements where each element can be assigned a single value. Each element has attributes such as type, range, or set of allowed values.

Examples of parameter data items include configuration tables and databases but not aeronautical data as they are beyond the scope of this document. Parameter data items may contain data that can:

a. Influence paths executed through the Executable Object Code.

b. Activate or deactivate software components and functions.

c. Adapt the software computations to the system configuration.

d. Be used as computational data.

e. Establish time and memory partitioning allotments.

f. Provide initial values to the software component.

Depending on how the parameter data item is to be used in the airborne system, the following should be addressed:

- User-modifiable software guidance.
- Option-selectable software guidance. In cases where the parameter data item activates or deactivates functions, the guidance for deactivated code should be addressed as well.
- Field-loadable software guidance. Of particular concern is detection of corrupted parameter data items, as well as incompatibility between the Executable Object Code and parameter data items.

The parameter data item should be assigned the same software level as the software component using it.

For more information regarding verification of parameter data items, see 6.6.

2.5.2 User-Modifiable Software

A user-modifiable component is that part of the software that may be changed by the user within the modification constraints without certification authority review, if the system requirements provide for user modification. A non-modifiable component is that which is not intended to be changed by the user. The potential effects of user modification are determined by the system safety assessment process and used to develop the software requirements, and then, the software verification process activities. Designing for user-modifiable software is discussed further in section 5.2.3. A change that affects the non-modifiable software, its protection, or the modifiable software boundaries is a software modification and is discussed in section 12.1.1.

Guidance for user-modifiable software includes:

a. The user-modifiable software should not adversely affect safety, operational capabilities, flight crew workload, any non-modifiable software components, or any software protection mechanism used. Unless this can be established, the software may not be classified as user-modifiable. The safety impact of displaying information based on user-modifiable software should also be considered.

b. When the system requirements provide for user modification, then users may modify software within the modification constraints without certification authority review.

c. The system requirements should specify the mechanisms that prevent the user modification from affecting system safety whether or not they are correctly implemented. The software that provides the protection for user modification should be at the same software level as the function it is protecting from errors in the modifiable component.

d. If the system requirements do not include provision for user modification, the software should not be modified by the user unless compliance with this document is demonstrated for the modification.

e. At the time of the user modification, the user should take responsibility for all aspects of the user-modifiable software, for example, software configuration management, software quality assurance, and software verification.

f. The applicant should provide the necessary information to enable the user to manage the software in such a way that the safety of the aircraft is not compromised.

2.5.3 Commercial-Off-The-Shelf Software

COTS software included in airborne systems or equipment should satisfy the objectives of this document.

If deficiencies exist in the software life cycle data of COTS software, the data should be augmented to satisfy the objectives of this document. The guidance in section 12.1.4, Upgrading a Development Baseline, and section 12.3.4, Product Service History, may be relevant in this instance.

2.5.4 Option-Selectable Software

Some airborne systems and equipment may include optional functions that may be selected by software-programmed options rather than by hardware connector pins. The option-selectable software functions are used to select a particular configuration within the target computer. See 4.2.h, 5.2.4, and 6.4.4.3.d.2 for guidance on deactivated code.

When software programmed options are included, a means should be provided to ensure that inadvertent selections involving non-approved configurations for the target computer within the installation environment cannot be made.

2.5.5 Field-Loadable Software

Field-loadable airborne software refers to software that can be loaded without removing the system or equipment from its installation. The safety-related requirements associated with the software loading function are part of the system requirements. If the inadvertent enabling of the software loading function could induce a system failure condition, then a safety-related requirement for the software loading function is specified in the system requirements.

System safety considerations relating to field-loadable software include:

- Detection of corrupted or partially loaded software.
- Determination of the effects of loading the inappropriate software.
- Hardware/software compatibility.
- Software/software compatibility.
- Aircraft/software compatibility.
- Inadvertent enabling of the field-loading function.
- Loss or corruption of the software configuration identification display.

Guidance for field-loadable software includes:

a. Unless otherwise justified by the system safety assessment process, the detection mechanism for partial or corrupted software loads should be assigned the same failure condition or software level as the most severe failure condition or software level associated with the function that uses the software load.

b. 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.

c. The software loading function, including support systems and procedures, should include a means to detect incorrect software and/or hardware and/or aircraft combinations and should provide protection appropriate to the failure condition of the function. If the software consists of multiple configuration items, their compatibility should be ensured.

d. If software is part of an airborne display mechanism that is the means for ensuring that the aircraft conforms to a certified configuration, then that software should either be developed to the highest software level of the software to be loaded, or the system safety assessment process should justify the integrity of an end-to-end check of the software configuration identification.

2.5.6 Software Considerations in System Verification

Guidance for system verification is beyond the scope of this document. However, the software life cycle processes aid and interact with the system verification process and may be able to satisfy some system verification process objectives. Software design details that relate to the system functionality need to be made available to aid system verification.

2.6 SYSTEM CONSIDERATIONS IN SOFTWARE LIFE CYCLE PROCESSES

Credit may be taken from system life cycle processes for the satisfaction, or partial satisfaction, of the software objectives as defined in this document. In such cases, the system activities for which credit is being sought should be shown to meet the applicable objectives of this document with evidence of the completion of planned activities and their outputs identified as part of the software life cycle data.

CHAPTER 3

SOFTWARE LIFE CYCLE

This section discusses the software life cycle processes, software life cycle definition, and transition criteria between software life cycle processes. This document does not prescribe preferred software life cycles and interactions between them. The separation of the processes is not intended to imply a structure for the organization(s) that perform them. For each software product, the software life cycle(s) is constructed that includes these processes.

3.1 SOFTWARE LIFE CYCLE PROCESSES

The software life cycle processes are:

a. The software planning process that defines and coordinates the activities of the software development and integral processes for a project. Section 4 describes the software planning process.

b. The software development processes that produce the software product. These processes are the software requirements process, the software design process, the software coding process, and the integration process. Section 5 describes the software development processes.

c. The integral processes that ensure the correctness and control of, and confidence in the software life cycle processes and their outputs. The integral processes are the software verification process, the software configuration management process, the software quality assurance process, and the certification liaison process. It is important to understand that the integral processes are performed concurrently with the software planning and development processes throughout the software life cycle. Sections 6 through 9 describe the integral processes.

3.2 SOFTWARE LIFE CYCLE DEFINITION

A project defines one or more software life cycle(s) by choosing the activities for each process, specifying a sequence for the activities, and assigning responsibilities for the activities.

For a specific project, the sequencing of these processes is determined by attributes of the project, such as system functionality and complexity, software size and complexity, requirements stability, use of previously developed software, development strategies, and hardware availability. The usual sequence through the software development processes is requirements, design, coding, and integration.

Figure 3-1 illustrates the sequence of software development processes for several components of a single software product with different software life cycles. Component W implements a set of system requirements by developing the software requirements, using those requirements to define a software design, implementing that design into Source Code, and then integrating the component into the hardware. Component X illustrates the use of previously developed software used in a certified product. Component Y illustrates the use of a simple, partitioned function that can be coded directly from the software requirements. Component Z illustrates the use of a prototyping strategy. Usually, the goals of prototyping are to better understand the software requirements and to mitigate development and technical risks. The initial requirements are used as the basis to implement a prototype. This prototype is evaluated in an environment representative of the intended use of the system under development. Results of the evaluation are used to refine the requirements.

The processes of a software life cycle may be iterative, that is, entered and re-entered. The timing and degree of iteration varies due to the incremental development of system functions, complexity, requirements development, hardware availability, feedback to previous processes, and other attributes of the project.

The various parts of the selected software life cycle are tied together with a combination of incremental integration process and software verification process activities.

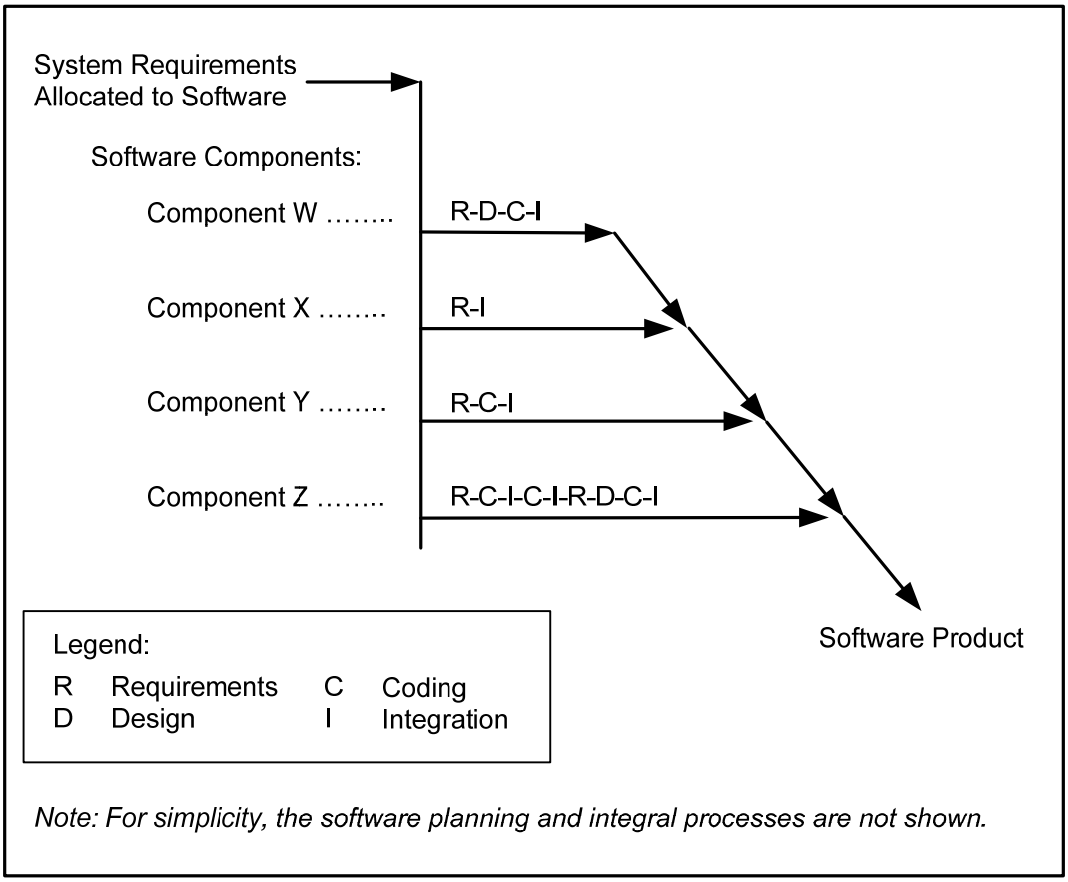

FIGURE 3-1: EXAMPLE OF A SOFTWARE PROJECT USING FOUR DIFFERENT DEVELOPMENT SEQUENCES

3.3 TRANSITION CRITERIA BETWEEN PROCESSES

Transition criteria are used to determine whether a process may be entered or re-entered. Each software life cycle process performs activities on inputs to produce outputs. A process may produce feedback to other processes and receive feedback from others. The definition of feedback includes how information is recognized, controlled, and resolved by the receiving process. An example of a feedback is problem reporting.

The transition criteria will depend on the planned sequence of software development processes and integral processes, and may be affected by the software level. Examples of transition criteria that may be chosen are: the software verification process reviews have been performed, the input is an identified configuration item, and a traceability analysis has been completed for the input.

Not every input to a process need be complete before that process can be initiated, if the transition criteria established for the process are satisfied.

If a process acts on partial inputs, the inputs to the process should be examined to ensure that they meet transition criteria. Also, subsequent inputs to the process should be examined to determine that the previous outputs of the software development and software verification processes are still valid.

CHAPTER 4

SOFTWARE PLANNING PROCESS

This section discusses the objectives and activities of the software planning process. This process produces the software plans and standards that direct the software development processes and the integral processes. Table A-1 of Annex A is a summary of the objectives and outputs of the software planning process by software level.

4.1 SOFTWARE PLANNING PROCESS OBJECTIVES

The purpose of the software planning process is to define the means of producing software that will satisfy its requirements and provide the level of confidence that is consistent with the software level. The objectives of the software planning process are:

a. The activities of the software development processes and integral processes of the software life cycle that will address the system requirements and software level(s) are defined (see 4.2).

b. The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria are determined (see 3).

c. The software life cycle environment, including the methods and tools to be used for the activities of each software life cycle process has been selected and defined (see 4.4).

d. Additional considerations, such as those discussed in section 12, have been addressed, if necessary.

e. Software development standards consistent with the system safety objectives for the software to be produced are defined (see 4.5).

f. Software plans that comply with sections 4.3 and 11 have been produced.

g. Development and revision of the software plans are coordinated (see 4.3).

4.2 SOFTWARE PLANNING PROCESS ACTIVITIES

Effective planning is a determining factor in producing software that satisfies the guidance of this document. Activities for the software planning process include:

a. The software plans should be developed that provide direction to the personnel performing the software life cycle processes. See also 9.1.

b. The software development standards to be used for the project should be defined or selected.

c. Methods and tools should be chosen that aid error prevention and provide defect detection in the software development processes.

d. The software planning process should provide coordination between the software development and integral processes to provide consistency among strategies in the software plans.

e. The means should be specified to revise the software plans as a project progresses.

f. When multiple-version dissimilar software is used in a system, the software planning process should choose the methods and tools to achieve the dissimilarity necessary to satisfy the system safety objectives.

g. For the software planning process to be complete, the software plans and software development standards should be under change control and reviews of them completed (see 4.6).

h. If deactivated code is planned, the software planning process should describe how the deactivation mechanism and deactivated code will be defined and verified to satisfy system safety objectives.

i. If user-modifiable software is planned, related processes, tools, environment, and data items substantiating the design (see 5.2.3) should be specified in the software plans and standards.

j. When parameter data items are planned, the following should be addressed:

   1. The way that parameter data items are used.

   2. The software level of the parameter data items.

   3. The processes to develop, verify, and modify parameter data items, and any associated tool qualification.

   4. Software load control and compatibility.

k. The software planning process should address any additional considerations that are applicable.

l. If software development activities will be performed by a supplier, planning should address supplier oversight.

Other software life cycle processes may begin before completion of the software planning process if transition criteria for the specific process activity are satisfied.

4.3 SOFTWARE PLANS

The software plans define the means of satisfying the objectives of this document. They specify the organizations that will perform those activities. The software plans are:

- The Plan for Software Aspects of Certification (see 11.1) serves as the primary means for communicating the proposed development methods to the certification authority for agreement, and defines the means of compliance with this document.
- The Software Development Plan (see 11.2) defines the software life cycle(s), software development environment, and the means by which the software development process objectives will be satisfied.
- The Software Verification Plan (see 11.3) defines the means by which the software verification process objectives will be satisfied.
- The Software Configuration Management Plan (see 11.4) defines the means by which the software configuration management process objectives will be satisfied.
- The Software Quality Assurance Plan (see 11.5) defines the means by which the software quality assurance process objectives will be satisfied.

Activities for the software plans include:

a. The software plans should comply with this document.

b. The software plans should define the transition criteria for software life cycle processes by specifying:

   1. The inputs to the process, including feedback from other processes.

   2. Any integral process activities that may be required to act on these inputs.

   3. Availability of tools, methods, plans, and procedures.

c. The software plans should state the procedures to be used to implement software changes prior to use on a certified product. Such changes may be as a result of feedback from other processes and may cause a change to the software plans.

4.4 SOFTWARE LIFE CYCLE ENVIRONMENT PLANNING

Planning for the software life cycle environment defines the methods, tools, procedures, programming languages, and hardware that will be used to develop, verify, control, and produce the software life cycle data (see 11) and software product. Examples of how the software environment chosen can have a beneficial effect on the software include enforcing standards, detecting errors, and implementing error prevention and fault tolerance methods. The software life cycle environment is a potential error source that can contribute to failure conditions. Composition of this software life cycle environment may be influenced by the safety-related requirements determined by the system safety assessment process, for example, the use of dissimilar, redundant components.

The goal of error prevention methods is to avoid errors during the software development processes that might contribute to a failure condition. The basic principle is to choose requirements development and design methods, tools, and programming languages that limit the opportunity for introducing errors, and verification methods that ensure that errors introduced are detected. The goal of fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process.

The considerations presented above may affect:

a. The methods and notations used in the software requirements process and software design process.

b. The programming language(s) and methods used in the software coding process.

c. The software development environment tools.

d. The software verification and software configuration management tools.

e. The need for tool qualification (see 12.2).

4.4.1 Software Development Environment

The software development environment is a significant factor in the production of high quality software. The software development environment can also adversely affect the production of software in several ways. For example, a compiler could introduce errors by producing a corrupted output or a linker could fail to reveal a memory allocation error that is present. Activities for the selection of software development environment methods and tools include:

a. During the software planning process, the software development environment should be chosen to reduce its potential risk to the software being developed.

b. The use of tools or combinations of tools and parts of the software development environment should be chosen to achieve the necessary level of confidence that an error introduced by one part would be detected by another. An acceptable environment is produced when both parts are consistently used together. This selection includes the assessment of the need for tool qualification.

c. The software verification process activities or software development standards, which include consideration of the software level, should be defined to reduce potential software development environment-related errors.

d. If certification credit is sought for use of the tools in combination, the sequence of operation of the tools should be specified in the appropriate plan.

e. If optional features of software tools are chosen for use in a project, the effects of the options should be examined and specified in the appropriate plan. This is especially important for compilers and autocode generators.

f. Known tool problems and limitations should be assessed and those issues which can adversely affect airborne software should be addressed.

4.4.2 Language and Compiler Considerations

Upon successful completion of verification of the software product, the compiler is considered acceptable for that product. For this to be valid, the software verification process needs to consider particular features of the programming language and compiler. The software planning process considers these features when choosing a programming language and planning for verification. Activities include:

a. Some compilers have features intended to optimize performance of the object code. If the test cases give coverage consistent with the software level, the correctness of the optimization need not be verified. Otherwise, the impact of these features on structural coverage analysis should be determined. Additional information can be found in section 6.4.4.2.

b. To implement certain features, compilers for some languages may produce object code that is not directly traceable to the Source Code, for example, initialization, built-in error detection, or exception handling (see 6.4.4.2.b). The software planning process should provide a means to detect this object code and to ensure verification coverage, and should define the means in the appropriate plan.

c. If a new compiler, linkage editor, or loader version is introduced, or compiler options are changed during the software life cycle, previous tests and coverage analyses may no longer be valid. The verification planning should provide a means of reverification that is consistent with sections 6 and 12.1.3.

Note: Although the compiler is considered acceptable once all of the verification objectives are satisfied, the compiler is only considered acceptable for that product and not necessarily for other products.

4.4.3 Software Test Environment

Software test environment planning defines the methods, tools, procedures, and hardware that will be used to test the outputs of the integration process. Testing may be performed using the target computer, a target computer emulator, or a host computer simulator. Activities include:

a. The emulator or simulator may need to be qualified as described in section 12.2.

b. The differences between the target computer and the emulator or simulator, and the effects of these differences on the ability to detect errors and verify functionality, should be considered. Detection of those errors should be provided by the software verification process and specified in the Software Verification Plan.

4.5 SOFTWARE DEVELOPMENT STANDARDS

Software development standards define the rules and constraints for the software development processes. The software development standards include the Software Requirements Standards, the Software Design Standards, and the Software Code Standards. The software verification process uses these standards as a basis for evaluating the compliance of actual outputs of a process with intended outputs. Activities for development of the software standards include:

a. The software development standards should comply with section 11.

b. The software development standards should enable software components of a given software product or related set of products to be uniformly designed and implemented.

c. The software development standards should disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements.

d. Robustness should be considered in the software development standards.

NOTE 1: *In developing standards, consideration can be given to previous experience. Constraints and rules on development, design, and coding methods can be included to control complexity. Defensive programming practices may be considered to improve robustness.*

NOTE 2: *If allocated to software by system requirements, practices to detect and control errors in stored data, and refresh and monitor hardware status and configuration may be used to mitigate single event upsets.*

4.6 REVIEW OF THE SOFTWARE PLANNING PROCESS

Reviews of the software planning process are conducted to ensure that the software plans and software development standards comply with the guidance of this document and means are provided to execute them. Activities include:

a. Methods are chosen that enable the objectives of this document to be satisfied.

b. Software life cycle processes can be applied consistently.

c. Each process produces evidence that its outputs can be traced to their activity and inputs, showing the degree of independence of the activity, the environment, and the methods to be used.

d. The outputs of the software planning process are consistent and comply with section 11.

CHAPTER 5

SOFTWARE DEVELOPMENT PROCESSES

This section discusses the objectives and activities of the software development processes. The software development processes are applied as defined by the software planning process (see 4) and the Software Development Plan (see 11.2). Table A-2 of Annex A is a summary of the objectives and outputs of the software development processes by software level. The software development processes are:

- Software requirements process.
- Software design process.
- Software coding process.
- Integration process.

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.

NOTE: *The applicant may be required to justify software development processes that produce a single level of requirements.*

The development of software architecture involves decisions made about the structure of the software. During the software design process, the software architecture is defined and low-level requirements are developed. Low-level requirements are software requirements from which Source Code can be directly implemented without further information.

Each software development process may produce derived requirements. Some examples of requirements that might be determined to be derived requirements are:

- The need for interrupt handling software to be developed for the chosen target computer.
- The specification of a periodic monitor's iteration rate when not specified by the system requirements allocated to software.
- The addition of scaling limits when using fixed point arithmetic.

High-level requirements may include derived requirements, and low-level requirements may include derived requirements. In order to determine the effects of derived requirements on the system safety assessment and system requirements, all derived requirements should be made available to the system processes including the system safety assessment process.

5.1 SOFTWARE REQUIREMENTS PROCESS

The software requirements process uses the outputs of the system life cycle processes to develop the high-level requirements. These high-level requirements include functional, performance, interface, and safety-related requirements.

5.1.1 Software Requirements Process Objectives

The objectives of the software requirements process are:

a. High-level requirements are developed.

b. Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.

5.1.2 Software Requirements Process Activities

Inputs to the software requirements process include the system requirements, the hardware interface and system architecture (if not included in the requirements) from the system life cycle processes, and the Software Development Plan and the Software Requirements Standards from the software planning process. When the planned transition criteria have been satisfied, these inputs are used to develop the high-level requirements.

The primary output of this process is the Software Requirements Data (see 11.9).

The software requirements process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. The system functional and interface requirements that are allocated to software should be analyzed for ambiguities, inconsistencies, and undefined conditions.

b. Inputs to the software requirements process detected as inadequate or incorrect should be reported as feedback to the input source processes for clarification or correction.

c. Each system requirement that is allocated to software should be specified in the high-level requirements.

d. High-level requirements that address system requirements allocated to software to preclude system hazards should be defined.

e. The high-level requirements should conform to the Software Requirements Standards, and be verifiable and consistent.

f. The high-level requirements should be stated in quantitative terms with tolerances where applicable.

g. The high-level requirements should not describe design or verification detail except for specified and justified design constraints.

h. Derived high-level requirements and the reason for their existence should be defined.

i. Derived high-level requirements should be provided to the system processes, including the system safety assessment process.

j. If parameter data items are planned, the high-level requirements should describe how any parameter data item is used by the software. The high-level requirements should also specify their structure, the attributes for each of their data elements, and, when applicable, the value of each element. The values of the parameter data item elements should be consistent with the structure of the parameter data item and the attributes of its data elements.

5.2 SOFTWARE DESIGN PROCESS

The high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements that can be used to implement Source Code.

5.2.1 Software Design Process Objectives

The objectives of the software design process are:

a. The software architecture and low-level requirements are developed from the high-level requirements.

b. Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.

5.2.2 Software Design Process Activities

The software design process inputs are the Software Requirements Data, the Software Development Plan, and the Software Design Standards. When the planned transition criteria have been satisfied, the high-level requirements are used in the design process to develop software architecture and low-level requirements. This may involve one or more lower levels of requirements.

The primary output of the process is the Design Description (see 11.10) which includes the software architecture and the low-level requirements.

The software design process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. Low-level requirements and software architecture developed during the software design process should conform to the Software Design Standards and be traceable, verifiable, and consistent.

b. Derived low-level requirements and the reason for their existence should be defined and analyzed to ensure that the higher level requirements are not compromised.

c. Software design process activities could introduce possible modes of failure into the software or, conversely, preclude others. The use of partitioning or other architectural means in the software design may alter the software level assignment for some components of the software. In such cases, additional data should be defined as derived requirements and provided to the system processes, including the system safety assessment process.

d. Interfaces between software components, in the form of data flow and control flow, should be defined to be consistent between the components.

e. Control flow and data flow should be monitored when safety-related requirements dictate, for example, watchdog timers, reasonableness-checks, and cross-channel comparisons.

f. Responses to failure conditions should be consistent with the safety-related requirements.

g. Inadequate or incorrect inputs detected during the software design process should be provided to the system life cycle processes, the software requirements process, or the software planning process as feedback for clarification or correction.

NOTE: *The current state of software engineering does not permit a quantitative correlation between complexity and the attainment of system safety objectives. While no objective guidance can be provided, the software design process should avoid introducing complexity because as the complexity of software increases, it becomes more difficult to verify the design and to show that the safety-related requirements are satisfied.*

5.2.3 Designing for User-Modifiable Software

User-modifiable software is designed to be modified by its users. A modifiable component is that part of the software that is intended to be changed by the user and a non-modifiable component is that which is not intended to be changed by the user. User-modifiable software may vary in complexity. Examples include a single memory bit used to select one of two equipment options, a table of messages, or a memory area that can be programmed, compiled, and linked for maintenance functions. Software of any level can include a modifiable component.

The activities for user-modifiable software include:

a. The non-modifiable component should be protected from the modifiable component to prevent interference in the safe operation of the non-modifiable component. This protection can be enforced by hardware, by software, by the tools used to make the change, or by a combination of the three. If the protection is provided by software, it should be designed and verified at the same software level as the non-modifiable software. If the protection is provided by a tool, the tool should be categorized and qualified as defined in section 12.2.

b. The means provided to change the modifiable component should be shown to be the only means by which the modifiable component can be changed.

5.2.4 Designing for Deactivated Code

Systems or equipment may be designed to include several configurations, not all of which are intended to be used in every application. This can lead to deactivated code that cannot be executed, such as unselected functionality or unused library functions, or data that is not used. Deactivated code differs from dead code. The activities for deactivated code include:

a. A mechanism should be designed and implemented to assure that deactivated functions or components have no adverse impact on active functions or components.

b. Evidence should be available that the deactivated code is disabled for the environments where its use is not intended. Unintended execution of deactivated code due to abnormal system conditions is the same as unintended execution of activated code.

c. The development of deactivated code, like the development of the active code, should comply with the objectives of this document.

5.3 SOFTWARE CODING PROCESS

In the software coding process, the Source Code is implemented from the software architecture and the low-level requirements.

NOTE: *For the purpose of this document, compiling, linking, and loading are dealt with under the Integration Process (see 5.4).*

5.3.1 Software Coding Process Objectives

The objective of the software coding process is:

a. Source Code is developed from the low-level requirements.

5.3.2 Software Coding Process Activities

The coding process inputs are the low-level requirements and software architecture from the software design process, the Software Development Plan, and the Software Code Standards. The software coding process may be entered or re-entered when the planned transition criteria are satisfied. The Source Code is produced by this process based upon the software architecture and the low-level requirements.

The primary output of this process is Source Code (see 11.11).

The software coding process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. The Source Code should implement the low-level requirements and conform to the software architecture.

b. The Source Code should conform to the Software Code Standards.

c. Inadequate or incorrect inputs detected during the software coding process should be provided to the software requirements process, software design process, and/or software planning process as feedback for clarification or correction.

d. Use of autocode generators should conform to the constraints defined in the planning process.

5.4 INTEGRATION PROCESS

The target computer and the Source Code from the software coding process are used with the compiling, linking, and loading data (see 11.16) in the integration process to develop the integrated system or equipment.

5.4.1 Integration Process Objectives

The objective of the integration process is:

a. The Executable Object Code and its associated Parameter Data Item Files, if any, are produced and loaded into the target hardware for hardware/software integration.

5.4.2 Integration Process Activities

The integration process consists of software integration and hardware/software integration.

The integration process may be entered or re-entered when the planned transition criteria have been satisfied. The integration process inputs are the software architecture from the software design process, and the Source Code from the software coding process.

The outputs of the integration process are the object code; Executable Object Code (see 11.12); Parameter Data Item File (see 11.22); and the compiling, linking, and loading data. The integration process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. The object code and Executable Object Code should be generated from the Source Code and compiling, linking, and loading data. Any Parameter Data Item File should be generated.

b. Software integration should be performed on a host computer, a target computer emulator, or the target computer.

c. The software should be loaded into the target computer for hardware/software integration.

d. Inadequate or incorrect inputs detected during the integration process should be provided to the software requirements process, the software design process, the software coding process, or the software planning process as feedback for clarification or correction.

e. Patches should not be used in software submitted for use in a certified product to implement changes in requirements or architecture, or changes found necessary as a result of the software verification process activities. Patches may be used on a limited, case-by-case basis, for example, to resolve known deficiencies in the software development environment such as a known compiler problem.

f. When a patch is used, these should be available:

1. Confirmation that the software configuration management process can effectively track the patch.

2. An analysis to provide evidence that the patched software satisfies all the applicable objectives.

3. Justification in the Software Accomplishment Summary for use of a patch.

5.5 SOFTWARE DEVELOPMENT PROCESS TRACEABILITY

Software development process traceability activities include:

a. Trace Data, showing the bi-directional association between system requirements allocated to software and high-level requirements, is developed. The purpose of this Trace Data is to:

1. Enable verification of the complete implementation of the system requirements allocated to software.

2. Give visibility to those derived high-level requirements that are not directly traceable to system requirements.

b. Trace Data, showing the bi-directional association between the high-level requirements and low-level requirements, is developed. The purpose of this Trace Data is to:

1. Enable verification of the complete implementation of the high-level requirements.

2. Give visibility to those derived low-level requirements that are not directly traceable to high-level requirements and to the architectural design decisions made during the software design process.

c. Trace Data, showing the bi-directional association between low-level requirements and Source Code, is developed. The purpose of this Trace Data is to:

1. Enable verification that no Source Code implements an undocumented function.

2. Enable verification of the complete implementation of the low-level requirements.

CHAPTER 6

SOFTWARE VERIFICATION PROCESS

This section discusses the objectives and activities of the software verification process. Verification is a technical assessment of the outputs of the software planning process, software development processes, and the software verification process. The software verification process is applied as defined by the software planning process (see 4) and the Software Verification Plan (see 11.3). See 4.6 for the verification of the outputs of the planning process.

Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following sections use the term "verify" instead of "test" to discuss the software verification process activities, which are typically a combination of reviews, analyses, and tests.

Tables A-3 through A-7 of Annex A contain a summary of the objectives and outputs of the software verification process, by software level.

NOTE: *For lower software levels, less emphasis is on:*

- *Verification of Source Code.*
- *Verification of low-level requirements.*
- *Verification of the software architecture.*
- *Degree of test coverage.*
- *Control of verification procedures.*
- *Independence of software verification process activities.*
- *Overlapping software verification process activities, that is, multiple verification activities, each of which may be capable of detecting the same class of error.*
- *Robustness testing.*
- *Verification activities with an indirect effect on error prevention or detection, for example, conformance to software development standards.*

6.1 PURPOSE OF SOFTWARE VERIFICATION

The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Removal of the errors is an activity of the software development processes. The software verification process verifies that:

a. The system requirements allocated to software have been developed into high-level requirements that satisfy those system requirements.

b. The high-level requirements have been developed into software architecture and low-level requirements that satisfy the high-level requirements. If one or more levels of software requirements are developed between high-level requirements and low-level requirements, the successive levels of requirements are developed such that each successively lower level satisfies its higher level requirements. If code is generated directly from high-level requirements, this does not apply.

c. The software architecture and low-level requirements have been developed into Source Code that satisfies the low-level requirements and software architecture.

d. The Executable Object Code satisfies the software requirements (that is, intended function), and provides confidence in the absence of unintended functionality.

e. The Executable Object Code is robust with respect to the software requirements such that it can respond correctly to abnormal inputs and conditions.

f. The means used to perform this verification are technically correct and complete for the software level.

6.2 OVERVIEW OF SOFTWARE VERIFICATION PROCESS ACTIVITIES

Software verification process objectives are satisfied through a combination of reviews, analyses, the development of test cases and procedures, and the subsequent execution of those test procedures. Reviews and analyses provide an assessment of the accuracy, completeness, and verifiability of the software requirements, software architecture, and Source Code. The development of test cases and procedures may provide further assessment of the internal consistency and completeness of the requirements. The execution of the test procedures provides a demonstration of compliance with the requirements.

The inputs to the software verification process include the system requirements, the software requirements, software architecture, Trace Data, Source Code, Executable Object Code, and the Software Verification Plan.

The outputs of the software verification process are recorded in the Software Verification Cases and Procedures (see 11.13), the Software Verification Results (see 11.14), and the associated Trace Data (see 11.21).

The need for the requirements to be verifiable once they have been implemented in the software may itself impose additional requirements or constraints on the software development processes.

Software verification considerations include:

a. If the code tested is not identical to the airborne software, those differences should be specified and justified.

b. When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives defined in the Software Verification Plan or Software Verification Results.

c. Deficiencies and errors discovered during the software verification process should be reported to other software life cycle processes for clarification and correction as applicable.

d. Reverification should be conducted following corrective actions and/or changes that could impact the previously verified functionality. Reverification should ensure that the modification has been correctly implemented.

e. 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.

6.3 SOFTWARE REVIEWS AND ANALYSES

Reviews and analyses are applied to the outputs of the software development processes. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness. A review may consist of an inspection of an output of a process guided by a checklist or similar aid. An analysis may examine in detail the functionality, performance, traceability, and safety implications of a software component, and its relationship to other components within the system or equipment.

There may be cases where the verification objectives described in this section cannot be completely satisfied via reviews and analyses alone. In such cases, those verification objectives may be satisfied with additional testing of the software product.

For example, a combination of reviews, analyses, and tests may be developed to establish the worst-case execution time or verification of the stack usage.

6.3.1 Reviews and Analyses of High-Level Requirements

These review and analysis activities detect and report requirements errors that may have been introduced during the software requirements process. These review and analysis activities confirm that the high-level requirements satisfy these objectives:

a. Compliance with system requirements: The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the high-level requirements, and that derived requirements and the reason for their existence are correctly defined.

b. Accuracy and consistency: The objective is to ensure that each high-level requirement is accurate, unambiguous, and sufficiently detailed, and that the requirements do not conflict with each other.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially system response times and input/output hardware.

d. Verifiability: The objective is to ensure that each high-level requirement can be verified.

e. Conformance to standards: The objective is to ensure that the Software Requirements Standards were followed during the software requirements process and that deviations from the standards are justified.

f. Traceability: The objective is to ensure that the functional, performance, and safety-related requirements of the system that are allocated to software were developed into the high-level requirements.

g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

6.3.2 Reviews and Analyses of Low-Level Requirements

These review and analysis activities detect and report requirements errors that may have been introduced during the software design process. These review and analysis activities confirm that the low-level requirements satisfy these objectives:

a. Compliance with high-level requirements: The objective is to ensure that the low-level requirements satisfy the high-level requirements and that derived requirements and the design basis for their existence are correctly defined.

b. Accuracy and consistency: The objective is to ensure that each low-level requirement is accurate and unambiguous, and that the low-level requirements do not conflict with each other.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the low-level requirements and the hardware/software features of the target computer, especially the use of resources such as bus loading, system response times, and input/output hardware.

d. Verifiability: The objective is to ensure that each low-level requirement can be verified.

e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations from the standards are justified.

f. Traceability: The objective is to ensure that the high-level requirements and derived requirements were developed into the low-level requirements.

g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

6.3.3 Reviews and Analyses of Software Architecture

These review and analysis activities detect and report errors that may have been introduced during the development of the software architecture. These review and analysis activities confirm that the software architecture satisfies these objectives:

a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.

b. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow. If the interface is to a component of a lower software level, it should also be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer.

d. Verifiability: The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms.

e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restriction and design construct rules.

f. Partitioning integrity: The objective is to ensure that partitioning breaches are prevented.

6.3.4 Reviews and Analyses of Source Code

These review and analysis activities detect and report errors that may have been introduced during the software coding process. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. These review and analysis activities are usually confined to the Source Code and confirm that the Source Code satisfies these objectives:

a. Compliance with the low-level requirements: The objective is to ensure that the Source Code is accurate and complete with respect to the low-level requirements and that no Source Code implements an undocumented function.

b. Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture.

c. Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.

d. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, for example, complexity restrictions and code constraints. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.

e. Traceability: The objective is to ensure that the low-level requirements were developed into Source Code.

f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.

6.3.5 Reviews and Analyses of the Outputs of the Integration Process

These review and analysis activities detect and report errors that may have been introduced during the integration process. The objective is to:

a. Ensure that the outputs of the integration process are complete and correct.

Activities include conducting a detailed examination of the compiling, linking and loading data, and memory map. Typical examples of potential errors include:

- Compiler warnings.
- Incorrect hardware addresses.
- Memory overlaps.
- Missing software components.

6.4 SOFTWARE TESTING

Software testing is used to demonstrate that the software satisfies its requirements and to demonstrate with a high degree of confidence that errors that could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.

The objectives of software testing are to execute the software to confirm that:

a. The Executable Object Code complies with the high-level requirements.

b. The Executable Object Code is robust with the high-level requirements.

c. The Executable Object Code complies with the low-level requirements.

d. The Executable Object Code is robust with the low-level requirements.

e. The Executable Object Code is compatible with the target computer.

Figure 6-1 is a diagram of the software testing activities that may be used to achieve the software testing objectives. The diagram shows three types of testing, which are:

- Hardware/software integration testing: To verify correct operation of the software in the target computer environment.
- Software integration testing: To verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.
- Low-level testing: To verify the implementation of low-level requirements.

NOTE: *If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing or software integration testing, and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested.*

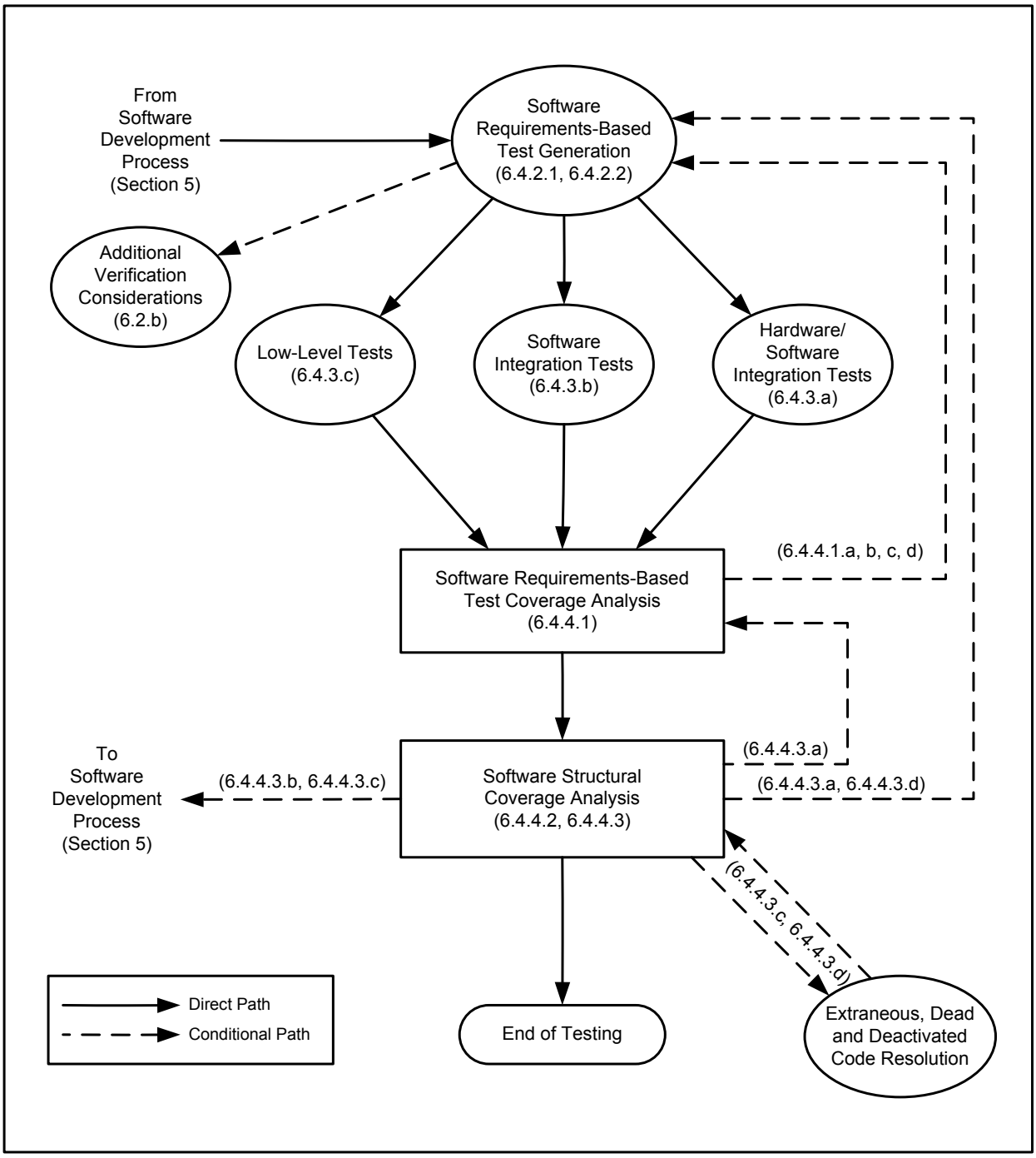

FIGURE 6-1: SOFTWARE TESTING ACTIVITIES

6.4.1 Test Environment

More than one test environment may be needed to satisfy the objectives for software testing. A preferred test environment includes the software loaded into the target computer and tested in an environment that closely resembles the behavior of the target computer environment.

NOTE: *In many cases, the requirements-based coverage and structural coverage necessary can be achieved only with more precise control and monitoring of the test inputs and code execution than generally possible in a fully integrated environment. Such testing may need to be performed on a small software component that is functionally isolated from other software components.*

Certification credit may be given for testing done using a target computer emulator or a host computer simulator. Activities related to the test environment include:

a. Selected tests should be performed in the integrated target computer environment, since some errors are only detected in this environment.

6.4.2 Requirements-Based Test Selection

Requirements-based testing is emphasized because this strategy has been found to be the most effective at revealing errors. Activities for requirements-based test selection include:

a. Specific test cases should be developed to include normal range test cases and robustness (abnormal range) test cases.

b. The specific test cases should be developed from the software requirements and the error sources inherent in the software development processes.

NOTE: *Robustness test cases are requirements-based. The robustness testing criteria cannot be fully satisfied if the software requirements do not specify the correct software response to abnormal conditions and inputs. The test cases may reveal inadequacies in the software requirements, in which case the software requirements should be modified. Conversely, if a complete set of requirements exists that covers all abnormal conditions and inputs, the robustness test cases will follow from those software requirements.*

c. Test procedures are generated from the test cases.

6.4.2.1 Normal Range Test Cases

Normal range test cases demonstrate the ability of the software to respond to normal inputs and conditions. Activities include:

a. Real and integer input variables should be exercised using valid equivalence classes and boundary values.

b. For time-related functions, such as filters, integrators, and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.

c. For state transitions, test cases should be developed to exercise the transitions possible during normal operation.

d. For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

6.4.2.2 Robustness Test Cases

Robustness test cases demonstrate the ability of the software to respond to abnormal inputs and conditions. Activities include:

a. Real and integer variables should be exercised using equivalence class selection of invalid values.

b. System initialization should be exercised during abnormal conditions.

c. The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.

d. For loops where the loop count is a computed value, test cases should be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.

e. A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.

f. For time-related functions, such as filters, integrators, and delays, test cases should be developed for arithmetic overflow protection mechanisms.

g. For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

6.4.3 Requirements-Based Testing Methods

The requirements-based testing methods discussed in this document are requirements-based hardware/software integration testing, requirements-based software integration testing, and requirements-based low-level testing. With the exception of hardware/software integration testing, these methods do not prescribe a specific test environment or strategy. Activities include:

a. Requirements-Based Hardware/Software Integration Testing: This testing method should concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. Requirements-based hardware/software integration testing ensures that the software in the target computer will satisfy the high-level requirements. Typical errors revealed by this testing method include:

- Incorrect interrupt handling.
- Failure to satisfy execution time requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads, and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.
- Errors in hardware/software interfaces.
- Incorrect behavior of control loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.

b. Requirements-Based Software Integration Testing: This testing method should concentrate on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. Requirements-based software integration testing ensures that the software components interact correctly with each other and satisfy the software requirements and software architecture. This method may be performed by expanding the scope of requirements through successive integration of code components with a corresponding expansion of the scope of the test cases. Typical errors revealed by this testing method include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

c. Requirements-Based Low-Level Testing: This testing method should concentrate on demonstrating that each software component complies with its low-level requirements. Requirements-based low-level testing ensures that the software components satisfy their low-level requirements. Typical errors revealed by this testing method include:

- Failure of an algorithm to satisfy a software requirement.
- Incorrect loop operations.
- Incorrect logic decisions.
- Failure to process correctly legitimate combinations of input conditions.
- Incorrect responses to missing or corrupted input data.
- Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
- Incorrect computation sequence.
- Inadequate algorithm precision, accuracy, or performance.

6.4.4 Test Coverage Analysis

Test coverage analysis is a two step process involving requirements-based coverage analysis and structural coverage analysis. The first step analyzes the test cases in relation to the software requirements to confirm that the selected test cases satisfy the specified criteria. The second step confirms that the requirements-based test procedures exercised the code structure to the applicable coverage criteria. If the structural coverage analysis showed the applicable coverage was not met, additional activities are identified for resolution of such situations as dead code (see 6.4.4.3)

The objectives for test coverage analysis are:

a. Test coverage of high-level requirements is achieved.

b. Test coverage of low-level requirements is achieved.

c. Test coverage of software structure to the appropriate coverage criteria is achieved.

d. Test coverage of software structure, both data coupling and control coupling, is achieved.

6.4.4.1 Requirements-Based Test Coverage Analysis

This analysis is to determine how well the requirements-based testing verified the implementation of the software requirements. This analysis may reveal the need for additional requirements-based test cases. Activities include:

a. Analysis, using the associated Trace Data, to confirm that test cases exist for each software requirement.

b. Analysis to confirm that test cases satisfy the criteria of normal and robustness testing as defined in section 6.4.2.

c. Resolution of any deficiencies identified in the analysis. Possible solutions are adding or enhancing test cases.

d. Analysis to confirm that all the test cases, and thus all the test procedures, used to achieve structural coverage, are traceable to requirements.

6.4.4.2 Structural Coverage Analysis

This analysis determines which code structure, including interfaces between components, was not exercised by the requirements-based test procedures. The requirements-based test cases may not have completely exercised the code structure, including interfaces, so structural coverage analysis is performed and additional verification produced to provide structural coverage. Activities include:

a. Analysis of the structural coverage information collected during requirements-based testing to confirm that the degree of structural coverage is appropriate to the software level.

b. 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.*

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

d. Structural coverage analysis resolution (see 6.4.4.3).

6.4.4.3 Structural Coverage Analysis Resolution

Structural coverage analysis may reveal code structure including interfaces that were not exercised during testing. Resolution will require additional software verification process activity. Causes of unexecuted code structure including interfaces, and associated activities to resolve them include:

a. Shortcomings in requirements-based test cases or procedures: The test cases should be augmented or test procedures changed to provide the missing coverage. The method(s) used to perform the requirements-based coverage analysis may need to be reviewed.

b. Inadequacies in software requirements: The software requirements should be modified and additional test cases developed and test procedures executed.

c. Extraneous code, including dead code: The code should be removed and an analysis performed to assess the effect and the need for reverification. If extraneous code is found at the Source Code or object code level, it may be allowed to remain only if analysis shows it does not exist in the Executable Object Code (for example, due to smart compiling, linking, or some other mechanism), and procedures are in place to prevent inclusion in future builds.

d. Deactivated code: Deactivated code should be handled in one of two ways, depending upon its defined category:

1. Category One: Deactivated code that is not intended to be executed in any current configuration used within any certified product. For this category, a combination of analysis and testing should show that the means by which the deactivated code could be inadvertently executed are prevented, isolated, or eliminated. Any reassignment of the software level for Category One deactivated code should be justified by the system safety assessment process and documented in the Plan for Software Aspects of Certification. Similarly, any alleviation of the software verification process for Category One deactivated code should be justified by the software development processes and documented in the Plan for Software Aspects of Certification.

2. Category Two: Deactivated code that is only executed in certain approved configurations of the target computer environment. The operational configuration needed for normal execution of this code should be established and additional test cases and test procedures developed to satisfy the required coverage objectives.

6.4.5 Reviews and Analyses of Test Cases, Procedures, and Results

These review and analysis activities confirm that the software testing satisfies these objectives:

a. Test cases: The objectives related to verification of test cases are presented in sections 6.4.4.a and 6.4.4.b.

b. Test procedures: The objective is to verify that the test cases, including expected results, were correctly developed into test procedures.

c. Test results: The objective is to ensure that the test results are correct and that discrepancies between actual and expected results are explained.

6.5 SOFTWARE VERIFICATION PROCESS TRACEABILITY

Software verification process traceability activities include:

a. Trace Data, showing the bi-directional association between software requirements and test cases, is developed. This Trace Data supports the requirements-based test coverage analysis.

b. Trace Data, showing the bi-directional association between test cases and test procedures, is developed. This Trace Data enables verification that the complete set of test cases has been developed into test procedures.

c. Trace Data, showing the bi-directional association between test procedures and test results, is developed. This Trace Data enables verification that the complete set of test procedures has been executed.

6.6 VERIFICATION OF PARAMETER DATA ITEMS

Verification of a parameter data item can be conducted separately from the verification of the Executable Object Code if all of the following apply:

- The Executable Object Code has been developed and verified by normal range testing to correctly handle all Parameter Data Item Files that comply with their defined structure and attributes.
- The Executable Object Code is robust with respect to Parameter Data Item Files structures and attributes.
- All behavior of the Executable Object Code resulting from the contents of the Parameter Data Item File can be verified.
- The structure of the life cycle data allows the parameter data item to be managed separately.

Unless all conditions above are met, parameter data items should not be verified separately from the rest of the software. In this case, the Executable Object Code and the Parameter Data Item Files should be verified together.

For parameter data items that can be verified separately from verification of the Executable Object Code, the objectives identified below apply. The objectives below may be achieved by a combination of tests, analyses, and reviews.

a. The Parameter Data Item File should be verified to comply with its structure as defined by the high-level requirements; this verification includes ensuring that the Parameter Data Item File does not contain any elements not defined by the high-level requirements. Each data element in the Parameter Data Item File should also be shown to have the correct value, to be consistent with other data elements, and to comply with its attributes as defined by the high-level requirements.

NOTE: *For certain data elements, their attributes may be the only aspect that needs to be verified. In other cases, the value of the data element may need to be verified.*

b. All elements of the Parameter Data Item File have been covered during verification.

If changes are made to the structure or attributes of the parameter data item, then the need for modification and reverification of the Executable Object Code should be analyzed.

CHAPTER 7

SOFTWARE CONFIGURATION MANAGEMENT PROCESS

This section discusses the objectives and activities of the software configuration management (SCM) process. The SCM process is applied as defined by the software planning process (see 4) and the Software Configuration Management Plan (see 11.4). Outputs of the SCM process are recorded in Software Configuration Management Records (see 11.18) or in other software life cycle data.

The SCM process, working in cooperation with the other software life cycle processes, assists in:

a. Providing a defined and controlled configuration of the software throughout the software life cycle.

b. Providing the ability to consistently replicate the Executable Object Code and Parameter Data Item Files, if any, for software manufacture or to regenerate it in case of a need for investigation or modification.

c. Providing control of process inputs and outputs during the software life cycle that ensures consistency and repeatability of process activities.

d. Providing a known point for review, assessing status, and change control by control of configuration items and the establishment of baselines.

e. Providing controls that ensure problems receive attention and changes are recorded, approved, and implemented.

f. Providing evidence of approval of the software by control of the outputs of the software life cycle processes.

g. Assessing the software product compliance with requirements.

h. Ensuring that secure physical archiving, recovery, and control are maintained for the configuration items.

7.1 SOFTWARE CONFIGURATION MANAGEMENT PROCESS OBJECTIVES

The SCM process objectives are:

a. Each configuration item and its successive versions are labeled unambiguously so that a basis is established for the control and reference of configuration items.

b. Baselines are defined for further software life cycle process activity and allow reference to, control of, and traceability between, configuration items.

c. The problem reporting process records process non-compliance with software plans and standards, records deficiencies of outputs of software life cycle processes, records anomalous behavior of software products, and ensures resolution of these problems.

d. Change control provides for recording, evaluation, resolution, and approval of changes throughout the software life cycle.

e. Change review ensures problems and changes are assessed, approved, or disapproved, approved changes are implemented, and feedback is provided to affected processes through problem reporting and change control methods defined during the software planning process.

f. Status accounting provides data for the configuration management of software life cycle processes with respect to configuration identification, baselines, Problem Reports, and change control.

g. Archival and retrieval ensures that the software life cycle data associated with the software product can be retrieved in case of a need to duplicate, regenerate, retest or modify the software product. The objective of the release activity is to ensure that only authorized software is used, especially for software manufacturing, in addition to being archived and retrievable.

h. Software load control ensures that the Executable Object Code and Parameter Data Item Files, if any, are loaded into the system or equipment with appropriate safeguards.

i. Software life cycle environment control ensures that the tools used to produce the software are identified, controlled, and retrievable.

The objectives for SCM are independent of software level. However, two categories of software life cycle data may exist based on the SCM controls applied to the data (see 7.3).

Table A-8 of Annex A is a summary of the objectives and outputs of the SCM process.

7.2 SOFTWARE CONFIGURATION MANAGEMENT PROCESS ACTIVITIES

The SCM process includes the activities of configuration identification, change control, baseline establishment, and archiving of the software product, including the related software life cycle data. The SCM process does not stop when the software product is approved by the certification authority, but continues throughout the service life of the system or equipment. If software life cycle activities will be performed by a supplier, then configuration management activities should be applied to the supplier.

7.2.1 Configuration Identification

Activities include:

a. Configuration identification should be established for the software life cycle data.

b. Configuration identification should be established for each configuration item, for each separately controlled component of a configuration item, and for combinations of configuration items that comprise a software product.

c. Configuration items should be configuration identified prior to the implementation of change control and traceability analysis.

d. A configuration item should be configuration identified before that item is used by other software life cycle processes, referenced by other software life cycle data, or used for software manufacture or software loading.

e. If the software product identification cannot be determined by physical examination (for example, part number plate examination), then the Executable Object Code and Parameter Data Item Files, if any, should contain configuration identification that can be accessed by other parts of the system or equipment. This may be applicable to field-loadable software.

7.2.2 Baselines and Traceability

Activities include:

a. Baselines should be established for configuration items used for certification credit. Intermediate baselines may be established to aid in controlling software life cycle process activities.

b. A software product baseline should be established for the software product and defined in the Software Configuration Index (see 11.16).

NOTE: *User-modifiable software is not included in the software product baseline, except for its associated protection and boundary components. Therefore, modifications may be made to user-modifiable software without affecting the configuration identification of the software product baseline.*

c. Baselines should be established in controlled software libraries, whether physical, electronic, or other, to ensure their integrity. Once a baseline is established, it should be protected from change.

d. Change control activities should be followed to develop a derivative baseline from an established baseline.

e. A baseline should be traceable to the baseline from which it was derived, if certification credit is sought for software life cycle process activities or data associated with the development of the previous baseline.

f. A configuration item should be traceable to the configuration item from which it was derived, if certification credit is sought for software life cycle process activities or data associated with the development of the previous configuration item.

g. A baseline or configuration item should be traceable either to the output it identifies or to the process with which it is associated.

7.2.3 Problem Reporting, Tracking, and Corrective Action

Activities include:

a. A Problem Report should be prepared that describes the process non-compliance with plans, output deficiency, or software anomalous behavior, and the corrective action taken, as defined in section 11.17.

NOTE: *Software life cycle process and software product problems may be recorded in separate problem reporting systems.*

b. Problem reporting should provide for configuration identification of affected configuration item(s) or definition of affected process activities, status reporting of Problem Reports, and approval and closure of Problem Reports.

c. Problem Reports that require corrective action of the software product or outputs of software life cycle processes should invoke the change control activity.

7.2.4 Change Control

Activities include:

a. Change control should preserve the integrity of the configuration items and baselines by providing protection against their change.

b. Change control should ensure that any change to a configuration item requires a change to its configuration identification.

c. Changes to baselines and to configuration items under change control to produce derivative baselines should be recorded, approved, and tracked. Problem reporting is related to change control, since resolution of a reported problem may result in changes to configuration items or baselines.

NOTE: *It is generally recognized that early implementation of change control assists the control and management of software life cycle process activities.*

d. Software changes should be traced to their origin and the software life cycle processes repeated from the point at which the change affects their outputs. For example, an error discovered at hardware/software integration, that is shown to result from an incorrect design, should result in design correction, code correction, and repetition of the associated integral process activities.

e. Throughout the change activity, software life cycle data affected by the change should be updated and records should be maintained for the change control activity.

The change control activity is aided by the change review activity.

7.2.5 Change Review

Activities include:

a. Assessment of the impact of the problem or proposed change on system requirements. Feedback should be provided to the system processes, including the system safety assessment process, and any responses from the system processes should be assessed.

b. Assessment of the impact of the problem or proposed change on software life cycle data identifying changes to be made and actions to be taken.

c. Confirmation that affected configuration items are configuration identified.

d. Feedback of Problem Report or change impact and decisions to affected processes.

7.2.6 Configuration Status Accounting

Activities include:

a. Reporting on configuration item identification, baseline identification, Problem Report status, change history, and release status.

b. Definition of the data to be maintained and the means of recording and reporting status of this data.

7.2.7 Archive, Retrieval, and Release

Activities include:

a. Software life cycle data associated with the software product should be retrievable from the approved source (for example, an archive at the developing organization or company).

b. Procedures should be established to ensure the integrity of the stored data, regardless of medium of storage, by:

   1. Ensuring that no unauthorized changes can be made.

   2. Selecting storage media that minimize regeneration errors or deterioration.

   3. Preventing loss or corruption of data over time. Depending on the storage media used, this may include periodically exercising the media or refreshing the archived data.

   4. Storing duplicate copies in physically separate archives that minimize the risk of loss in the event of a disaster.

c. The duplication process should be verified to produce accurate copies, and procedures should exist that ensure error-free copying of the Executable Object Code and Parameter Data Item Files, if any

d. Configuration items should be identified and released prior to use for software manufacture and the authority for their release should be established. As a minimum, the components of the software product loaded into the airborne system or equipment should be released. This includes the Executable Object Code and Parameter Data Item Files, if any, and may also include associated media for software loading.

NOTE: *Release is generally also required for the data that defines the approved software for loading into the airborne system or equipment. Definition of that data is outside the scope of this document, but may include the Software Configuration Index.*

e. Data retention procedures should be established to satisfy airworthiness requirements and enable software modifications.

NOTE: *Additional data retention considerations may include items such as business needs and future certification authority reviews, which are outside the scope of this document.*

7.3 DATA CONTROL CATEGORIES

Software life cycle data can be assigned to one of two configuration management control categories: Control Category 1 (CC1) and Control Category 2 (CC2). Table 7-1 defines the set of SCM process activities associated with each control category, where ● indicates the minimum activities that apply for software life cycle data of that category. CC2 activities are a subset of the CC1 activities.

The Annex A tables specify the control category by software level for the software life cycle data items.

TABLE 7-1: SCM PROCESS ACTIVITIES ASSOCIATED WITH CC1 and CC2 DATA

SCM Process Activity	Reference	CC1	CC2
Configuration Identification	7.2.1	●	●
Baselines	7.2.2.a 7.2.2.b 7.2.2.c 7.2.2.d 7.2.2.e	●	
Traceability	7.2.2.f 7.2.2.g	●	●
Problem Reporting	7.2.3	●	
Change Control - integrity and identification	7.2.4.a 7.2.4.b	●	●
Change Control - tracking	7.2.4.c 7.2.4.d 7.2.4.e	●	
Change Review	7.2.5	●	
Configuration Status Accounting	7.2.6	●	
Retrieval	7.2.7.a	●	●
Protection against Unauthorized Changes	7.2.7.b.1	●	●
Media Selection, Refreshing, Duplication	7.2.7.b.2 7.2.7.b.3 7.2.7.b.4 7.2.7.c	●	
Release	7.2.7.d	●	
Data Retention	7.2.7.e	●	●

7.4 SOFTWARE LOAD CONTROL

Software load control refers to the process by which programmed instructions and data are transferred from a master memory device into the system or equipment. For example, methods may include (subject to approval by the certification authority) the installation of factory pre-programmed memory devices or “in situ” re-programming of the system or equipment using a field loading device. Whichever method is used, software load control should include:

a. Procedures for part numbering and media identification that identify software configurations that are intended to be approved for loading into the airborne system or equipment.

b. Whether the software is delivered as an end item or is delivered installed in the airborne system or equipment, records should be kept that confirm software compatibility with the airborne system or equipment hardware.

7.5 SOFTWARE LIFE CYCLE ENVIRONMENT CONTROL

The software life cycle environment tools are defined by the software planning process and identified in the Software Life Cycle Environment Configuration Index (see 11.15). Activities include:

a. Configuration identification should be established for the Executable Object Code, or equivalent, of the tools used to develop, control, build, verify, and load the software.

b. The SCM process for controlling qualified tools should comply with the objectives associated with Control Category 1 or Control Category 2 data (see 7.3), according to the guidance provided by section 12.2.3.

c. Unless section 7.5b applies, the SCM process for controlling the Executable Object Code, or equivalent, for tools used to build and load the software (for example, compilers, assemblers, and linkage editors) should comply with the objectives associated with Control Category 2 data, as a minimum.

CHAPTER 8

SOFTWARE QUALITY ASSURANCE PROCESS

This section discusses the objectives and activities of the software quality assurance (SQA) process. The SQA process is applied as defined by the software planning process (see 4) and the Software Quality Assurance Plan (see 11.5). Outputs of the SQA process activities are recorded in Software Quality Assurance Records (see 11.19) or other software life cycle data.

The SQA process assesses the software life cycle processes and their outputs to obtain assurance that objectives are satisfied, deficiencies are detected, evaluated, tracked, and resolved, and software product and software life cycle data conform to certification requirements.

8.1 SOFTWARE QUALITY ASSURANCE PROCESS OBJECTIVES

The SQA process objectives provide confidence that the software life cycle processes produce software that conforms to its requirements by assuring that these processes are performed in compliance with the approved software plans and standards.

The objectives of the SQA process are to obtain assurance that:

a. Software plans and standards are developed and reviewed for compliance with this document and for consistency.

b. Software life cycle processes, including those of suppliers, comply with approved software plans and standards.

c. The transition criteria for the software life cycle processes are satisfied.

d. A conformity review of the software product is conducted.

Table A-9 of Annex A is a summary of the objectives and outputs of the SQA process.

8.2 SOFTWARE QUALITY ASSURANCE PROCESS ACTIVITIES

Activities for satisfying the SQA process objectives include:

a. The SQA process should take an active role in the activities of the software life cycle processes, and have those performing the SQA process enabled with the authority, responsibility, and independence to ensure that the SQA process objectives are satisfied.

b. The SQA process should provide assurance that software plans and standards are developed and reviewed for compliance with this document and for consistency.

c. The SQA process should provide assurance that the software life cycle processes comply with the approved software plans and standards.

d. The SQA process should include audits of the software life cycle processes during the software life cycle to obtain assurance that:

    1. Software plans are available as specified in section 4.2.

    2. Deviations from the software plans and standards are detected, recorded, evaluated, tracked, and resolved.

NOTE: *It is generally accepted that early detection of process deviations assists efficient achievement of software life cycle process objectives.*

    3. Approved deviations are recorded.

    4. The software development environment has been provided as specified in the software plans.

5. The problem reporting, tracking, and corrective action process activities comply with the Software Configuration Management Plan.

6. Inputs provided to the software life cycle processes by the system processes, including the system safety assessment process, have been addressed.

NOTE: *Monitoring of the activities of software life cycle processes may be performed to provide assurance that the activities are under control.*

e. The SQA process should provide assurance that the transition criteria for the software life cycle processes have been satisfied in compliance with the approved software plans.

f. The SQA process should provide assurance that software life cycle data is controlled in accordance with the control categories as defined in section 7.3 and the tables of Annex A.

g. Prior to the delivery of software products submitted as part of a certification application, a software conformity review should be conducted.

h. The SQA process should produce records of the SQA process activities (see 11.19), including audit results and evidence of completion of the software conformity review for each software product submitted as part of certification application.

i. The SQA process should provide assurance that supplier processes and outputs comply with approved software plans and standards.

8.3 SOFTWARE CONFORMITY REVIEW

The purpose of the software conformity review is to obtain assurance, for a software product submitted as part of a certification application, that the software life cycle processes are complete, software life cycle data is complete, and the Executable Object Code and Parameter Data Item Files, if any, are controlled and can be regenerated.

This review should determine that:

a. Planned software life cycle process activities for certification credit, including the generation of software life cycle data, have been completed and records of their completion are retained.

b. Software life cycle data developed from specific system requirements, safety-related requirements, or software requirements are traceable to those requirements.

c. Evidence exists that software life cycle data have been produced in accordance with software plans and standards, and is controlled in accordance with the SCM Plan.

d. Evidence exists that Problem Reports have been evaluated and have their status recorded.

e. Software requirement deviations are recorded and approved.

f. The Executable Object Code and Parameter Data Item Files, if any, can be regenerated from the archived Source Code.

g. The approved software can be loaded successfully through the use of released instructions.

h. Problem Reports deferred from a previous software conformity review are re-evaluated to determine their status.

i. If certification credit is sought for the use of previously developed software, the current software product baseline is traceable to the previous baseline and the approved changes to that baseline.

NOTE: *For post-certification software modifications, a subset of the software conformity review activities, as justified by the significance of the change, may be performed.*

CHAPTER 9

CERTIFICATION LIAISON PROCESS

The objectives of the certification liaison process are to:

a. Establish communication and understanding between the applicant and the certification authority throughout the software life cycle to assist the certification process.

b. Gain agreement on the means of compliance through approval of the Plan for Software Aspects of Certification.

c. Provide compliance substantiation.

The certification liaison process is applied as defined by the software planning process (see 4) and the Plan for Software Aspects of Certification (see 11.1). Table A-10 of Annex A is a summary of the objectives and outputs of this process.

9.1 MEANS OF COMPLIANCE AND PLANNING

The applicant proposes a means of compliance that defines how the development of the airborne system or equipment will satisfy the certification basis. The Plan for Software Aspects of Certification (see 11.1) defines the software aspects of the airborne system or equipment within the context of the proposed means of compliance. This plan also states the software level(s) as determined by the system safety assessment process.

Activities include:

a. Submitting the Plan for Software Aspects of Certification and other requested data to the certification authority for review.

b. Resolving issues identified by the certification authority concerning the planning for the software aspects of certification.

c. Obtaining agreement with the certification authority on the Plan for Software Aspects of Certification.

9.2 COMPLIANCE SUBSTANTIATION

The applicant provides evidence that the software life cycle processes satisfy the software plans, by making software life cycle data available to the certification authority for review. Certification authority reviews may take place at various facilities. For example, the reviews may take place at the applicant's facilities, the applicant's supplier's facilities, or at the certification authority’s facilities. This may involve discussions with the applicant or its suppliers. The applicant arranges these reviews of the activities of the software life cycle processes and makes software life cycle data available as needed.

Activities include:

a. Resolving issues raised by the certification authority as a result of its reviews.

b. Submitting the Software Accomplishment Summary (see 11.20) and Software Configuration Index (see 11.16) to the certification authority.

c. Submitting or making available other data or evidence of compliance requested by the certification authority.

9.3 MINIMUM SOFTWARE LIFE CYCLE DATA SUBMITTED TO CERTIFICATION AUTHORITY

The minimum software life cycle data that is submitted to the certification authority is:

a. Plan for Software Aspects of Certification.

b. Software Configuration Index.

c. Software Accomplishment Summary.

9.4 SOFTWARE LIFE CYCLE DATA RELATED TO TYPE DESIGN

Unless otherwise agreed by the certification authority, the regulations concerning retrieval and approval of software life cycle data related to the type design of the product to be certified applies to:

a. Software Requirements Data.

b. Design Description.

c. Source Code.

d. Executable Object Code and Parameter Data Item Files, if any.

e. Software Configuration Index.

f. Software Accomplishment Summary.

CHAPTER 10

OVERVIEW OF CERTIFICATION PROCESS

This section is an overview of the certification process with respect to software aspects of airborne systems and equipment, and is provided for information purposes only.

The airborne community and certification authorities use several terms related to aircraft approval for flight with its associated equipment. The terms used are “certification”, “approval”, and, with respect to tools, “qualification”.

“Certification” applies to aircraft, engines, or propellers; and, in respect of some certification authorities, auxiliary power units. The certification authorities consider the software as part of the airborne system or equipment installed on the certified product; that is, the certification authorities do not certify the software as a unique, stand-alone product.

Systems and equipment, including embedded software, should be “approved” in order to be accepted as a part of a certification. Approval by the certification authorities is given dependent upon a successful demonstration or by review of the products of the software life cycle. Any such approval currently has significance only within the context of a specific certification.

Tool “qualification” is discussed in section 12.2.

10.1 CERTIFICATION BASIS

The certification authority establishes the certification basis for the product to be certified in consultation with the applicant. The certification basis defines the particular regulations together with any special conditions to be applied beyond the published regulations.

When certified products are being modified, the certification authority considers the impact the modification has on the original certification basis. In some cases, the certification basis for the modification may not change from the original certification basis; however, the original means of compliance may not be applicable for showing that the modification complies with the certification basis and may need to be changed.

10.2 SOFTWARE ASPECTS OF CERTIFICATION

The certification authority assesses the Plan for Software Aspects of Certification for completeness and consistency with the means of compliance that was agreed upon to satisfy the certification basis. The certification authority satisfies itself that the software level(s) proposed by the applicant is consistent with the outputs of the system safety assessment process and other system life cycle data. The certification authority informs the applicant of issues with the proposed software plans that need to be satisfied prior to certification authority agreement.

10.3 COMPLIANCE DETERMINATION

Prior to certification, the certification authority determines that the product to be certified, including the software aspects of its systems or equipment, complies with the certification basis. For the software, this is accomplished by reviewing the Software Accomplishment Summary and evidence of compliance. The certification authority uses the Software Accomplishment Summary as an overview for the software aspects of certification.

The certification authority may review at its discretion the software life cycle processes and their outputs during the software life cycle, as discussed in section 9.2.

CHAPTER 11

SOFTWARE LIFE CYCLE DATA

Data is produced during the software life cycle to plan, direct, explain, define, record, or provide evidence of activities. This data enables the software life cycle processes, system or equipment certification, and post-certification modification of the software product. This section discusses the characteristics, form, configuration management controls, and content of the software life cycle data.

a. Characteristics: Software life cycle data should be:

   1. Unambiguous: Information is unambiguous if it is written in terms which only allow a single interpretation, aided, if necessary, by a definition.

   2. Complete: Information is complete when it includes necessary and relevant requirements and/or descriptive material; responses are defined for the range of valid input data; figures used are labeled; and terms and units of measure are defined.

   3. Verifiable: Information is verifiable if it can be checked for correctness by a person or tool.

   4. Consistent: Information is consistent if there are no conflicts within it.

   5. Modifiable: Information is modifiable if it is structured and has a style such that changes can be made completely, consistently, and correctly while retaining the structure.

   6. Traceable: Information is traceable if the origin of its components can be determined.

b. Form: The form of the software life cycle data should provide for the efficient retrieval and review of software life cycle data throughout the service life of the airborne system or equipment. The data and the specific form of the data should be specified in the Plan for Software Aspects of Certification.

NOTE 1: *The software life cycle data may be held in a number of forms (for example, held electronically or in printed form).*

NOTE 2: *The applicant may package software life cycle data items in any manner the applicant finds convenient (for example, as individual data items or as a combined data item).*

NOTE 3: *The Plan for Software Aspects of Certification and the Software Accomplishment Summary may be required as separate documents by some certification authorities.*

NOTE 4: *The term "data" refers to evidence and other information and does not imply the format such data should take.*

c. Configuration management controls: Software life cycle data can be placed in one of two categories associated with the software configuration management controls applied: CC1 and CC2 (see 7.3). The minimum control category assigned to each data item, and its variation by software level is specified in the tables of Annex A. If additional data items than those described herein are produced as evidence to aid the certification process, they should be, as a minimum, under CC2 controls.

d. Content: The software life cycle data descriptions provided in the following sections identify the data that is generally produced by the software life cycle. The descriptions are not intended to describe all data that may be necessary to develop a software product, and are not intended to imply a particular data packaging method or organization of the data within a package. The descriptions of the content of software life cycle data items provided herein are not all encompassing and should be read in conjunction with the body of this document and adapted to the needs of the applicant.

11.1 PLAN FOR SOFTWARE ASPECTS OF CERTIFICATION

The Plan for Software Aspects of Certification (PSAC) is the primary means used by the certification authority for determining whether an applicant is proposing a software life cycle that is commensurate with the rigor required for the level of software being developed. This plan should include:

a. System overview: This section provides an overview of the system, including a description of its functions and their allocation to the hardware and software, the architecture, processor(s) used, hardware/software interfaces, and safety features.

b. Software overview: This section briefly describes the software functions with emphasis on the proposed safety and partitioning concepts. Examples include resource sharing, redundancy, fault tolerance, mitigation of single event upset, and timing and scheduling strategies.

c. Certification considerations: This section provides a summary of the certification basis, including the means of compliance, as relating to the software aspects of certification. This section also states the proposed software level(s) and summarizes the justification provided by the system safety assessment process, including potential software contributions to failure conditions.

d. Software life cycle: This section defines the software life cycle to be used and includes a summary of each of the software life cycle processes for which detailed information is defined in their respective software plans. The summary explains how the objectives of each software life cycle process will be satisfied, and specifies the organizations to be involved, the organizational responsibilities, and the system life cycle processes and certification liaison process responsibilities.

e. Software life cycle data: This section specifies the software life cycle data that will be produced and controlled by the software life cycle processes. This section also describes the relationship of the data to each other or to other data defining the system, the software life cycle data to be submitted to the certification authority, the form of the data, and the means by which the data will be made available to the certification authority.

f. Schedule: This section describes the means the applicant will use to provide the certification authority with visibility of the activities of the software life cycle processes so reviews can be planned.

g. Additional considerations: This section describes specific considerations that may affect the certification process. Examples include alternative methods of compliance, tool qualification, previously developed software, option-selectable software, user-modifiable software, deactivated code, COTS software, field-loadable software, parameter data items, multiple-version dissimilar software, and product service history.

h. Supplier oversight: This section describes the means of ensuring that supplier processes and outputs will comply with approved software plans and standards.

11.2 SOFTWARE DEVELOPMENT PLAN

The Software Development Plan (SDP) is a description of the software development procedures and software life cycle(s) to be used to satisfy the software development process objectives. It may be included in the Plan for Software Aspects of Certification. This plan should include:

a. Standards: Identification of the Software Requirements Standards, Software Design Standards, and Software Code Standards for the project. Also, references to the standards for previously developed software, including COTS software, if those standards are different.

b. Software life cycle: A description of the software life cycle processes to be used to form the specific software life cycle(s) to be used on the project, including the transition criteria for the software development processes. This description is distinct from the summary provided in the Plan for Software Aspects of Certification, in that it provides the detail necessary to ensure proper implementation of the software life cycle processes.

c. Software development environment: A statement of the chosen software development environment in terms of hardware and software, including:

   1. The requirements development method(s) and tools to be used.

   2. The design method(s) and tools to be used.

   3. The coding method(s), programming language(s), coding tool(s) to be used, and when applicable, options and constraints of autocode generators.

   4. The compilers, linkage editors, and loaders to be used.

   5. The hardware platforms for the tools to be used.

11.3 SOFTWARE VERIFICATION PLAN

The Software Verification Plan (SVP) is a description of the verification procedures to be used to satisfy the software verification process objectives. These procedures may vary by software level as defined in the tables of Annex A. This plan should include:

a. Organization: Organizational responsibilities within the software verification process and interfaces with the other software life cycle processes.

b. Independence: A description of the methods for establishing verification independence, when required.

c. Verification methods: A description of the verification methods to be used for each activity of the software verification process:

   1. Review methods, including checklists or other aids.

   2. Analysis methods, including traceability and coverage analysis.

   3. Testing methods, including the method for selecting test cases, the test procedures to be used, and the test data to be produced.

d. Verification environment: A description of the equipment for testing, the testing and analysis tools, and how to apply these tools and hardware test equipment. Section 4.4.3b provides guidance on indicating target computer and simulator or emulator differences.

e. Transition criteria: The transition criteria for entering the software verification process.

f. Partitioning considerations: If partitioning is used, the methods used to verify the integrity of the partitioning.

g. Compiler assumptions: A description of the assumptions made by the applicant about the correctness of the compiler, linkage editor, or loader (see 4.4.2).

h. Reverification method: For software modification, a description of the methods for identifying, analyzing, and verifying the affected areas of the software and the changed parts of the Executable Object Code.

i. Previously developed software: For previously developed software, if the initial compliance baseline for the verification process does not comply with this document, a description of the methods to satisfy the objectives of this document.

j. Multiple-version dissimilar software: If multiple-version dissimilar software is used, a description of the software verification process activities (see 12.3.2).

11.4 SOFTWARE CONFIGURATION MANAGEMENT PLAN

The Software Configuration Management Plan establishes the methods to be used to achieve the objectives of the SCM process throughout the software life cycle. This plan should include:

a. Environment: A description of the SCM environment to be used, including procedures, tools, methods, standards, organizational responsibilities, and interfaces.

b. Activities: A description of the SCM process activities in the software life cycle:

1. Configuration identification: Items to be identified, when they will be identified, the identification methods for software life cycle data (for example, part numbering), and the relationship of software identification and the system or equipment identification.

2. Baselines and traceability: The means of establishing baselines, what baselines will be established, when these baselines will be established, the software library controls, and the configuration item and baseline traceability.

3. Problem reporting: The content and identification of Problem Reports for the software product and software life cycle processes, when they will be written, the method of closing Problem Reports, and the relationship to the change control activity.

4. Change control: Configuration items and baselines to be controlled, when they will be controlled, the problem/change control activities that control them, pre-certification controls, post-certification controls, and the means of preserving the integrity of baselines and configuration items.

5. Change review: The method of handling feedback from and to the software life cycle processes; the methods of assessing and prioritizing problems, approving changes, and handling their resolution or change implementation; and the relationship of these methods to the problem reporting and change control activities.

6. Configuration status accounting: The data to be recorded to enable reporting configuration management status, definition of where that data will be kept, how it will be retrieved for reporting, and when it will be available.

7. Archive, retrieval, and release: The integrity controls, the release method and authority, and data retention.

8. Software load control: A description of the software load control safeguards and records.

9. Software life cycle environment controls: Controls for the tools used to develop, build, verify, and load the software, addressing sections 11.4.b.1 through 11.4.b.7. This includes control of tools to be qualified.

10. Software life cycle data controls: Controls associated with CC1 and CC2 data.

c. Transition criteria: The transition criteria for entering the SCM process.

d. SCM data: A definition of the software life cycle data produced by the SCM process, including SCM Records, the Software Configuration Index, and the Software Life Cycle Environment Configuration Index.

e. Supplier control: The means of applying SCM process requirements to suppliers.

11.5 SOFTWARE QUALITY ASSURANCE PLAN

The Software Quality Assurance Plan establishes the methods to be used to achieve the objectives of the SQA process. The SQA Plan may include descriptions of process improvement, metrics, and progressive management methods. This plan should include:

a. Environment: A description of the SQA environment, including scope, organizational responsibilities and interfaces, standards, procedures, tools, and methods.

b. Authority: A statement of the SQA authority, responsibility, and independence, including the SQA approval of software products.

c. Activities: The SQA activities that are to be performed for each software life cycle process and throughout the software life cycle including:

   1. SQA methods, for example, reviews, audits, reporting, inspections, and monitoring of the software life cycle processes.

   2. Activities related to the problem reporting, tracking, and corrective action system.

   3. A description of the software conformity review activity.

d. Transition criteria: The transition criteria for entering the SQA process.

e. Timing: The timing of the SQA process activities in relation to the activities of the software life cycle processes.

f. SQA Records: A definition of the records to be produced by the SQA process.

g. Supplier oversight: A description of the means of ensuring that suppliers' processes and outputs will comply with the plans and standards.

11.6 SOFTWARE REQUIREMENTS STANDARDS

Software Requirements Standards define the methods, rules, and tools to be used to develop the high-level requirements. These standards should include:

a. The methods to be used for developing software requirements, such as structured methods.

b. Notations to be used to express requirements, such as data flow diagrams and formal specification languages.

c. Constraints on the use of the tools used for requirements development.

d. The method to be used to provide derived requirements to the system processes.

11.7 SOFTWARE DESIGN STANDARDS

Software Design Standards define the methods, rules, and tools to be used to develop the software architecture and low-level requirements. These standards should include:

a. Design description method(s) to be used.

b. Naming conventions to be used.

c. Conditions imposed on permitted design methods, for example, scheduling, and the use of interrupts and event-driven architectures, dynamic tasking, re-entry, global data, and exception handling, and rationale for their use.

d. Constraints on the use of the design tools.

e. Constraints on design, for example, exclusion of recursion, dynamic objects, data aliases, and compacted expressions.

f. Complexity restrictions, for example, maximum level of nested calls or conditional structures, use of unconditional branches, and number of entry/exit points of code components.

11.8 SOFTWARE CODE STANDARDS

Software Code Standards define the programming languages, methods, rules, and tools to be used to code the software. These standards should include:

a. Programming language(s) to be used and/or defined subset(s). For a programming language, reference the data that unambiguously defines the syntax, the control behavior, the data behavior, and side-effects of the language. This may require limiting the use of some features of a language.

b. Source Code presentation standards, for example, line length restriction, indentation, and blank line usage and Source Code documentation standards, for example, name of author, revision history, inputs and outputs, and affected global data.

c. Naming conventions for components, subprograms, variables, and constants.

d. Conditions and constraints imposed on permitted coding conventions, such as the degree of coupling between software components and the complexity of logical or numerical expressions and rationale for their use.

e. Constraints on the use of the coding tools.

11.9 SOFTWARE REQUIREMENTS DATA

Software Requirements Data is a definition of the high-level requirements including the derived requirements. This data should include:

a. Description of the allocation of system requirements to software, with attention to safety-related requirements and potential failure conditions.

b. Functional and operational requirements under each mode of operation.

c. Performance criteria, for example, precision and accuracy.

d. Timing requirements and constraints.

e. Memory size constraints.

f. Hardware and software interfaces, for example, protocols, formats, frequency of inputs, and frequency of outputs.

g. Failure detection and safety monitoring requirements.

h. Partitioning requirements allocated to software, how the partitioned software components interact with each other, and the software level(s) of each partition.

11.10 DESIGN DESCRIPTION

The Design Description is a definition of the software architecture and the low-level requirements that will satisfy the high-level requirements. This data should include:

a. A detailed description of how the software satisfies the specified high-level requirements, including algorithms, data structures, and how software requirements are allocated to processors and tasks.

b. The description of the software architecture defining the software structure to implement the requirements.

c. The input/output description, for example, a data dictionary, both internally and externally throughout the software architecture.

d. The data flow and control flow of the design.

e. Resource limitations, the strategy for managing each resource and its limitations, the margins, and the method for measuring those margins, for example, timing and memory.

f. Scheduling procedures and inter-processor/inter-task communication mechanisms, including time-rigid sequencing, preemptive scheduling, Ada rendezvous, and interrupts.

g. Design methods and details for their implementation, for example, software loading, user-modifiable software, or multiple-version dissimilar software.

h. Partitioning methods and means of preventing partition breaches.

i. Descriptions of the software components, whether they are new or previously developed, and, if previously developed, reference to the baseline from which they were taken.

j. Derived requirements resulting from the software design process.

k. If the system contains deactivated code, a description of the means to ensure that the code cannot be enabled in the target computer.

l. Rationale for those design decisions that are traceable to safety-related system requirements.

11.11 SOURCE CODE

This data consists of code written in source language(s). The Source Code is used with the compiling, linking, and loading data in the integration process to develop the integrated system or equipment. For each Source Code component, this data should include the software identification, including the name and date of revision and/or version, as applicable.

11.12 EXECUTABLE OBJECT CODE

The Executable Object Code consists of a form of code that is directly usable by the processing unit of the target computer and is, therefore, the software that is loaded into the hardware or system.

11.13 SOFTWARE VERIFICATION CASES AND PROCEDURES

Software Verification Cases and Procedures detail how the software verification process activities are implemented. This data should include descriptions of the:

a. Review and analysis procedures: The scope and depth of the review or analysis methods to be used, in addition to the description in the Software Verification Plan.

b. Test cases: The purpose of each test case, set of inputs, conditions, expected results to achieve the required coverage criteria, and the pass/fail criteria.

c. Test procedures: The step-by-step instructions for how each test case is to be set up and executed, how the test results are evaluated, and the test environment to be used.

11.14 SOFTWARE VERIFICATION RESULTS

The Software Verification Results are produced by the software verification process activities. Software Verification Results should:

a. For each review, analysis, and test, indicate each procedure that passed or failed during the activities and the final pass/fail results.

b. Identify the configuration item or software version reviewed, analyzed, or tested.

c. Include the results of tests, reviews, and analyses, including coverage analyses and traceability analyses.

Any discrepancies found should be recorded and tracked via problem reporting.

Additionally, evidence provided in support of the system processes' assessment of information provided by the software processes (see 2.2.1.f and 2.2.1.g) should be considered to be Software Verification Results.

11.15 SOFTWARE LIFE CYCLE ENVIRONMENT CONFIGURATION INDEX

The Software Life Cycle Environment Configuration Index (SECI) identifies the configuration of the software life cycle environment. This index is written to aid reproduction of the hardware and software life cycle environment for software regeneration, reverification, or software modification, and should:

a. Identify the software life cycle environment hardware and its operating system software.

b. Identify the tools to be used during the development of the software. Examples include compilers, linkage editors, loaders, data integrity tools such as tools that calculate and embed checksums or cyclical redundancy checks, and any autocode generator with its associated options.

c. Identify the test environment used to verify the software product, for example, the software testing and analysis tools.

d. Identify qualified tools and their associated tool qualification data.

NOTE: *This data may be included in the Software Configuration Index.*

11.16 SOFTWARE CONFIGURATION INDEX

The Software Configuration Index (SCI) identifies the configuration of the software product. Specific configuration identifiers and version identifiers should be provided.

NOTE: *The SCI can contain one data item or a set (hierarchy) of data items. The SCI can contain the items listed below or it may reference another SCI or other configuration identified data that specifies the individual items and their versions.*

The SCI should identify:

a. The software product.

b. Executable Object Code and Parameter Data Item Files, if any.

c. Each Source Code component.

d. Previously developed software in the software product, if used.

e. Software life cycle data.

f. Archive and release media.

g. Instructions for building the Executable Object Code and Parameter Data Item Files, if any, including, for example, instructions and data for compiling and linking; and the procedures used to recover the software for regeneration, testing, or modification.

h. Reference to the Software Life Cycle Environment Configuration Index (see 11.15), if it is packaged separately.

i. Data integrity checks for the Executable Object Code, if used.

j. Procedures, methods, and tools for making modifications to the user-modifiable software, if any.

k. Procedures and methods for loading the software into the target hardware.

NOTE: *The SCI may be produced for one software product version or it may be extended to contain data for several alternative or successive software product versions.*

11.17 PROBLEM REPORTS

Problem Reports are a means to identify and record the resolution to software product anomalous behavior, process non-compliance with software plans and standards, and deficiencies in software life cycle data. Problem Reports should include:

a. Identification of the configuration item and/or the software life cycle process activity in which the problem was observed.

b. Identification of the configuration item(s) to be modified or a description of the process to be changed.

c. A problem description that enables the problem to be understood and resolved. The problem description should contain sufficient detail to facilitate the assessment of the potential safety or functional effects of the problem.

d. A description of the corrective action taken to resolve the reported problem.

11.18 SOFTWARE CONFIGURATION MANAGEMENT RECORDS

The results of the SCM process activities are recorded in SCM Records. Examples include configuration identification lists, baseline or software library records, change history reports, archive records, and release records. These examples do not imply that records of these specific types need to be produced.

NOTE: *Due to the integral nature of the SCM process, its outputs will often be included as parts of other software life cycle data.*

11.19 SOFTWARE QUALITY ASSURANCE RECORDS

The results of the SQA process activities are recorded in SQA Records. These may include SQA review or audit reports, meeting minutes, records of authorized process deviations, or software conformity review records.

11.20 SOFTWARE ACCOMPLISHMENT SUMMARY

The Software Accomplishment Summary is the primary data item for showing compliance with the Plan for Software Aspects of Certification. This summary should include:

a. System overview: This section provides an overview of the system, including a description of its functions and their allocation to hardware and software, the architecture, the processor(s) used, the hardware/software interfaces, and safety features. This section also describes any differences from the system overview in the Plan for Software Aspects of Certification.

b. Software overview: This section briefly describes the software functions with emphasis on the safety and partitioning concepts used, and explains differences from the software overview proposed in the Plan for Software Aspects of Certification.

c. Certification considerations: This section restates the certification considerations described in the Plan for Software Aspects of Certification and describes any differences.

d. Software life cycle: This section summarizes the actual software life cycle(s) and explains differences from the software life cycle and software life cycle processes proposed in the Plan for Software Aspects of Certification.

e. Software life cycle data: This section describes any differences from the proposals made in the Plan for Software Aspects of Certification for the software life cycle data produced, the relationship of the data to each other and to other data defining the system, and the means by which the data was made available to the certification authority. This section explicitly references, by configuration identifiers and version, the applicable Software Configuration Index and Software Life Cycle Environment Configuration Index. Detailed information regarding configuration identifiers and specific versions of software life cycle data is provided in the Software Configuration Index.

f. Additional considerations: This section summarizes any specific considerations that may warrant the attention of the certification authority. It explains any differences from the proposals contained in the Plan for Software Aspects of Certification regarding such considerations. Reference should be made to data items applicable to these matters, such as issue papers or special conditions.

g. Supplier oversight: This section describes how supplier processes and outputs comply with plans and standards.

h. Software identification: This section identifies the software configuration by part number and version.

i. Software characteristics: This section states the Executable Object Code size, timing margins including worst-case execution time, memory margins, resource limitations, and the means used for measuring each characteristic.

j. Change history: If applicable, this section includes a summary of software changes with attention to changes made due to failures affecting safety, and identifies any changes from and improvements to the software life cycle processes since the previous certification.

k. Software status: This section contains a summary of Problem Reports unresolved at the time of certification. The Problem Report summary includes a description of each problem and any associated errors, functional limitations, operational restrictions, potential adverse effect(s) on safety together with a justification for allowing the Problem Report to remain open, and details of any mitigating action that has been or needs to be carried out.

l. Compliance statement: This section includes a statement of compliance with this document and a summary of the methods used to demonstrate compliance with criteria specified in the software plans. This section also addresses additional rulings made by the certification authority and any deviations from the software plans, standards, and this document not covered elsewhere in the Software Accomplishment Summary.

11.21 TRACE DATA

Trace Data establishes the associations between life cycle data items contents. Trace Data should be provided that demonstrates bi-directional associations between:

a. System requirements allocated to software and high-level requirements.

b. High-level requirements and low-level requirements.

c. Low-level requirements and Source Code.

d. Software Requirements and test cases.

e. Test cases and test procedures.

f. Test procedures and test results.

11.22 PARAMETER DATA ITEM FILE

The Parameter Data Item File consists of a form of data that is directly usable by the processing unit of the target computer.

Software life cycle data should be produced for each instantiation of a Parameter Data Item. If packaged separately, this data should include a reference to the Software Accomplishment Summary of the associated Executable Object Code.

CHAPTER 12

ADDITIONAL CONSIDERATIONS

The previous sections of this document provide guidance for satisfying certification requirements in which the applicant submits evidence of the software life cycle processes as described in those sections. This section provides guidance on additional considerations where objectives and/or activities may replace, modify, or add to some or all of the objectives and/or activities defined in the rest of this document. The use of additional considerations and the proposed impact on the guidance provided in the other sections of this document should be agreed on a case-by-case basis with the certification authorities.

12.1 USE OF PREVIOUSLY DEVELOPED SOFTWARE

The guidance of this section discusses the issues associated with the use of previously developed software, including the assessment of modifications; the effect of changing an aircraft installation, application environment, or development environment; upgrading a development baseline; and SCM and SQA considerations. The intention to use previously developed software is stated in the Plan for Software Aspects of Certification. Unresolved Problem Reports associated with the previously developed software should be evaluated for impact.

12.1.1 Modifications to Previously Developed Software

This guidance discusses modifications to previously developed software where the outputs of the previous software life cycle processes comply with this document. Modification may result from requirement changes, the detection of errors, and/or software enhancements.

Activities include:

a. The revised outputs of the system safety assessment process should be reviewed considering the proposed modifications.

b. If the software level is revised, the guidance of section 12.1.4 should be considered.

c. Both the impact of the software requirements changes and the impact of software architecture changes should be analyzed, including the consequences of software requirement changes upon other requirements and the coupling between several software components that may result in reverification effort involving more than the modified area.

d. The area affected by a change should be determined. This may be done by data flow analysis, control flow analysis, timing analysis, traceability analysis, or a combination of these analyses.

e. Areas affected by the change should be reverified in accordance with section 6.

12.1.2 Change of Aircraft Installation

Airborne systems or equipment containing software that has been previously "approved" at a certain software level and under a specific certification basis may be used in a new aircraft installation. Activities include:

a. The system safety assessment process assesses the new aircraft installation and determines the software level and the certification basis. No additional effort will be required if these are the same for the new installation as they were in the previous installation.

b. If functional modifications are required for the new installation, the guidance of section 12.1.1 should be satisfied.

c. If the previous development activity did not produce outputs required to substantiate the system safety objectives of the new installation, the guidance of section 12.1.4 should be satisfied.

12.1.3 Change of Application or Development Environment

Use and modification of previously developed software may involve a new development environment, a new target processor or other hardware, or integration with other software than that used for the original application.

New development environments may increase or reduce some activities within the software life cycle. New application environments may require activities in addition to software life cycle process activities that address modifications.

Changes to an application or development environment should be identified, analyzed, and reverified.

Activities include:

a. If a new development environment uses software tools, the guidance of section 12.2 may be applicable.

b. The rigor of the evaluation of an application change should consider the complexity and sophistication of the programming language. For example, the rigor of the evaluation for Ada generics will be greater if the generic parameters are different in the new application. For object-oriented languages, the rigor will be greater if the objects that are inherited are different in the new application.

c. Using a different autocode generator or a different set of autocode generator options may change the Source Code or object code generated. The impact of any changes should be analyzed.

d. If a different compiler or different set of compiler options are used, resulting in different object code, the results from a previous software verification process activity using the object code may not be valid and should not be used for the new application. In this case, previous test results may no longer be valid for the structural coverage criteria of the new application. Similarly, compiler assumptions about optimization may not be valid.

e. If a different processor is used, then a change impact analysis is performed to determine:

    1. Software components that are new or will need to be modified as a result of changing the processor, including any modification for hardware/software integration.

    2. The results from a previous software verification process activity directed at the hardware/software interface that may be used for the new application.

    3. Previous hardware/software integration tests that should be executed for the new application. It is expected that there will always be a minimal set of tests to be run.

    4. Additional hardware/software integration tests and reviews that may be necessary.

f. If a hardware item, other than the processor, is changed and the design of the software isolates the interfacing modules from other modules then a change impact analysis should be performed to:

    1. Determine the software modules or interfaces that are new or will be modified to accommodate the changed hardware component.

    2. Determine the extent of reverification required.

g. Verification of software interfaces should be conducted where previously developed software is used with different interfacing software. A change impact analysis may be used to determine the extent of reverification required.

12.1.4 Upgrading a Development Baseline

Guidance follows for software whose software life cycle data from a previous application are determined to be inadequate or do not satisfy the objectives of this document, due to the requirements associated with a new application. This guidance is intended to aid in satisfying the objectives of this document when applied to:

- COTS software.
- Airborne software developed to other guidance.
- Airborne software developed prior to the existence of this document.
- Software previously developed to this document at a lower software level.

Activities for upgrading a development baseline include:

a. The objectives of this document should be satisfied while taking advantage of software life cycle data of the previous development that satisfy the objectives for the new application.

b. Software aspects of certification should be based on the failure conditions and software level(s) as determined by the system safety assessment process. Comparison to failure conditions of the previous application will determine areas that may need to be upgraded.

c. Software life cycle data from a previous development should be evaluated to ensure that the software verification process objectives of the software level are satisfied for the new application to the necessary level of rigor and independence.

d. Reverse engineering may be used to regenerate software life cycle data that is inadequate or missing in satisfying the objectives of this document. In addition to producing the software product, additional activities may need to be performed to satisfy the software verification process objectives.

e. If use of product service history is planned to satisfy the objectives of this document in upgrading a development baseline, section 12.3.4 should be considered.

f. The applicant should specify the strategy for accomplishing compliance with this document in the Plan for Software Aspects of Certification.

12.1.5 Software Configuration Management Considerations

If previously developed software is used, the software configuration management process activities for the new application should include, in addition to the activities of section 7:

a. Providing traceability from the software product and software life cycle data of the previous application to the new application.

b. Providing change control that enables problem reporting, problem resolution, and tracking of changes to software components used in more than one application.

12.1.6 Software Quality Assurance Considerations

If previously developed software is used, the software quality assurance activities should include, in addition to the activities of section 8:

a. Providing assurance that the software components satisfy or exceed the software life cycle criteria of the software level for the new application.

b. Providing assurance that changes to the software life cycle processes are stated in the software plans.

12.2 TOOL QUALIFICATION

12.2.1 Determining if Tool Qualification is Needed

Qualification of a tool is needed when processes of this document are eliminated, reduced, or automated by the use of a software tool without its output being verified as specified in section 6.

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.

The tool qualification process may be applied to a single tool, a collection of tools, or one or more functions within a tool. For a tool with multiple functions, if protection of tool functions can be demonstrated, only those functions that are used to eliminate, reduce or automate software life cycle processes, and whose outputs are not verified, need be qualified. Protection is the use of a mechanism to ensure that a tool function cannot adversely impact another tool function.

A tool is qualified only for use on a specific system where the intention to use the tool is stated in the Plan for Software Aspects of Certification that supports the system. If a tool previously qualified on one system is proposed for use on another system, it should be re-qualified within the context of that other system.

12.2.2 Determining the Tool Qualification Level

If tool qualification is needed, the impact of the tool use in the software life cycle processes should be assessed in order to determine its tool qualification level (TQL). The following criteria should be used to determine the impact of the tool:

a. Criteria 1: A tool whose output is part of the airborne software and thus could insert an error.

b. Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:

   1. Verification process(es) other than that automated by the tool, or

   2. Development process(es) that could have an impact on the airborne software.

c. Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error.

If the tool eliminates, reduces, or automates processes in this document and its output is not verified as specified in section 6, the appropriate TQL is as shown in Table 12-1. Five levels of tool qualification, TQL-1 to TQL-5, are identified based on the tool use and its potential impact in the software life cycle processes. TQL-1 is the most rigorous level and TQL-5 is the least rigorous level. When assessing the impact of a given tool, the criteria should be considered sequentially from criteria 1 to criteria 3. The tool qualification level should be coordinated with the certification authority as early as possible.

TABLE 12-1: TOOL QUALIFICATION LEVEL DETERMINATION

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

12.2.3 Tool Qualification Process

The objectives, activities, guidance, and life cycle data required for each Tool Qualification Level are described in ED-215, "Software Tool Qualification Considerations."

12.3 ALTERNATIVE METHODS

Some methods were not discussed in the previous sections of this document because of inadequate maturity at the time this document was written or limited applicability for airborne software. It is not the intention of this document to restrict the implementation of any current or future methods. Any single alternative method discussed in this section may be used in satisfying one or more of the objectives in this document. In addition, alternative methods may be used to support one another.

An alternative method cannot be considered in isolation from the suite of software development processes. The effort for obtaining certification credit of an alternative method is dependent on the software level and the impact of the alternative method on the software life cycle processes. Guidance for using an alternative method includes:

a. An alternative method should be shown to satisfy the objectives of this document or the applicable supplement.

b. The applicant should specify in the Plan for Software Aspects of Certification, and obtain agreement from the certification authority for:

   1. The impact of the proposed method on the software development processes.

   2. The impact of the proposed method on the software life cycle data.

   3. The rationale for use of the alternative method that shows that the system safety objectives are satisfied. One technique for presenting the rationale for using an alternative method is an assurance case, in which arguments are explicitly given to link the evidence to the claims of compliance with the system safety objectives.

c. The rationale should be substantiated by software plans, processes, expected results, and evidence of the use of the method.

12.3.1 Exhaustive Input Testing

There are situations where the software component of an airborne system or equipment is simple and isolated such that the set of inputs and outputs can be bounded. If so, it may be possible to demonstrate that exhaustive testing of this input space can be substituted for one or more of the software verification process activities identified in section 6.

For this alternative method, activities include:

a. Defining the complete set of valid inputs and outputs of the software.

b. Performing an analysis that confirms the isolation of the inputs to the software.

c. Developing rationale for the exhaustive input test cases and procedures.

d. Developing the test cases, test procedures, and test results.

12.3.2 Considerations for Multiple-Version Dissimilar Software Verification

Guidance follows concerning the software verification process as it applies to multiple-version dissimilar software. If the software verification process is modified because of the use of multiple-version dissimilar software, evidence should be provided that the software verification process objectives are satisfied and that equivalent error detection is achieved for each software version.

Multiple, dissimilar versions of the software are produced using combinations of these techniques:

- The Source Code is implemented in two or more different programming languages.
- The object code is generated using two or more different compilers.
- Each software version of Executable Object Code executes on a separate, dissimilar processor, or on a single processor with the means to provide partitioning between the software versions.
- The software requirements, software design, and/or Source Code are developed by two or more development teams whose interactions are managed.
- The software requirements, software design, and/or Source Code are developed in two or more software development environments, and/or each version is verified using separate test environments.
- The Executable Object Code is linked and loaded using two or more different linkage editors and two or more different loaders.
- The software requirements, software design, and/or Source Code are developed in conformance with two or more different Software Requirements Standards, Software Design Standards, and/or Software Code Standards, respectively.

When multiple versions of software are used, the software verification methods may be modified from those used to verify single version software. They will apply to software development process activities that are multi-thread, such as separate, multiple development teams. The software verification process is dependent on the combined hardware and software architectures since this affects the dissimilarity of the multiple software versions. Additional software verification process objectives to be satisfied are to demonstrate that:

a. Inter-version compatibility requirements are satisfied, including compatibility during normal and abnormal operations and state transitions.

b. Equivalent error detection is achieved.

Other changes in software verification process activities may be agreed with the certification authority, if the changes are substantiated by rationale that confirms equivalent software verification coverage.

12.3.2.1 Independence of Multiple-Version Dissimilar Software

When multiple-version dissimilar software versions are developed independently using a managed method, the development processes have the potential to reveal certain classes of errors such that verification of each software version is equivalent to independent verification of the software development processes. Activities include:

a. The applicant should demonstrate that different teams with limited interaction developed each software version's software requirements, software design, and Source Code.

b. Independent test coverage analyses should still be performed as with a single version.

NOTE: *Section 12.3.2.1 only addresses the subject of independence. Reduction of software levels is not discussed or intended.*

12.3.2.2 Multiple Processor-Related Verification

When each version of dissimilar software executes on a different type of processor, the verification of some aspects of compatibility of the code with the processor (see 6.4.3.a) may be replaced by verification to ensure that the multiple types of processor produce the correct outputs. This verification consists of integration tests in which the outputs of the multiple versions are cross-compared in requirements-based test cases. The applicant should complete the following activities:

a. Show that equivalent error detection is achieved.

b. Show that each processor was designed by a different developer.

c. Show that the outputs of the multiple versions are equivalent.

12.3.2.3 Multiple-Version Source Code Verification

The guidance for structural coverage analysis (see 6.4.4.2) may be modified for systems or equipment using multiple-version dissimilar software. For structural coverage analysis, the accompanying Level A activity (see 6.4.4.2.b) to evaluate any additional code (generated by a compiler, linker, or other means) that is not directly traceable to Source Code statements need not be done provided that the applicant completes the following activities:

a. Show that each version of software is coded using a different programming language.

b. Show that each compiler used is from a different developer.

12.3.2.4 Tool Qualification for Multiple-Version Dissimilar Software

If multiple-version dissimilar software is used, the tool qualification process may be modified, if evidence is available that the multiple software tools used in the software development process are dissimilar. This depends on the demonstration of equivalent software verification process activity in the development of the multiple software versions using dissimilar software tools. The applicant should complete the following activities:

a. Show that each tool was obtained from a different developer.

b. Show that each tool has a dissimilar design.

12.3.2.5 Multiple Simulators and Verification

If separate, dissimilar simulators are used to verify multiple-version dissimilar software versions, then the approach to tool qualification of the simulators may be modified. This depends on the demonstration of equivalent software verification process activity in the simulation of the multiple software versions using multiple simulators. Unless it can be justified as unnecessary for multiple simulators to be dissimilar, the applicant should complete the following activities:

a. Provide evidence that each simulator was developed by a different team.

b. Provide evidence that each simulator has different requirements, a different design, and a different programming language.

c. Provide evidence that each simulator executes on a different processor.

NOTE: *When a multiple processor system using multiple, dissimilar versions of software are executing on identical processors, it may be difficult to demonstrate dissimilarity of simulators because of the reliance on information obtained from a common source, for example, the processor manufacturer.*

12.3.3 Software Reliability Models

Many methods for predicting software reliability based on developmental metrics have been published, for example, software structure, defect detection rate, etc. This document does not provide guidance for those types of methods, because at the time of writing, currently available methods did not provide results in which confidence can be placed.

12.3.4 Product Service History

If equivalent safety for the software can be demonstrated by the use of the software's product service history, some certification credit may be granted. The acceptability of this method is dependent on:

- Configuration management of the software.
- Effectiveness of problem reporting activity.
- Stability and maturity of the software.
- Relevance of product service history environment.
- Length of the product service history.
- Actual error rates in the product service history.
- Impact of modifications.

Use of service history data for certification credit is predicated upon sufficiency, relevance, and types of problems occurring during the service history period. The use, conditions of use, and results of software service history should be defined, assessed by the system processes, including the system safety assessment process, and submitted to the appropriate certification authority. Guidance for determining applicability of service history and the length of service history needed is presented below.

12.3.4.1 Relevance of Service History

The following applies for establishing the relevance of service history:

a. Type of service history: Service history to be used should be defined and agreement obtained from the certification authority. Service history data should be provided using a measure relevant to the operations of the system. For example, flight hours is an appropriate measure for software that is used continuously during flight, such as flight control software, and the number of demands is an appropriate measure for software that is executed on demand, such as landing gear software.

b. Known configuration: The applicant should show that the software and associated evidence used to comply with system safety objectives have been under configuration management throughout the product service history.

c. Operating time collection process: The applicant should show that the means of collecting and calculating flight hours for software that is used continuously during flight or number of demands for software that is executed on demand is sufficiently accurate and complete. Flight hours or number of demands should account for changes in any factors that are important to the intended application including but not limited to:

   1. Software and system configuration.
   2. Operational mode or state.
   3. Operating environment.

d. Changes to the software: Configuration changes during the product service history should be identified. An analysis should be conducted to determine whether the changes made to the software alter the applicability of the service history data for the period preceding the changes.

e. Usage and Environment: The intended software usage should be analyzed to show the relevance of the product service history.

   1. It should be assured that software capabilities to be used are exercised in all operational modes.
   2. Analysis should also be performed to assure that relevant permutations of input data are executed.
   3. The operating environment used to collect the service history data should be assessed to show relevance to the intended use in the proposed application. If the operating environments of the existing and proposed applications differ, additional verification should confirm compliance with the system safety objectives in the target environment.
   4. If credit is being sought for compatibility with the hardware environment, then the relationship between the service history environment and the intended environment should be addressed. The impact of any hardware modifications during the service history period should also be assessed.

f. Deactivated code: Analysis should be performed to show that any code that was deactivated during the period of service history is not activated in the new environment. If it is found that previously deactivated code is activated in the new environment, additional verification should be conducted.

12.3.4.2 Sufficiency of Accumulated Service History

The required amount of service history is determined by:

a. The system safety objectives of the software and the software level.

b. Any differences in service history environment and system operational environment.

c. The objectives from sections 4 to 9 being addressed by service history.

d. Evidence, in addition to service history, addressing those objectives.

12.3.4.3 Collection, Reporting, and Analysis of Problems Found During Service History

a. Problem reporting process: The applicant should show that problem reporting during the product service history period provides assurance that representative data is available and that problems during the service history period were reported and recorded, and are retrievable.

1. The specific data to be collected should be agreed on with the certification authority and should include the following for each recorded problem, in addition to the items in section 11.17:

i. The hardware/software configuration in effect when the problem occurred.

ii. The operating environment within which the problem occurred.

iii. The operating mode or state within which the problem occurred.

iv. Any application-specific information needed for problem assessment.

v. Classification of the problem with respect to severity, safety significance, and whether the problem was the result of a change in the software configuration since the start of service history data collection.

vi. Assessment of whether the problem was:

- Reproducible.
- Recoverable.
- Related to other previously reported problems, including, but not limited to, a common cause.

2. The chronological trend of Problem Reports should be evaluated and any increasing trend explained.

3. The completeness of the software's error history should also be addressed. This includes:

i. The ability to detect faults and maintain a fault log.

ii. The means for operators to report problems.

iii. The completeness of the problem reporting records.

iv. The means for the applicant to determine the system safety impact of any open Problem Reports, and to determine whether problems that were not safety-related during the service history will be safety-related in the intended environment. Problems that were not safety-related in the service experience environment, but which will be safety-related in the intended environment, might indicate the need for additional verification.

v. The means for the applicant to determine the number of occurrences of a specific problem.

b. Process-related problems: Those problems that are indicative of an inadequate process, such as design or code errors, should be indicated separately from those whose cause are outside the scope of this document, such as hardware or system requirements errors.

c. Safety-related problems: All in-service problems should be evaluated to determine which problems were safety-related, and to confirm that all safety-related problems have been corrected.

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

The following items should be specified in the Plan for Software Aspects of Certification and agreed with the certification authority when seeking certification credit for service history:

a. Rationale for claiming relevant service history, addressing the items in section 12.3.4.1.

b. Amount of service history needed together with the rationale. This should include the items in section 12.3.4.2, any censoring rules for data used in estimation, and measured parameters, if applicable. This data should be provided using measures relevant to the operations of the system.

c. Rationale for calculating the total relevant service history period, including factors such as operational modes, the number of independently operating copies in the installation and in service, and the definition of "normal operation" and "normal operation time."

NOTE: *If the error rate is greater than that identified in the plan, these errors should be analyzed and the analyses reviewed with the certification authority. The length of the service history period may need to be extended or service history may be inapplicable as an alternative means of compliance.*

d. Definition of what was counted as an error and rationale for that definition. This should address the items in section 12.3.4.3a.

e. Proposed acceptable error rates and rationale for the product service history period in relation to the system safety and proposed error rates. This should address the items in sections 12.3.4.3b and 12.3.4.3c.

f. Definition of criteria for problems that would invalidate service history under section 12.3.4.3 or for other reasons.

g. Criteria for errors that will be corrected; how they will be corrected and verified; and rationale for any defects for which no action will be taken.

h. Objectives in sections 4 to 9 to be addressed through the use of service history.

ANNEX A

PROCESS OBJECTIVES AND OUTPUTS BY SOFTWARE LEVEL

References in these tables point to those sections in the text that define the particular objectives, related activities, and outputs.

The tables include guidance for:

a. The process objectives applicable for each software level. For level E software, see 2.3.3.

b. The independence by software level of the software life cycle process activities applicable to satisfy that process's objectives.

c. The control category by software level for the software life cycle data produced by the software life cycle process activities (see 7.3).

These tables should not be used as a checklist. These tables do not reflect all aspects of compliance to this document. In order to fully understand the guidance, the full body of this document should be considered.

The following legend applies to “Applicability by Software Level” and “Control Category by Software Level” for all tables:

LEGEND:	●	The objective should be satisfied with independence.
	❍	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant’s discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

TABLE A-1: SOFTWARE PLANNING PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	The activities of the software life cycle processes are defined.	4.1.a	4.2.a 4.2.c 4.2.d 4.2.e 4.2.g 4.2.i 4.2.l 4.3.c	O	O	O	O	PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②
2	The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria, is defined.	4.1.b	4.2i 4.3.b	O	O	O		PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	
3	Software life cycle environment is selected and defined.	4.1.c	4.4.1 4.4.2.a 4.4.2.b 4.4.2.c 4.4.3	O	O	O		PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	
4	Additional considerations are addressed.	4.1.d	4.2.f 4.2.h 4.2.i 4.2.j 4.2.k	O	O	O	O	PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②
5	Software development standards are defined.	4.1.e	4.2.b 4.2.g 4.5	O	O	O		SW Requirements Standards SW Design Standards SW Code Standards	11.6 11.7 11.8	① ① ①	① ① ①	② ② ②	
6	Software plans comply with this document.	4.1.f	4.3.a 4.6	O	O	O		Software Verification Results	11.14	②	②	②	
7	Development and revision of software plans are coordinated.	4.1.g	4.2.g 4.6	O	O	O		Software Verification Results	11.14	②	②	②	

TABLE A-2: SOFTWARE DEVELOPMENT PROCESSES

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	High-level requirements are developed.	5.1.1.a	5.1.2.a 5.1.2.b 5.1.2.c 5.1.2.d 5.1.2.e 5.1.2.f 5.1.2.g 5.1.2.j 5.5.a	O	O	O	O	Software Requirements Data Trace Data	11.9 11.21	① ①	① ①	① ①	① ①
2	Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.	5.1.1.b	5.1.2.h 5.1.2.i	O	O	O	O	Software Requirements Data	11.9	①	①	①	①
3	Software architecture is developed.	5.2.1.a	5.2.2.a 5.2.2.d	O	O	O	O	Design Description	11.10	①	①	①	②
4	Low-level requirements are developed.	5.2.1.a	5.2.2.a 5.2.2.e 5.2.2.f 5.2.2.g 5.2.3.a 5.2.3.b 5.2.4.a 5.2.4.b 5.2.4.c 5.5.b	O	O	O		Design Description Trace Data	11.10 11.21	① ①	① ①	① ①	
5	Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.	5.2.1.b	5.2.2.b 5.2.2.c	O	O	O		Design Description	11.10	①	①	①	
6	Source Code is developed.	5.3.1.a	5.3.2.a 5.3.2.b 5.3.2.c 5.3.2.d 5.5.c	O	O	O		Source Code Trace Data	11.11 11.21	① ①	① ①	① ①	
7	Executable Object Code and Parameter Data Item Files, if any, are produced and loaded in the target computer.	5.4.1.a	5.4.2.a 5.4.2.b 5.4.2.c 5.4.2.d 5.4.2.e 5.4.2.f	O	O	O	O	Executable Object Code Parameter Data Item File	11.12 11.22	① ①	① ①	① ①	① ①

TABLE A-3: VERIFICATION OF OUTPUTS OF SOFTWARE REQUIREMENTS PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	High-level requirements comply with system requirements.	6.3.1.a	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
2	High-level requirements are accurate and consistent.	6.3.1.b	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
3	High-level requirements are compatible with target computer.	6.3.1.c	6.3.1	○	○			Software Verification Results	11.14	②	②		
4	High-level requirements are verifiable.	6.3.1.d	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
5	High-level requirements conform to standards.	6.3.1.e	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
6	High-level requirements are traceable to system requirements.	6.3.1.f	6.3.1	○	○	○	○	Software Verification Results	11.14	②	②	②	②
7	Algorithms are accurate.	6.3.1.g	6.3.1	●	●	○		Software Verification Results	11.14	②	②	②	

TABLE A-4: VERIFICATION OF OUTPUTS OF SOFTWARE DESIGN PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Low-level requirements comply with high-level requirements.	6.3.2.a	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
2	Low-level requirements are accurate and consistent.	6.3.2.b	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
3	Low-level requirements are compatible with target computer.	6.3.2.c	6.3.2	○	○			Software Verification Results	11.14	②	②		
4	Low-level requirements are verifiable.	6.3.2.d	6.3.2	○	○			Software Verification Results	11.14	②	②		
5	Low-level requirements conform to standards.	6.3.2.e	6.3.2	○	○	○		Software Verification Results	11.14	②	②	②	
6	Low-level requirements are traceable to high-level requirements.	6.3.2.f	6.3.2	○	○	○		Software Verification Results	11.14	②	②	②	
7	Algorithms are accurate.	6.3.2.g	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
8	Software architecture is compatible with high-level requirements.	6.3.3.a	6.3.3	●	○	○		Software Verification Results	11.14	②	②	②	
9	Software architecture is consistent.	6.3.3.b	6.3.3	●	○	○		Software Verification Results	11.14	②	②	②	
10	Software architecture is compatible with target computer.	6.3.3.c	6.3.3	○	○			Software Verification Results	11.14	②	②		
11	Software architecture is verifiable.	6.3.3.d	6.3.3	○	○			Software Verification Results	11.14	②	②		
12	Software architecture conforms to standards.	6.3.3.e	6.3.3	○	○	○		Software Verification Results	11.14	②	②	②	
13	Software partitioning integrity is confirmed.	6.3.3.f	6.3.3	●	○	○	○	Software Verification Results	11.14	②	②	②	②

TABLE A-5: VERIFICATION OF OUTPUTS OF SOFTWARE CODING & INTEGRATION PROCESSES

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Source Code complies with low-level requirements.	6.3.4.a	6.3.4	●	●	○		Software Verification Results	11.14	②	②	②	
2	Source Code complies with software architecture.	6.3.4.b	6.3.4	●	○	○		Software Verification Results	11.14	②	②	②	
3	Source Code is verifiable.	6.3.4.c	6.3.4	○	○			Software Verification Results	11.14	②	②		
4	Source Code conforms to standards.	6.3.4.d	6.3.4	○	○	○		Software Verification Results	11.14	②	②	②	
5	Source Code is traceable to low-level requirements.	6.3.4.e	6.3.4	○	○	○		Software Verification Results	11.14	②	②	②	
6	Source Code is accurate and consistent.	6.3.4.f	6.3.4	●	○	○		Software Verification Results	11.14	②	②	②	
7	Output of software integration process is complete and correct.	6.3.5.a	6.3.5	○	○	○		Software Verification Results	11.14	②	②	②	
8	Parameter Data Item File is correct and complete	6.6.a	6.6	●	●	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②
9	Verification of Parameter Data Item File is achieved.	6.6.b	6.6	●	●	○		Software Verification Results	11.14	②	②	②	

TABLE A-6: TESTING OF OUTPUTS OF INTEGRATION PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Executable Object Code complies with high-level requirements.	6.4.a	6.4.2 6.4.2.1 6.4.3 6.5	O	O	O	O	Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	② ② ②
2	Executable Object Code is robust with high-level requirements.	6.4.b	6.4.2 6.4.2.2 6.4.3 6.5	O	O	O	O	Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	② ② ②
3	Executable Object Code complies with low-level requirements.	6.4.c	6.4.2 6.4.2.1 6.4.3 6.5	●	●	O		Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	
4	Executable Object Code is robust with low-level requirements.	6.4.d	6.4.2 6.4.2.2 6.4.3 6.5	●	O	O		Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	
5	Executable Object Code is compatible with target computer.	6.4.e	6.4.1.a 6.4.3.a	O	O	O	O	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②

TABLE A-7: VERIFICATION OF VERIFICATION PROCESS RESULTS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Test procedures are correct.	6.4.5.b	6.4.5	●	○	○		Software Verification Results	11.14	②	②	②	
2	Test results are correct and discrepancies explained.	6.4.5.c	6.4.5	●	○	○		Software Verification Results	11.14	②	②	②	
3	Test coverage of high-level requirements is achieved.	6.4.4.a	6.4.4.1	●	○	○	○	Software Verification Results	11.14	②	②	②	②
4	Test coverage of low-level requirements is achieved.	6.4.4.b	6.4.4.1	●	○	○		Software Verification Results	11.14	②	②	②	
5	Test coverage of software structure (modified condition/decision coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●				Software Verification Results	11.14	②			
6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●	●			Software Verification Results	11.14	②	②		
7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●	●	○		Software Verification Results	11.14	②	②	②	
8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.d	6.4.4.2.c 6.4.4.2.d 6.4.4.3	●	●	○		Software Verification Results	11.14	②	②	②	
9	Verification of additional code, that cannot be traced to Source Code, is achieved.	6.4.4.c	6.4.4.2.b	●				Software Verification Results	11.14	②			

TABLE A-8: SOFTWARE CONFIGURATION MANAGEMENT PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	Configuration items are identified.	7.1.a	7.2.1	O	O	O	O	SCM Records	11.18	②	②	②	②
2	Baselines and traceability are established.	7.1.b	7.2.2	O	O	O	O	Software Configuration Index SCM Records	11.16 11.18	① ②	① ②	① ②	① ②
3	Problem reporting, change control, change review, and configuration status accounting are established.	7.1.c 7.1.d 7.1.e 7.1.f	7.2.3 7.2.4 7.2.5 7.2.6	O	O	O	O	Problem Reports SCM Records	11.17 11.18	② ②	② ②	② ②	② ②
4	Archive, retrieval, and release are established.	7.1.g	7.2.7	O	O	O	O	SCM Records	11.18	②	②	②	②
5	Software load control is established.	7.1.h	7.4	O	O	O	O	SCM Records	11.18	②	②	②	②
6	Software life cycle environment control is established.	7.1.i	7.5	O	O	O	O	Software Life Cycle Environment Configuration Index SCM Records	11.15 11.18	① ②	① ②	① ②	② ②

TABLE A-9: SOFTWARE QUALITY ASSURANCE PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Assurance is obtained that software plans and standards are developed and reviewed for compliance with this document and for consistency.	8.1.a	8.2.b 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
2	Assurance is obtained that software life cycle processes comply with approved software plans.	8.1.b	8.2.a 8.2.c 8.2.d 8.2.f 8.2.h 8.2.i	●	●	●	●	SQA Records	11.19	②	②	②	②
3	Assurance is obtained that software life cycle processes comply with approved software standards.	8.1.b	8.2.a 8.2.c 8.2.d 8.2.f 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
4	Assurance is obtained that transition criteria for the software life cycle processes are satisfied.	8.1.c	8.2.e 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
5	Assurance is obtained that software conformity review is conducted.	8.1.d	8.2.g 8.2.h 8.3	●	●	●	●	SQA Records	11.19	②	②	②	②

TABLE A-10: CERTIFICATION LIAISON PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	Communication and understanding between the applicant and the certification authority is established.	9.a	9.1.b 9.1.c	O	O	O	O	Plan for Software Aspects of Certification	11.1	①	①	①	①
2	The means of compliance is proposed and agreement with the Plan for Software Aspects of Certification is obtained.	9.b	9.1.a 9.1.b 9.1.c	O	O	O	O	Plan for Software Aspects of Certification	11.1	①	①	①	①
3	Compliance substantiation is provided.	9.c	9.2.a 9.2.b 9.2.c	O	O	O	O	Software Accomplishment Summary Software Configuration Index	11.20 11.16	① ①	① ①	① ①	① ①

ANNEX B

ACRONYMS AND GLOSSARY OF TERMS

Acronym	Meaning
ARP	Aerospace Recommended Practice
ATM	Air Traffic Management
CAST	Certification Authorities Software Team
CC1	Control Category 1
CC2	Control Category 2
CNS	Communication, Navigation and Surveillance
COTS	Commercial-Off-The-Shelf
CRC	Cyclic Redundancy Check
DO	Document
EASA	European Aviation Safety Agency
EUROCAE	European Organization for Civil Aviation Equipment
FAA	Federal Aviation Administration
IDAL	Item Development Assurance Level
I/O	Input/Output
MC/DC	Modified Condition/Decision Coverage
PMC	Program Management Committee
PSAC	Plan for Software Aspects of Certification
RTCA	RTCA, Inc.
SAE	Society of Automotive Engineers
SC	Special Committee
SCI	Software Configuration Index
SCM	Software Configuration Management
SDP	Software Development Plan
SECI	Software Life Cycle Environment Configuration Index
SQA	Software Quality Assurance
SVP	Software Verification Plan
SW	Software
TOR	Terms of Reference
TQL	Tool Qualification Level
U.S.A.	United States of America
WG	Working Group

Glossary

These definitions are provided for the terms used in this document. If a term is not defined in this annex, it is possible that it is defined instead in the body of this document.

Activity – Tasks that provide a means of meeting the objectives.

Aeronautical data – Data used for aeronautical applications such as navigation, flight planning, flight simulators, terrain awareness, and other purposes.

Airborne – A qualifier used to denote software, equipment, or systems onboard an aircraft.

Algorithm – A finite set of well-defined rules that give a sequence of operations for performing a specific task.

Alternative method – Different approach to satisfy one or more objectives of this document.

Anomalous behavior – Behavior that is inconsistent with specified requirements.

Applicant – A person or organization seeking approval from the certification authority.

Approval – The act or instance of giving formal recognition or official sanction.

Approved source – The location of the software life cycle data to be retrieved is identified in the Software Configuration Index. The "approved source" could be a software configuration management library, an electronic archive, or an organization other than the developing organization.

Assurance – The planned and systematic actions necessary to provide adequate confidence and evidence that a product or process satisfies given requirements.

Audit – An independent examination of the software life cycle processes and their outputs to confirm required attributes.

Autocode generator – A coding tool that automatically produces Source Code or object code from low-level requirements.

Baseline – The approved, recorded configuration of one or more configuration items, that thereafter serves as the basis for further development, and that is changed only through change control procedures.

Boolean expression – An expression that results in a TRUE or FALSE value.

Boolean operator – An operator with Boolean operands that yields a Boolean result.

Certification – Legal recognition by the certification authority that a product, service, organization, or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization, or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval, or other documents as required by national laws and procedures. In particular, certification of a product involves: (a) the process of assessing the design of a product to ensure that it complies with a set of requirements applicable to that type of product so as to demonstrate an acceptable level of safety; (b) the process of assessing an individual product to ensure that it conforms with the certified type design; (c) the issuance of a certificate required by national laws to declare that compliance or conformity has been found with requirements in accordance with items (a) or (b) above.

Certification authority –Organization or person responsible within the state, country, or other relevant body, concerned with the certification or approval of a product in accordance with requirements.

NOTE 1: *A matter concerned with aircraft, engine, propeller or, by region, auxiliary power unit type certification or with equipment approval would usually be addressed by the certification authority; matters concerned with continuing airworthiness might be addressed by what would be referred to as the airworthiness authority.*

NOTE 2: *The certification authority may use a system under which the actual determinations of the compliance can be made by an approved design organization or by authorized individuals.*

Certification credit – Acceptance by the certification authority that a process, product, or demonstration satisfies a certification requirement.

Certification liaison process – The process that establishes communication, understanding, and agreements between the applicant and the certification authority.

Change control – (1) The process of recording, evaluating, approving or disapproving, and coordinating changes to configuration items after formal establishment of their configuration identification or to baselines after their establishment. (2) The systematic evaluation, coordination, approval or disapproval, and implementation of approved changes in the configuration of a configuration item after formal establishment of its configuration identification or to baselines after their establishment.

NOTE: *This term may be called "configuration control" in other industry documents.*

Code – The implementation of particular data or a particular computer program in a symbolic form, such as Source Code, object code, or machine code.

Commercial-Off-The-Shelf (COTS) software – 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.

Compacted expressions – Representations of Source Code where many constructs are combined into a single expression.

Compiler – Program that translates Source Code statements of a high level language, such as FORTRAN or Pascal, into object code.

Component – A self-contained part, combination of parts, subassemblies, or units that performs a distinct function of a system.

Condition – A Boolean expression containing no Boolean operators except for the unary operator (NOT).

Configuration identification – (1) The process of designating the configuration items in a system and recording their characteristics. (2) The approved documentation that defines a configuration item.

Configuration item – (1) One or more hardware or software components treated as a unit for configuration management purposes. (2) Software life cycle data treated as a unit for configuration management purposes.

Configuration management – (1) The process of (a) identifying and defining the configuration items of a system; (b) controlling the release and change of these items throughout the software life cycle; (c) recording and reporting the status of configuration items and Problem Reports; and (d) verifying the completeness and correctness of configuration items. (2) A discipline applying technical and administrative direction and surveillance to (a) identify and record the functional and physical characteristics of a configuration item; (b) control changes to those characteristics; and (c) record and report change control processing and implementation status.

Configuration status accounting – The recording and reporting of the information necessary to manage a configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of approved changes.

Control category – Configuration management controls placed on software life cycle data. The two categories, CC1 and CC2, define the software configuration management processes and activities applied to control software life cycle data.

Control coupling – The manner or degree by which one software component influences the execution of another software component.

Control program – A computer program designed to schedule and to supervise the execution of programs in a computer system.

Coverage analysis – The process of determining the degree to which a proposed software verification process activity satisfies its objective.

Data coupling – The dependence of a software component on data not exclusively under the control of that software component.

Data dictionary – The detailed description of data, parameters, variables, and constants used by the system.

Database – A set of data, part or the whole of another set of data, consisting of at least one file that is sufficient for a given purpose or for a given data processing system.

Deactivated code – Executable Object Code (or data) that is traceable to a requirement and, by design, is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component such as unused legacy code, unused library functions, or future growth code; or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options. The following examples are often mistakenly categorized as deactivated code but should be identified as required for implementation of the design/requirements: defensive programming structures inserted for robustness, including compiler-inserted object code for range and array index checks, error or exception handling routines, bounds and reasonableness checking, queuing controls, and time stamps.

Dead code – Executable Object Code (or data) which exists as a result of a software development error but cannot be executed (code) or used (data) in any operational configuration of the target computer environment. It is not traceable to a system or software requirement. The following exceptions are often mistakenly categorized as dead code but are necessary for implementation of the requirements/design: embedded identifiers, defensive programming structures to improve robustness, and deactivated code such as unused library functions.

Decision – A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition.

Decision coverage – Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once.

Derived requirements – 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.

Embedded identifier – Identification attributes of the software, for example, creation date, part number, linker integrity verification checksum or cyclic redundancy check (CRC), or version identification, included in the target Executable Object Code.

Emulator – A device, computer program, or system that accepts the same inputs and produces the same output as a given system using the same object code.

End-to-end numerical resolution – Measure of the numerical precision resulting from computations through the integrated system.

Equivalence class – The partition of the input domain of a program such that a test of a representative value of the class is equivalent to a test of other values of the class.

Equivalent safety – Level of safety achieved using an alternative method to satisfy both the objectives of this document and the system safety objectives.

Error – With respect to software, a mistake in requirements, design, or code.

Executable Object Code – A form of code that is directly usable by the processing unit of the target computer and is, therefore, a compiled, assembled, and linked binary image that is loaded into the target computing hardware.

Extraneous code – Code (or data) that is not traceable to any system or software requirement. An example of extraneous code is legacy code that was incorrectly retained although its requirements and test cases were removed. Another example of extraneous code is dead code.

Failure – The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.

Failure condition – The effect on the aircraft and its occupants both direct and consequential caused or contributed to by one or more failures, considering relevant adverse operational and environmental conditions. A failure condition is classified according to the severity of its effect as defined in advisory material issued by the certification authority.

Fault – A manifestation of an error in software. A fault, if it occurs, may cause a failure.

Fault tolerance – The built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults.

Formal methods – Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. A formal method is a formal analysis carried out on a formal model.

Hardware/software integration – The process of combining the software into the target computer.

High-level requirements – Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture.

Host computer – The computer on which the software is developed.

Independence – 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 equivalence to the human verification activity. (2) For the software quality assurance process, independence also includes the authority to ensure corrective action.

Integral process – A process which assists the software development processes and other integral processes and, therefore, remains active throughout the software life cycle. The integral processes are the software verification process, the software quality assurance process, the software configuration management process, and the certification liaison process.

Integrity – An attribute of the system or an item indicating that it can be relied upon to work correctly on demand.

Interrupt – A suspension of a task, such as the execution of a computer program, caused by an event external to that task, and performed in such a way that the task can be resumed.

Low-level requirements – Software requirements developed from high-level requirements, derived requirements, and design constraints from which Source Code can be directly implemented without further information.

Means of compliance – The intended method(s) to be used by the applicant to satisfy the requirements stated in the certification basis for an aircraft, engine, propeller, or, by region, auxiliary power unit. Examples include statements, drawings, analyses, calculations, testing, simulation, inspection, and environmental qualification. Advisory material issued by the certification authority is used, if appropriate.

Media – Devices or materials which act as a means of transferring or storing software, for example, programmable read-only memory, magnetic tapes or discs, and paper.

Memory device – An article of hardware capable of storing machine-readable computer programs and associated data. Examples include an integrated circuit chip, a circuit card containing integrated circuit chips, a core memory, a disk, or a magnetic tape.

Modified condition/decision coverage – Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

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.

Multiple-version dissimilar software – Two or more software components that satisfy the same functional requirements, but are intentionally different from one another. Example approaches include the use of separate development organizations or the use of different development techniques. Common mode errors may be minimized by using multiple-version dissimilar software techniques.

Object code – A low-level representation of the computer program not usually in a form directly usable by the target computer but in a form which includes relocation information in addition to the processor instruction information.

Objective – When this document is identified as a means of compliance to the regulations, the objectives are requirements that should be met to demonstrate compliance.

Parameter data item – A set of data that, when in the form of a Parameter Data Item File, influence the behavior of the software without modifying the Executable Object Code and that is managed as a separate configuration item. Examples include databases and configuration tables.

Parameter Data Item File – The representation of the parameter data item that is directly usable by the processing unit of the target computer. A Parameter Data Item File is an instantiation of the parameter data item containing defined values for each data element.

Part number – A set of numbers, letters, or other characters used to identify a configuration item.

Partitioning – A technique for providing isolation between software components to contain and/or isolate faults.

Patch – 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.

Previously developed software – Software already developed for use. This encompasses a wide range of software, including COTS software through software developed to previous or current software guidance.

Process – A collection of activities performed in the software life cycle to produce a definable output or product.

Product service history – A contiguous period of time during which the software is operated within a known environment, and during which successive failures are recorded.

Release – The act of formally making available and authorizing the use of a retrievable configuration item.

Reverification – The evaluation of the results of a modification process, for example correction of errors or the introduction of new or additional functionality, to ensure correctness and consistency with respect to the inputs and standards provided to that process.

Reverse engineering – 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.

Robustness – The extent to which software can continue to operate correctly despite abnormal inputs and conditions.

Safety monitoring – A means of protecting against specific failure conditions by directly monitoring a function for failures that would result in a failure condition.

Service experience – Intervals of time during which the software is operated within a known relevant and controlled environment, during which successive failures are recorded.

Service history data –Data collected during the service history period.

Simulator – A device, computer program, or system used during software verification, that accepts the same inputs and produces the same output as a given system, using object code that is derived from the original object code.

Single event upset – Random bit flip in data that can occur in hardware.

Software – Computer programs and, possibly, associated documentation and data pertaining to the operation of a computer system.

Software architecture – The structure of the software selected to implement the software requirements.

Software assurance – The planned and systematic actions necessary to provide confidence and evidence that a software product or process satisfies given requirements.

Software change – A modification in Source Code, object code, Executable Object Code, or its related documentation from its baseline.

Software conformity review – A review, typically conducted at the end of a software development project, for the purpose of assuring that the software life cycle processes are complete, software life cycle data is complete, and the Executable Object Code is controlled and can be regenerated.

Software development standards –Standards which define the rules and constraints for the software development processes. The software development standards include the Software Requirements Standards, the Software Design Standards, and the Software Code Standards.

Software integration – The process of combining code components.

Software level – The designation that is assigned to a software component as determined by the system safety assessment process. The software level establishes the rigor necessary to demonstrate compliance with this document.

NOTE: *Other industry documents may use a different term for the designation resulting from the system safety assessment process. One example is the term "item development assurance level" (IDAL), which for software is synonymous with the term "software level."*

Software library – A controlled repository containing a collection of software and related data and documents designed to aid in software development, use, or modification. Examples include software development library, master library, program library, and software repository.

Software life cycle – (1) An ordered collection of processes determined by an organization to be sufficient and adequate to produce a software product. (2) The period of time that begins with the decision to produce or modify a software product and ends when the product is retired from service.

Software partitioning – The process of separating, usually with the express purpose of isolating one or more attributes of the software, to prevent specific interactions and cross-coupling interference.

Software product – The set of computer programs, and associated documentation and data, designated for delivery to a user. In the context of this document, this term refers to software intended for use in airborne applications and the associated software life cycle data.

Software requirement – A description of what is to be produced by the software given the inputs and constraints. Software requirements include both high-level requirements and low-level requirements.

Software tool – A computer program used to help develop, test, analyze, produce, or modify another program or its documentation. Examples are an automated design tool, a compiler, test tools, and modification tools.

Source Code – Code written in source languages, such as assembly language and/or high level language, in a machine-readable form for input to an assembler or a compiler.

Standard – A rule or basis of comparison used to provide both guidance in and assessment of the performance of a given activity or the content of a specified data item.

Statement coverage – Every statement in the program has been invoked at least once.

NOTE: *Statement is as defined by the programming language.*

Structural coverage analysis – An evaluation of the code structure, including interfaces, exercised during requirements-based testing.

Structure – A specified arrangement or interrelation of parts to form a whole.

Supplement – Guidance used in conjunction with this document that addresses the unique nature of a specific approach, method, or technique. A supplement adds, deletes, or otherwise modifies: objectives, activities, explanatory text, and software life cycle data in this document.

System – A collection of hardware and software components organized to accomplish a specific function or set of functions.

System architecture – The structure of the hardware and the software selected to implement the system requirements.

System safety assessment process – An ongoing, systematic, comprehensive evaluation of the proposed system to show that relevant safety-related requirements are satisfied. The major activities within this process include: functional hazard assessment, preliminary system safety assessment, and system safety assessment. The rigor of the activities will depend on the criticality, complexity, novelty, and relevant service experience of the system concerned.

Task – The basic unit of work from the standpoint of a control program.

Test case – A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test procedure – Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

Testing – The process of exercising a system or system component to verify that it satisfies specified requirements and to detect errors.

Tool qualification – The process necessary to obtain certification credit for a software tool within the context of a specific airborne system.

Trace data – Data providing evidence of traceability of development and verification processes' software life cycle data without implying the production of any particular artifact. Trace data may show linkages, for example, through the use of naming conventions or through the use of references or pointers either embedded in or external to the software life cycle data.

Traceability – An association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation.

Transition criteria – The minimum conditions, as defined by the software planning process, to be satisfied to enter a process.

Type design – For the purposes of this document, the type design consists of the following: (1) The drawings and specifications, and a listing of those drawings and specifications, necessary to define the configuration and the design features of the certified product shown to comply with the requirements for that product; (2) Any other data necessary to allow, by comparison, the determination of the airworthiness of later products of the same type.

NOTE: *Use of the term "product" in this definition refers to any item for which type certificates are granted by the certification authorities, for example, aircraft, engines, propellers, and, by region, auxiliary power units.*

Unbounded recursive algorithm – 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.

User-modifiable software – Software intended for modification without review by the certification authority, the airframe manufacturer, or the equipment vendor, if within the modification constraints established during the original certification project.

Validation – The process of determining that the requirements are the correct requirements and that they are complete. The system life cycle processes may use software requirements and derived requirements in system validation.

Verification – The evaluation of the outputs of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process.

APPENDIX A

BACKGROUND OF ED-12/DO-178 DOCUMENT

1 PRIOR DOCUMENT VERSION HISTORY

In May 1980, the Radio Technical Commission for Aeronautics, now RTCA, Inc., established Special Committee 145 (SC-145), "Digital Avionics Software," to develop and document software practices that would support the development of software-based airborne systems and equipment. The European Organisation for Civil Aviation Electronics, now the European Organisation for Civil Aviation Equipment (EUROCAE), had previously established Working Group 12 (WG-12) to produce a similar document and, in October 1980, was ready to publish document ED-35, "Recommendations on Software Practice and Documentation for Airborne Systems." EUROCAE elected to withhold publication of its document and, instead, to work in concert with RTCA to develop a common guidance. SC-145 produced RTCA Document DO-178, "Software Considerations in Airborne Systems and Equipment Certification," which was approved by the RTCA Executive Committee and published by RTCA in January 1982. EUROCAE published ED-12 shortly thereafter; its technical content was identical to DO-178.

Early in 1983, the RTCA Executive Committee determined that DO-178 should be revised to reflect the experience gained in the certification of the aircraft and engines containing software based systems and equipment. It established Special Committee 152 (SC-152) for this purpose.

As a result of this committee's work, a revised RTCA document, DO-178A, "Software Considerations in Airborne Systems and Equipment Certification," was published in 1985. Shortly thereafter, EUROCAE published ED-12A (through WG-12), which was identical in technical content to DO-178A.

In early 1989, the Federal Aviation Administration (FAA) formally requested that RTCA establish a Special Committee for the review and revision of DO-178A. Since its release in 1985, the aircraft manufacturers, the avionics industry, and the certification authorities throughout the world had used DO-178A, or the equivalent EUROCAE ED-12A, as the primary source of the guidelines to determine the acceptability of airborne systems and equipment containing software.

However, rapid advances in software technology, which were not envisioned by SC-152 (or WG-12), and differing interpretations which were applied to some crucial areas, indicated that the guidance required revision. Accordingly, an RTCA Ad Hoc committee was formed with representatives from ARINC, the Airline Pilots Association, the National Business Aircraft Association, the Air Transport Association, and the FAA to consider the FAA request. The group reviewed the issues and experience associated with the application of DO-178A and concluded that a Special Committee should be authorized to revise DO-178A. The RTCA Executive Committee established Special Committee 167 (SC-167) during the autumn of 1989 to accomplish this task. EUROCAE WG-12 was re-established to work with SC-167.

The cooperative efforts of SC-167 and WG-12 culminated in the publication of RTCA document DO-178B in December 1992. Shortly thereafter, EUROCAE published ED-12B, which was identical in technical content to DO-178B.

2 RTCA/EUROCAE COMMITTEE ACTIVITIES IN THE PRODUCTION OF THIS DOCUMENT

Since 1992, the aviation industry and certification authorities around the world have used the considerations in ED-12B/DO-178B as an acceptable means of compliance for software approval in the certification of airborne systems and equipment. As experience was gained in the use of ED-12B/DO-178B, questions arose regarding the document’s content and application. Some of these questions were addressed through the work of SC-190/WG-52 in the development of ED-94B/DO-248B.

However, ED-94B/DO-248B did not contain additional guidance for use as compliance, only clarifications. Additionally, advances in hardware and software technology resulted in software development methodologies and issues which were not adequately addressed in ED-12B/DO-178B or ED-94B/DO-248B.

In 2004, the FAA and aviation industry representatives initiated a discussion with RTCA concerning the advances in software technology since 1992, when ED-12B/DO-178B was published. RTCA then requested a Software Ad Hoc committee evaluate the issues and determine the need for improved guidance in light of these advancements in technology. The Software Ad Hoc committee, which included European participants, recommended to RTCA that a special committee be formed to address these issues. In December 2004, RTCA and EUROCAE approved the sponsorship of such a joint special committee/working group, Special Committee 205/Working Group 71, SC-205/WG-71.

a. The Terms of Reference (TORs) provided to the SC/WG were:

   1. Modify ED-12B/DO-178B to become ED-12C/DO-178C, or other document number.
   2. Modify ED-94B/DO-248B to become ED-94C/DO-248C, or other document number.
   3. Consolidate Software Development Guidance.
   4. Consolidate Software Development Guidelines.
   5. Develop and document technology-specific or method-specific guidance and guidelines.
   6. Determine, document and report the effects of ED-12C/DO-178C or other modified documents to ED-109/DO-278 and recommend direction to ensure consistency.
   7. Develop and document rationale for each ED-12B/DO-178B objective.
   8. Evaluate the issues in the ‘Software Issues List.xls’ spreadsheet produced by the Software Ad Hoc committee and other identified issues. Determine ‘if’, ‘where’ and ‘how’ each issue should be addressed.
   9. Coordinate SC/WG products with software certification authorities via Certification Authorities Software Team (CAST) or other appropriate groups.
   10. Coordinate with other groups and existing organizations (for example, SAE S18, WG-63, SC-200/WG-60), as appropriate.
   11. Report to the SC/WG’s governing body the direction being taken by the committee within 6-9 months after the first SC/WG meeting.
   12. Work with RTCA and EUROCAE to explore and implement ways of expanding the usability of the deliverables (for example, hypertext electronic versions).
   13. Modify ED-109/DO-278 to become ED-109A/DO-278A, or other document number.
   14. Submit to EUROCAE and RTCA a ED-12C/DO-178C and ED-109A/DO-278A commonality analysis when documents are finalized.

b. The RTCA Program Management Committee (PMC) directed the SC/WG to maintain or adhere to the following while accomplishing the TORs:

   1. Maintain the current objective-based approach for software assurance.
   2. Maintain the technology independent nature of the ED-12B/DO-178B and ED-109/DO-278 objectives.

3. Evaluate issues as brought forth to the SC/WG. For any candidate guidance modifications determine if the issue can be satisfied first in guideline related documents.

4. Modifications to ED-12B/DO-178B and ED-109/DO-278 should:

   i. In the context of maintaining backward compatibility with ED-12B/DO-178B, make those changes to the existing text that are needed to adequately address the current states of the art and practice in software development in support of system safety, to address emerging trends, and to allow change with technology.

   ii. Consider the economic impact relative to system certification or approval without compromising system safety.

   iii. Address clear errors or inconsistencies in ED-12B/DO-178B and ED-109/DO-278.

   iv. Fill any clear gaps in ED-12B/DO-178B and ED-109/DO-278.

   v. Meet a documented need to a defined assurance benefit.

   vi. Report any proposed changes to the number of software levels or mapping of levels to hazard categories to the SC/WG's governing body and provide a documented substantiated need, at the earliest feasible opportunity. Communicate back to the SC/WG at large, any concerns of the governing body.

   vii. Ensure that all deliverables produced by the committee contain consistent and complete usability mechanisms (for example, indexes, glossaries).

c. Seven joint RTCA/EUROCAE sub-groups were formed to address the TORs:

   1. Documentation Integration.

   2. Issues and Rationale.

   3. Tool Qualification.

   4. Model-Based Development and Verification.

   5. Object-Oriented Technology.

   6. Formal Methods.

   7. Special Considerations and CNS/ATM.

d. The cooperative efforts of SC-205 and WG-71 culminated in the publication of RTCA document DO-178C and EUROCAE document ED-12C.

3. SUMMARY OF DIFFERENCES BETWEEN ED-12C AND ED-12B

ED-12C is an update to ED-12B. The ED-12C updates fall into a variety of categories:

a. Errors and Inconsistencies: ED-12C addressed ED-12B's known errors and inconsistencies. For example, ED-12C has addressed the errata of ED-12B and has removed inconsistencies between the different tables of ED-12B Annex A.

b. Consistent Terminology: ED-12C addressed issues regarding the use of specific terms such as "guidance", "guidelines", "purpose", "goal", "objective", and "activity" by changing the text so that the use of those terms is consistent throughout the document.

c. Wording Improvements: ED-12C made wording improvements throughout the document. All such changes were made simply to make the document more precise; they were not meant to change the original intent of ED-12B.

d. Objectives and Activities: ED-12C reinforced the point that, in order to fully understand the recommendations, the full body of this document should be considered. For example, Annex A now includes references to each activity as well as to each objective; and section 1.4, titled “How to Use This Document” reinforces the point that activities are a major part of the overall guidance.

e. Supplements: ED-12C recognized that new software development techniques may result in new issues. Rather than expanding text to account for all the current software development techniques (and being revised yet again to account for future techniques), ED-12C acknowledged that one or more supplements may be used in conjunction with ED-12C to modify the guidance for specific techniques. Section 12 was impacted since planned supplements more completely address those specific techniques.

f. Tool Qualification (Section 12.2): The terms "development tool" and "verification tool" are replaced by three tool qualification criteria that determine the applicable tool qualification level (TQL) in regard of the software level. The guidance to qualify a tool is removed in ED-12C, but provided in a domain independent, external document, referenced in section 12.2

g. Coordinated System/Software Aspects: ED-12C updated section 2, which provides system aspects relating to software development, to reflect current system practices. The updates were based upon coordination with the committees which were updating ED-79/ARP4754 (system-level guidance) at the same time SC-205/WG-71 was updating ED-12B (software-level guidance).

h. ED-12B “Hidden” Objectives: ED-12C added the so-called “hidden objectives” to Annex A:

    1. A means for detecting additional code that is not directly traceable to the Source Code and a means to ensure its verification coverage are defined (see objective 9 of Table A-7).

    2. Assurance is obtained that software plans and standards are developed and reviewed for consistency (see objective 1 of Table A-9).

        i. General Topics: ED-12C addressed some general topics that resulted in changes to several sections of the document. The topics included a variety of subjects such as applicant’s oversight of suppliers, parameter data items, and traceability. In addressing these topics, two additional objectives were added to Annex A:

    1. Parameter Data Item File is correct and complete (see objective 8 of Table A-5).

    2. Verification of Parameter Data Item File is achieved (see objective 9 of Table A-5).

Also, Trace Data was identified as software life cycle data (see 11.21, Table A-2, and Table A-6).

j. ED-12B Gaps and Clarifications: ED-12C addressed several specific issues that resulted in change to only one or two paragraphs. Each such change may have an impact upon the applicant as these changes either addressed clear gaps in ED-12B or clarified guidance that was subject to differing interpretations.

    1. Examples of gaps addressed include:

        i. The “Modified Condition/Decision Coverage” (MC/DC) definition changed. Masking MC/DC and Short Circuit, as well as ED-12B’s interpretation of MC/DC (often termed Unique-Cause MC/DC), are now allowed (see Glossary).

ii. Derived requirements should now be provided to the system processes, including the system safety assessment process, rather than just provided to the system safety assessment process (see 5.1.1.b and 5.2.1.b).

2. Examples of clarifications include:

i. Clarified that the structural coverage analysis of data and control coupling between code components should be achieved by assessing the results of the requirements-based tests (see 6.4.4.2.c).

ii. Clarified that all tests added to achieve structural coverage are based on requirements (see 6.4.4.2.d).

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)

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)

EUROCAE Representative:

Gilbert Amato	EUROCAE (until September 2009)
Roland Mallwitz	EUROCAE (from October 2009)

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